МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«Казанский (Приволжский) федеральный университет»
Елабужский институт (филиал) КФУ

**УТВЕРЖДАЮ**
Заместитель директора по
образовательной деятельности
_____ С.Ю. Бахвалов
«_19_» _____мая_____ 2025 г.
МП

**Программа дисциплины (модуля)**
*Промышленное программирование*

Направление подготовки/специальность: 15.03.06 Мехатроника и робототехника

Направленность (профиль) подготовки (специальности): Физические основы мехатроники и робототехники

Квалификация: бакалавр

Форма обучения: очная

Язык обучения: русский

Год начала обучения по образовательной программе: - 2025

**Содержание**

1. Перечень планируемых результатов обучения по дисциплине (модулю), соотнесенных с планируемыми результатами освоения ОПОП ВО

2. Место дисциплины (модуля) в структуре ОПОП ВО

3. Объем дисциплины (модуля) в зачетных единицах с указанием количества часов, выделенных на контактную работу обучающихся с преподавателем (по видам учебных занятий) и на самостоятельную работу обучающихся

4. Содержание дисциплины (модуля), структурированное по темам (разделам) с указанием отведенного на них количества академических часов и видов учебных занятий

4.1. Структура и тематический план контактной и самостоятельной работы по дисциплине (модулю)

4.2. Содержание дисциплины (модуля)

5. Перечень учебно-методического обеспечения для самостоятельной работы обучающихся по дисциплине (модулю)

6. Фонд оценочных средств по дисциплине (модулю)

7. Перечень литературы, необходимой для освоения дисциплины (модуля)

8. Перечень ресурсов информационно-телекоммуникационной сети "Интернет", необходимых для освоения дисциплины (модуля)

9. Методические указания для обучающихся по освоению дисциплины (модуля)

10. Перечень информационных технологий, используемых при осуществлении образовательного процесса по дисциплине (модулю), включая перечень программного обеспечения и информационных справочных систем (при необходимости)

11. Описание материально-технической базы, необходимой для осуществления образовательного процесса по дисциплине (модулю)

12. Средства адаптации преподавания дисциплины (модуля) к потребностям обучающихся инвалидов и лиц с ограниченными возможностями здоровья

13. Приложение №1. Фонд оценочных средств

14. Приложение №2. Перечень литературы, необходимой для освоения дисциплины (модуля)

15. Приложение №3. Перечень информационных технологий, используемых для освоения дисциплины (модуля), включая перечень программного обеспечения и информационных справочных систем

Программу дисциплины разработал(а)(и) старший преподаватель Галиев Н.М. (Кафедра математики и прикладной информатики)

**1. Перечень планируемых результатов обучения по дисциплине (модулю), соотнесенных с планируемыми результатами освоения ОПОП ВО**

Обучающийся, освоивший дисциплину (модуль), должен обладать следующими компетенциями:

| Шифр компетенции | Расшифровка приобретаемой компетенции |
|---|---|
| ПК-2 | Способен разрабатывать, отлаживать, внедрять и сопровождать программное обеспечение мехатронных и робототехнических систем |
| ПК-2.1 | Знать способы разработки, отладки и сопровождения программного обеспечения для мехатронных и робототехнических систем |
| ПК-2.2 | Уметь разрабатывать, отлаживать и сопровождать программное обеспечение для мехатронных и робототехнических систем |
| ПК-2.2 | Владеть навыками разработки, отладки и сопровождения программного обеспечения для мехатронных и робототехнических систем |

Обучающийся, освоивший дисциплину (модуль):

Должен знать:
принципы и основные элементы объектной модели разработки программного обеспечения для мехатронных и робототехнических систем.

Должен уметь:
проектировать и разрабатывать прототипы программных компонентов для мехатронных и робототехнических систем, учитывая принципы объектного подхода (G4, SOLID).

Должен владеть:
способностью разрабатывать пользовательские приложения для мехатронных и робототехнических систем в рамках объектного подхода.

**2. Место дисциплины (модуля) в структуре ОПОП ВО**

Данная дисциплина (модуль) включена в Блок 1 "Дисциплины (модули)" Б1.В.ДВ.03.01 основной профессиональной образовательной программы 15.03.06 «Мехатроника и робототехника» (Физические основы мехатроники и робототехники) и относится к дисциплинам по выбору, части, формируемой участниками образовательных отношений.
Осваивается на 3 курсе в 6 семестре.

**3. Объем дисциплины (модуля) в зачетных единицах с указанием количества часов, выделенных на контактную работу обучающихся с преподавателем (по видам учебных занятий) и на самостоятельную работу обучающихся**

Общая трудоемкость дисциплины составляет 4 зачетных(ые) единиц(ы) на 144 часа(ов).
Контактная работа - 30 часа(ов), в том числе лекции - 10 часа(ов), практические занятия - 0 часа(ов), лабораторные работы - 20 часа(ов), контроль самостоятельной работы - 0 часа(ов).
Самостоятельная работа - 78 часа(ов).
Контроль (зачёт / экзамен) - 36 часа(ов).
Форма промежуточного контроля дисциплины: экзамен в 6 семестре

**4. Содержание дисциплины (модуля), структурированное по темам (разделам) с указанием отведенного на них количества академических часов и видов учебных занятий**

**4.1 Структура и тематический план контактной и самостоятельной работы по дисциплине (модулю)**

| N | Разделы дисциплины / модуля | Семестр | Виды и часы контактной работы, их трудоемкость (в часах) | | | Самостоятельная работа |
|---|---|---|---|---|---|---|
| | | | Лекции | Практические занятия | Лабораторные работы | |
| 1. | Тема 1. Основные концепции C++. | 8 | 2 | 0 | 6 | 18 |
| 2. | Тема 2. Базовые и продвинутые методы управления ресурсами на C++. | 8 | 2 | 0 | 6 | 20 |
| 3. | Тема 3. Применение шаблонов C++ для проектирования программ. Разработка и проектирование многопоточных программ. | 8 | 4 | 0 | 4 | 20 |
| 4. | Тема 4. Разработка и проектирование графического пользовательского интерфейса. | 8 | 2 | 0 | 4 | 20 |
| | Итого: 144 часа (из них 36 часов контроль) | | 10 | 0 | 20 | 78 |

**4.2 Содержание дисциплины (модуля)**

**Тема 1. Основные концепции C++.**

Развитие и стандартизация языка C++. Типы данных. Наследование. Полиморфизм. Инкапсуляция. Специальные функции. Шаблоны. Стандартная библиотека.

**Тема 2. Базовые и продвинутые методы управления ресурсами на C++**

Концепции C++. Основные стратегии копирования-владения. Стратегия запрета копирования. Стратегия исключительного владения. Стратегия глубокого копирования. Стратегия совместного владения. Определение функции обмена состояниями для класса. Удаление промежуточных копий компилятором. Реализация семантики перемещения. Возможные варианты реализации стратегии совместного владения. Стратегия исключительного владения и семантика перемещения. Жизненный цикл ресурса и объекта-владельца ресурса

**Тема 3. Применение шаблонов C++ для проектирования программ. Разработка и проектирование многопоточных программ**

Шаблон C++. Обзор промышленных программных библиотек. Многопоточное программирование. Управление потоками. Разделение данных между потоками. Синхронизация параллельных операций. Проектирование параллельных программ. Продвинутое управление потоками. Тестирование и отладка многопоточных приложений.

**Тема 4. Разработка и проектирование графического пользовательского интерфейса**

Графический пользовательский интерфейс. Проектирование графического пользовательского интерфейса на C++. Инструментарий разработчика. Разбор существующего проекта с открытым исходным кодом.

**5. Перечень учебно-методического обеспечения для самостоятельной работы обучающихся по дисциплине (модулю)**

Самостоятельная работа обучающихся выполняется по заданию и при методическом руководстве преподавателя, но без его непосредственного участия. Самостоятельная работа подразделяется на самостоятельную работу на аудиторных занятиях и на внеаудиторную самостоятельную работу. Самостоятельная работа обучающихся включает как полностью самостоятельное освоение отдельных тем (разделов) дисциплины, так и проработку тем (разделов), осваиваемых во время аудиторной работы. Во время самостоятельной работы

обучающиеся читают и конспектируют учебную, научную и справочную литературу, выполняют задания, направленные на закрепление знаний и отработку умений и навыков, готовятся к текущему и промежуточному контролю по дисциплине.

Организация самостоятельной работы обучающихся регламентируется нормативными документами, учебно-методической литературой и электронными образовательными ресурсами, включая:

Порядок организации и осуществления образовательной деятельности по образовательным программам высшего образования – программам бакалавриата, программам специалитета, программам магистратуры (утвержденный приказом Министерства науки и высшего образования Российской Федерации от 6 апреля 2021 года № 245)

Порядок организации и осуществления образовательной деятельности по образовательным программам высшего образования - программам бакалавриата, программам специалитета, программам магистратуры (утвержден приказом Министерства образования и науки Российской Федерации от 5 апреля 2017 года №301)

Письмо Министерства образования Российской Федерации №14-55-996ин/15 от 27 ноября 2002 г. "Об активизации самостоятельной работы студентов высших учебных заведений"

Устав федерального государственного автономного образовательного учреждения "Казанский (Приволжский) федеральный университет"

Правила внутреннего распорядка федерального государственного автономного образовательного учреждения высшего профессионального образования "Казанский (Приволжский) федеральный университет"

Локальные нормативные акты Казанского (Приволжского) федерального университета

**6. Фонд оценочных средств по дисциплине (модулю)**

Фонд оценочных средств по дисциплине (модулю) включает оценочные материалы, направленные на проверку освоения компетенций, в том числе знаний, умений и навыков. Фонд оценочных средств включает оценочные средства текущего контроля и оценочные средства промежуточной аттестации.

В фонде оценочных средств содержится следующая информация:

- соответствие компетенций планируемым результатам обучения по дисциплине (модулю);
- критерии оценивания сформированности компетенций;
- механизм формирования оценки по дисциплине (модулю);
- описание порядка применения и процедуры оценивания для каждого оценочного средства;
- критерии оценивания для каждого оценочного средства;
- содержание оценочных средств, включая требования, предъявляемые к действиям обучающихся, демонстрируемым результатам, задания различных типов.

Фонд оценочных средств по дисциплине находится в Приложении 1 к программе дисциплины (модулю).

**7. Перечень литературы, необходимой для освоения дисциплины (модуля)**

Освоение дисциплины (модуля) предполагает изучение учебной литературы. Литература может быть доступна обучающимся в одном из двух вариантов (либо в обоих из них):

- в электронном виде - через электронные библиотечные системы на основании заключенных КФУ договоров с правообладателями;
- в печатном виде - в Научной библиотеке Елабужского института КФУ. Обучающиеся получают учебную литературу на абонементе по читательским билетам в соответствии с правилами пользования Научной библиотекой.

Электронные издания доступны дистанционно из любой точки при введении обучающимся своего логина и пароля от личного кабинета в системе "Электронный университет". При использовании печатных изданий библиотечный фонд должен быть укомплектован ими из расчета не менее 0,25 экземпляра на каждого обучающегося из числа лиц, одновременно осваивающих данную дисциплину

Перечень литературы, необходимой для освоения дисциплины (модуля), находится в Приложении 2 к рабочей программе дисциплины. Он подлежит обновлению при изменении условий договоров КФУ с правообладателями электронных изданий и при изменении комплектования фондов Научной библиотеки Елабужского института КФУ.

**8. Перечень ресурсов информационно-телекоммуникационной сети "Интернет", необходимых для освоения дисциплины (модуля)**

Многопоточное программирование - http://ders.stml.net/cpp/mtprog/mtprog.html

Руководство. Управление ресурсами (C++) - https://docs.microsoft.com/ru-ru/cpp/windows/how-to-copy-resources?view=msvc-160

Основы программирования - https://purecodecpp.com/

**9. Методические указания для обучающихся по освоению дисциплины (модуля)**

| Вид работ | Методические рекомендации |
|---|---|
| лекции | Лекционные занятия проводятся с использованием интерактивных технологий и предполагают активное участие студентов. Для подготовки к занятиям рекомендуется выделять в материале проблемные вопросы, затрагиваемые преподавателем в лекции, и группировать информацию вокруг них. Желательно выделять в используемой литературе постановки вопросов, на которые разными авторам могут быть даны различные ответы. На основании постановки таких вопросов следует собирать аргументы в пользу различных вариантов решения поставленных проблем. |
| лабораторные работы | Лабораторные занятия — это одна из разновидностей практического занятия, являющаяся эффективной формой учебных занятий в организации высшего образования. Лабораторные занятия имеют выраженную специфику в зависимости от учебной дисциплины, углубляют и закрепляют теоретические знания. На этих занятиях студенты осваивают конкретные методы изучения дисциплины, обучаются экспериментальным способам анализа, умению работать с приборами и современным оборудованием. Лабораторные занятия дают наглядное представление об изучаемых явлениях и процессах, студенты осваивают постановку и ведение эксперимента, учатся умению наблюдать, оценивать полученные результаты, делать выводы и обобщения. Отчёт по итогам выполненных лабораторных работ выполняется на листах белой бумаги формата А4 в печатном или рукописном виде. При оформлении отчёта используется сквозная нумерация страниц, считая титульный лист первой страницей. Номер страницы на титульном листе не ставится. Номера страницы ставятся по центру вверху. При оформлении отчёта в печатном виде желательно соблюдать следующие требования. Для заголовков: полужирный шрифт, 14 пт, центрированный. Для основного текста: нежирный шрифт, 14 пт, выравнивание по ширине. Во всех случаях тип шрифта - Times New Roman, отступ абзаца 1.25 см, полуторный междустрочный интервал. Поля: левое - 3 см, правое - 1 см, верхнее и нижнее - 2 см. Отчет должен содержать следующие элементы: 1) Титульный лист с обязательным указанием варианта; 2) Цель работы; 3) Задание; 4) Основная часть; 5) Вывод |
| самостоя-тельная работа | Самостоятельная работа студентов по дидактической сути представляет собой комплекс условий обучения, организуемых преподавателем и направленных на самоподготовку учащихся. Учебная деятельность протекает без непосредственного участия преподавателя и заключается в проработке лекционного материала, подготовке к лабораторным занятиям; изучении учебной литературы из основного и дополнительного списка. |
| экзамен | Экзамен по курсу проводится по билетам. В каждом билете один теоретический вопрос и одна задача. После ответа студенту могут быть заданы дополнительные вопросы, как по материалам билета, так и по основным определениям курса в целом.<br>При подготовке к экзамену необходимо опираться, прежде всего, на конспекты лекций и рекомендованные источники информации, весь объём работы рекомендуется распределять равномерно по дням, отведённым для подготовки к экзамену и контролировать каждый день выполнения работы. |

**10. Перечень информационных технологий, используемых при осуществлении образовательного процесса по дисциплине (модулю), включая перечень программного обеспечения и информационных справочных систем (при необходимости)**

Перечень информационных технологий, используемых при осуществлении образовательного процесса по дисциплине (модулю), включая перечень программного обеспечения и информационных справочных систем, представлен в Приложении 3 к рабочей программе дисциплины (модуля).

**11. Описание материально-технической базы, необходимой для осуществления образовательного процесса по дисциплине (модулю)**

Материально-техническое обеспечение образовательного процесса по дисциплине (модулю) включает в себя следующие компоненты:

Учебная аудитория для проведения учебных занятий лекционного типа, семинарского типа, групповых и индивидуальных консультаций, текущего контроля и промежуточной аттестации № 61

Комплект мебели для преподавателя – 1 шт., посадочные места для обучающихся – 30 шт., одноместные столы – 12 шт., компьютерные столы – 18 шт., компьютеры – 19 шт., интерактивная панель – 1 шт., меловая доска настенная – 1 шт., выход в интернет, внутривузовская компьютерная сеть, доступ в электронную информационно-образовательную среду.

Помещение для самостоятельной работы № 10

Посадочные места для пользователей – 28 шт., металлические двусторонние стеллажи для книг – 11 шт., книжный шкаф открытый – 5 шт., проектор – 1 шт., ноутбуки для пользователей – 11 шт., шкаф каталожный – 8 шт., шкаф для одежды – 1 шт., ксерокс – 1 шт., рабочий стол библиотекаря – 1 шт., компьютер библиотекаря – 1 шт., вешалка для одежды – 1 шт., жалюзи рулонные «Омега» с фотопечатью – 4 шт., стенд настенный (бронированное стекло) – 4 шт., шкаф-витрина встроенный в арку – 2 шт., шкаф-витрина стеклянный – 2 шт., стеллаж трубчатый с деревянными полками – 2 шт., рабочий стол для инвалидов и лиц с ОВЗ – 2 шт., стол СИ-1 рабочий для инвалидов-колясочников – 1 шт., компьютер– 2 шт., наушники – 2 шт., устройство «Говорящая книга» (тифлоплеер) – 2 шт., видеоувеличитель – 2 шт., радиокласс – 1 шт., портативный тактильный дисплей - 1 шт., сканирующая читающая машина - 1 шт., сканер – 1 шт., веб-камера – 1 шт., выход в интернет, внутривузовская компьютерная сеть, доступ в электронную информационно-образовательную среду.

**12. Средства адаптации преподавания дисциплины к потребностям обучающихся инвалидов и лиц с ограниченными возможностями здоровья**

При необходимости в образовательном процессе применяются следующие методы и технологии, облегчающие восприятие информации обучающимися инвалидами и лицами с ограниченными возможностями здоровья:

- создание текстовой версии любого нетекстового контента для его возможного преобразования в альтернативные формы, удобные для различных пользователей;

- создание контента, который можно представить в различных видах без потери данных или структуры, предусмотреть возможность масштабирования текста и изображений без потери качества, предусмотреть доступность управления контентом с клавиатуры;

- создание возможностей для обучающихся воспринимать одну и ту же информацию из разных источников - например, так, чтобы лица с нарушениями слуха получали информацию визуально, с нарушениями зрения - аудиально;

- применение программных средств, обеспечивающих возможность освоения навыков и умений, формируемых дисциплиной, за счёт альтернативных способов, в том числе виртуальных лабораторий и симуляционных технологий;

- применение дистанционных образовательных технологий для передачи информации, организации различных форм интерактивной контактной работы обучающегося с преподавателем, в том числе вебинаров, которые могут быть использованы для проведения виртуальных лекций с возможностью взаимодействия всех участников дистанционного обучения, проведения семинаров, выступления с докладами и защиты выполненных работ, проведения тренингов, организации коллективной работы;

- применение дистанционных образовательных технологий для организации форм текущего и промежуточного контроля;

- увеличение продолжительности сдачи обучающимся инвалидом или лицом с ограниченными возможностями здоровья форм промежуточной аттестации по отношению к установленной продолжительности их сдачи:

- продолжительности сдачи зачёта или экзамена, проводимого в письменной форме, - не более чем на 90 минут;

- продолжительности подготовки обучающегося к ответу на зачёте или экзамене, проводимом в устной форме, - не более чем на 20 минут;

- продолжительности выступления обучающегося при защите курсовой работы - не более чем на 15 минут.

Программа составлена в соответствии с требованиями ФГОС ВО и учебным планом по направлению 15.03.06 «Мехатроника и робототехника» и профилю подготовки "Физические основы мехатроники и робототехники".

*Приложение №1*
*к рабочей программе дисциплины (модуля)*
*Б1.В.ДВ.03.01 Промышленное программирование*

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
"Казанский (Приволжский) федеральный университет"
Елабужский институт (филиал) КФУ

**Фонд оценочных средств по дисциплине (модулю)**
*Промышленное программирование*

Направление подготовки: 15.03.06 Мехатроника и робототехника
Профиль подготовки: Физические основы мехатроники и робототехники
Квалификация выпускника: бакалавр
Форма обучения: очная
Язык обучения: русский
Год начала обучения по образовательной программе: 2025

**СОДЕРЖАНИЕ**

1. Соответствие компетенций планируемым результатам обучения по дисциплине (модулю)
2. Критерии оценивания сформированности компетенций
3. Распределение оценок за формы текущего контроля и промежуточную аттестацию
4. Оценочные средства, порядок их применения и критерии оценивания
4.1. Оценочные средства текущего контроля
4.1.1. Лабораторные работы
4.1.1.1. Порядок проведения.
4.1.1.2 Критерии оценивания
4.1.1.3. Содержание оценочного средства
4.1.2. Компьютерная программа
4.1.2.1. Порядок проведения.
4.1.2.2 Критерии оценивания
4.1.2.3. Содержание оценочного средства
4.2. Оценочные средства промежуточной аттестации экзамен
4.2.1. Устный или письменный ответ на вопрос
4.2.1.1. Порядок проведения.
4.2.1.2. Критерии оценивания.
4.2.1.3. Оценочные средства.
4.2.2. Практическое задание
4.2.2.1. Порядок проведения.
4.2.2.2. Критерии оценивания.
4.2.2.3. Оценочные средства.

**1. Соответствие компетенций планируемым результатам обучения по дисциплине (модулю)**

| Код и наименование компетенции | Индикаторы достижения компетенций для данной дисциплины | Оценочные средства текущего контроля и промежуточной аттестации |
|---|---|---|
| ПК-2 Способен разрабатывать, отлаживать, внедрять и сопровождать программное обеспечение мехатронных и робототехнических систем | Знать принципы и основные элементы объектной модели разработки программного обеспечения для мехатронных и робототехнических систем<br><br>Уметь проектировать и разрабатывать прототипы программных компонентов для мехатронных и робототехнических систем, учитывая принципы объектного подхода (G4, SOLID).<br><br>Владеть способностью разрабатывать пользовательские приложения для мехатронных и робототехнических систем в рамках объектного подхода | Компьютерная программа по темам<br>Тема 1. Основные концепции С++.<br>Тема 2. Базовые и продвинутые методы управления ресурсами на С++.<br>Тема 3. Применение шаблонов С++ для проектирования программ. Разработка и проектирование многопоточных программ.<br>Тема 4. Разработка и проектирование графического пользовательского интерфейса.<br><br>Лабораторные работы по темам<br>Тема 1. Основные концепции С++.<br>Тема 2. Базовые и продвинутые методы управления ресурсами на С++.<br>Тема 3. Применение шаблонов С++ для проектирования программ. Разработка и проектирование многопоточных программ.<br>Тема 4. Разработка и проектирование графического пользовательского интерфейса.<br><br>Промежуточная аттестация:<br>*экзамен* |

**2. Критерии оценивания сформированности компетенций**

| Компетенция | Зачтено | | | Не зачтено |
|---|---|---|---|---|
| | **Высокий уровень (отлично) (86-100 баллов)** | **Средний уровень (хорошо) (71-85 баллов)** | **Низкий уровень (удовлетворительно) (56-70 баллов)** | **Ниже порогового уровня (неудовлетворительно) (0-55 баллов)** |
| *ПК-2* | Знает принципы и основные элементы объектной модели разработки программного обеспечения для мехатронных и робототехнических систем | Знает принципы и основные элементы объектной модели разработки программного обеспечения для мехатронных и робототехнических систем. Допускает незначительные ошибки при ответе на вопрос или решении поставленной задачи | Знает некоторые принципы и основные элементы объектной модели разработки программного обеспечения для мехатронных и робототехнических систем. Допускает типичные ошибки при ответе на вопрос или решении поставленной задачи | Не знает принципы и основные элементы объектной модели разработки программного обеспечения для мехатронных и робототехнических систем |
| | Умеет проектировать и разрабатывать прототипы программных компонентов для мехатронных и робототехнических систем, учитывая принципы объектного подхода (G4, SOLID). | Умеет проектировать и разрабатывать прототипы программных компонентов для мехатронных и робототехнических систем, учитывая принципы объектного подхода (G4, SOLID). Допускает незначительные ошибки при ответе на вопрос или решении поставленной задачи | Умеет проектировать и разрабатывать прототипы программных компонентов для мехатронных и робототехнических систем, учитывая принципы объектного подхода (G4, SOLID). Допускает типичные ошибки при ответе на вопрос или решении | Не умеет проектировать и разрабатывать прототипы программных компонентов для мехатронных и робототехнических систем, учитывая принципы объектного подхода (G4, SOLID). |

| | | | | |
|---|---|---|---|---|
| | | | поставленной задачи | |
| | Владеет способностью разрабатывать пользовательские приложения для мехатронных и робототехнических систем в рамках объектного подхода | Владеет способностью разрабатывать пользовательские приложения для мехатронных и робототехнических систем в рамках объектного подхода. Допускает незначительные ошибки при ответе на вопрос или решении поставленной задачи | Владеет способностью разрабатывать пользовательские приложения для мехатронных и робототехнических систем в рамках объектного подхода. Допускает типичные ошибки при ответе на вопрос или решении поставленной задачи | Не владеет способностью разрабатывать пользовательские приложения для мехатронных и робототехнических систем в рамках объектного подхода |

**3. Распределение оценок за формы текущего контроля и промежуточную аттестацию**

**Текущий контроль:**
Лабораторные работы по темам
Тема 1. Основные концепции С++.
Тема 2. Базовые и продвинутые методы управления ресурсами на С++.
Тема 3. Применение шаблонов С++ для проектирования программ. Разработка и проектирование многопоточных программ.
Тема 4. Разработка и проектирование графического пользовательского интерфейса.
Максимальное количество баллов по БРС - 30.

Компьютерная программа по темам
Тема 1. Основные концепции С++.
Тема 2. Базовые и продвинутые методы управления ресурсами на С++.
Тема 3. Применение шаблонов С++ для проектирования программ. Разработка и проектирование многопоточных программ.
Тема 4. Разработка и проектирование графического пользовательского интерфейса.

Максимальное количество баллов по БРС - 20.

Итого 30+20=50 баллов

**Промежуточная аттестация** - экзамен – 50 баллов
Промежуточная аттестация проводится после завершения изучения дисциплины или ее части в форме, определяемой учебным планом образовательной программы с целью оценить работу обучающегося, степень усвоения теоретических знаний, уровень сформированности компетенций.
Преподаватель, принимающий экзамен, обеспечивает случайное распределение вариантов экзаменационных заданий между обучающимися с помощью билетов и/или с применением компьютерных технологий; вправе задавать обучающемуся дополнительные вопросы и давать дополнительные задания помимо тех, которые указаны в билете.

Устный или письменный ответ – 20 баллов.
Практическое задание – 30 баллов.
Итого 20+30=50 баллов.
Общее количество баллов по дисциплине за текущий контроль и промежуточную аттестацию: 50+50=100 баллов.

Соответствие баллов и оценок:
Для экзамена:
86-100 – отлично
71-85 – хорошо
56-70 – удовлетворительно
0-55 – неудовлетворительно

**4. Оценочные средства, порядок их применения и критерии оценивания**
**4.1. Оценочные средства текущего контроля**

**4.1.1. Лабораторные работы**

Тема 1. Основные концепции C++.

Тема 2. Базовые и продвинутые методы управления ресурсами на C++.

Тема 3. Применение шаблонов C++ для проектирования программ. Разработка и проектирование многопоточных программ.

Тема 4. Разработка и проектирование графического пользовательского интерфейса.

**4.1.1.1. Порядок проведения.**

Лабораторные работы проводятся в часы аудиторной работы.

Перед выполнением каждой работы студенты-бакалавры должны проработать соответствующий материал, используя конспекты теоретических занятий, периодические издания, учебно-методические пособия и учебники.

По окончании занятий студенты оформляют отчет по каждой работе, соблюдая следующую форму:

- Наименование темы;
- Цель работы;
- Задание и содержание выполненной работы,
- Письменные ответы на контрольные вопросы.
- Выводы по проделанной работе.
- Список использованных источников.

**4.1.1.2 Критерии оценивания**

**27-30 баллов ставится, если обучающийся:**

Правильно выполнил все задания. Продемонстрировал высокий уровень владения материалом. Проявлены превосходные способности применять знания и умения к выполнению конкретных заданий.

**22-26 баллов ставится, если обучающийся:**

Правильно выполнил большую часть заданий. Присутствуют незначительные ошибки. Продемонстрирован хороший уровень владения материалом. Проявлены средние способности применять знания и умения к выполнению конкретных заданий.

**18-21 баллов ставится, если обучающийся:**

Задания выполнил более чем наполовину. Присутствуют серьёзные ошибки. Продемонстрирован удовлетворительный уровень владения материалом. Проявлены низкие способности применять знания и умения к выполнению конкретных заданий.

**0-17 баллов ставится, если обучающийся:**

Задания выполнил менее чем наполовину. Продемонстрирован неудовлетворительный уровень владения материалом. Проявлены недостаточные способности применять знания и умения к выполнению конкретных заданий.

**4.1.1.3. Содержание оценочного средства**

**Лабораторная работа 1. Умные указатели**

**Задание №1**

Если в вашем классе есть умный указатель в качестве элемента вашего класса, то почему вы должны стараться избегать динамического выделения объектов этого класса?

**Ответ**:

Если в моем классе есть умный указатель в качестве элемента, я стараюсь избегать динамического выделения объектов этого класса, чтобы предотвратить проблемы с управлением памятью и множественным владением. Умные указатели уже автоматически управляют временем жизни объектов, которые они хранят. Если я динамически выделю объекты, это может привести к утечкам памяти или двойному освобождению памяти, поскольку умные указатели могут попытаться освободить память, которая была уже освобождена вручную. Вместо этого я предпочитаю использовать объекты на стеке или предоставлять их через умные указатели, которые гарантируют правильное управление памятью.

**Задание №2**

Измените следующую программу, заменив обычный указатель на умный указатель std::unique_ptr, где это необходимо:

```
#include <iostream>

class Fraction
{
private:
    int m_numerator = 0;
    int m_denominator = 1;

public:
    Fraction(int numerator = 0, int denominator = 1) :
        m_numerator(numerator), m_denominator(denominator)
    {
```

```
    }

    friend std::ostream& operator<<(std::ostream& out, const Fraction &f1)
    {
        out << f1.m_numerator << "/" << f1.m_denominator;
        return out;
    }
};

void printFraction(const Fraction* const ptr)
{
    if (ptr)
        std::cout << *ptr;
}

int main()
{
    Fraction *ptr = new Fraction(7, 9);

    printFraction(ptr);

    delete ptr;

    return 0;
}
```

**Ответ:**

программа, в которой обычный указатель заменен на std::unique_ptr:

```
#include <iostream>
#include <memory> // Для std::unique_ptr

class Fraction
{
private:
    int m_numerator = 0;
    int m_denominator = 1;

public:
    Fraction(int numerator = 0, int denominator = 1) :
        m_numerator(numerator), m_denominator(denominator)
    {
    }

    friend std::ostream& operator<<(std::ostream& out, const Fraction &f1)
    {
        out << f1.m_numerator << "/" << f1.m_denominator;
        return out;
    }
};

void printFraction(const std::unique_ptr<Fraction>& ptr)
{
    if (ptr)
        std::cout << *ptr;
}

int main()
{
    std::unique_ptr<Fraction> ptr = std::make_unique<Fraction>(7, 9);

    printFraction(ptr);
```

```
    // Удалять ptr не нужно - unique_ptr автоматически освобождает память при
выходе из области видимости.

    return 0;
}
```

- Обычный указатель был заменен на std::unique_ptr, что устраняет необходимость в явном освобождении памяти и делает управление ресурсами более безопасным и менее подверженным ошибкам.

**Лабораторная работа 2. Управление памятью**

Задание. Напишите программу, которая выделяет 1000 блоков памяти по 120 байт каждый и выводит статистику (malloc_stats). Сколько блоков памяти было выделено в разделе «малых блоков»? Измените программу так, чтобы размер блока рассчитывался как i*1024, где i – номер итерации цикла. Повторите эксперимент и ответьте на тот же вопрос.

**Ответ:**

Вот пример кода, который выполняет поставленную задачу.

Программа для первого случая (1000 блоков по 120 байт)

```
#include <iostream>
#include <cstdlib>

int main() {
    const int numBlocks = 1000;
    const size_t blockSize = 120;
    void* blocks[numBlocks];

    // Выделение памяти
    for (int i = 0; i < numBlocks; ++i) {
        blocks[i] = malloc(blockSize);
    }

    // Вывод статистики о выделенной памяти
    malloc_stats();

    // Освобождение памяти
    for (int i = 0; i < numBlocks; ++i) {
        free(blocks[i]);
    }

    return 0;
}
```

Программа для второго случая (размер блока i * 1024)

```
#include <iostream>
#include <cstdlib>

int main() {
    const int numBlocks = 1000;
    void* blocks[numBlocks];

    // Выделение памяти с изменяющимся размером блоков
    for (int i = 0; i < numBlocks; ++i) {
        size_t blockSize = (i + 1) * 1024; // Размер блока i * 1024
        blocks[i] = malloc(blockSize);
    }

    // Вывод статистики о выделенной памяти
    malloc_stats();

    // Освобождение памяти
    for (int i = 0; i < numBlocks; ++i) {
        free(blocks[i]);
    }

    return 0;
```

```
}
```
1. Выделение памяти: В первом случае выделяется 1000 блоков фиксированного размера 120 байт. Во втором случае каждый блок имеет размер, равный номеру итерации, умноженному на 1024.
2. Статистика: После выделения памяти вызывается malloc_stats(), чтобы вывести статистику о выделенной и освобожденной памяти.
3. Освобождение памяти: В конце программы выделенная память освобождается с помощью free().

**Лабораторная работа №3. Шаблоны**

***Задача ««Углубление» функции»***

Реализовать с использованием метапрограммирования шаблон функции, который позволил бы применять заданную функцию многократно.

deep<f, 3>(10);  *// То же, что и f(f(f(10)))*

deep<g, 1>(x);  *// g(x)*

**Ответ:**

Чтобы реализовать шаблон функции, который применяет заданную функцию многократно (в соответствии с заданным уровнем "глубины"), можно использовать метапрограммирование в C++. Мы можем использовать рекурсивные шаблоны, чтобы добиться этого.

Пример реализации

```
#include <iostream>

// Пример функции
int f(int x) {
    std::cout << "Applying f: " << x << std::endl;
    return x + 1; // Возвращает x + 1
}

int g(int x) {
    std::cout << "Applying g: " << x << std::endl;
    return x * 2; // Возвращает x * 2
}

// Основной шаблон для глубины
template<typename Func, int depth>
struct deep_impl {
    static auto apply(int x) {
        return deep_impl<Func, depth - 1>::apply(Func::apply(x));
    }
};

// Специализация для depth = 0
template<typename Func>
struct deep_impl<Func, 0> {
    static auto apply(int x) {
        return x; // Просто возвращаем значение
    }
};

// Упрощаем вызов
template<typename Func, int depth>
auto deep(int x) {
    return deep_impl<Func, depth>::apply(x);
}

// Создаем структуру для функций
struct FuncF {
    static int apply(int x) {
        return f(x);
    }
};

struct FuncG {
    static int apply(int x) {
        return g(x);
```

```
    }
};

int main() {
    int x = 10;
    int result1 = deep<FuncF, 3>(x); // Применит f 3 раза
    std::cout << "Result of deep<FuncF, 3>(10): " << result1 << std::endl;

    int result2 = deep<FuncG, 1>(x); // Применит g 1 раз
    std::cout << "Result of deep<FuncG, 1>(10): " << result2 << std::endl;

    return 0;
}
```

1. Определение функций: Функции f и g определены для демонстрации. Функция f увеличивает переданное значение на 1, а функция g удваивает его.
2. Шаблоны deep_impl:
   - Этот шаблон является рекурсивным. Он использует параметр depth для определения, сколько раз функция должна быть применена.
   - Каждый раз, когда apply вызывается на уровне depth, он вызывает apply на следующем уровне (depth - 1), передавая результат текущего применения функции.
3. Специализация для глубины 0:
   - Когда достигается depth = 0, возвращается текущее значение без применения функции.
4. Функции-обертки:
   - FuncF и FuncG являются структурами, каждая из которых содержит статическую функцию apply, вызывающую соответствующую функцию.
5. Упрощение вызова:
   - Функция deep использует deep_impl для применения функции на заданном уровне.

При запуске программы будет выполнен вызов deep<FuncF, 3>(10), что эквивалентно f(f(f(10))), и deep<FuncG, 1>(10), что эквивалентно g(10). Затем программа выводит результаты этих операций.

***Задача «Шаблонная функция sum_all»***

Реализовать функцию sum_all, которая позволяла бы суммировать аргумент практически любого типа, который ей передан.

**Входные данные**

vector<int> v1 = { 1, 2, 3 };
vector<double> v2 = { 1, 2, 3 };
vector<string> v3 = { "a", "bc", "def" };
vector<char> v4 = { 'a', 'b', 'c' };

**Описание задачи**

sum_all будет представлять из шаблон функции с частичной специализацией для vector<T>.

**Выходные данные**

sum_all(5); *// 5*
sum_all(3.0); *// 3.0*
sum_all(v1); *// 6*
sum_all(v2); *// 6.0*
sum_all(v3); *// строка "abcdef"*
sum_all(v4); *// строка "abc"*

**Ответ:**

Для реализации функции sum_all, которая обрабатывает различные типы, в том числе int, double, string, а также векторы этих типов, мы можем использовать шаблоны и частичную специализацию в C++.

Пример реализации

```
#include <iostream>
#include <vector>
#include <string>
#include <numeric> // Для std::accumulate
#include <type_traits>

// Основной шаблон для sum_all
template<typename T>
T sum_all(T value) {
    return value;
}
```

```cpp
// Частичная специализация для std::vector<T>
template<typename T>
T sum_all(const std::vector<T>& vec) {
    // Если тип T является строкой или символом, то склеиваем строки
    if constexpr (std::is_same_v<T, std::string>) {
        return std::accumulate(vec.begin(), vec.end(), std::string{});
    } else if constexpr (std::is_same_v<T, char>) {
        return std::accumulate(vec.begin(), vec.end(), std::string{});
    } else {
        // Для остальных типов, используем стандартную сумму
        return std::accumulate(vec.begin(), vec.end(), T{});
    }
}

int main() {
    std::vector<int> v1 = { 1, 2, 3 };
    std::vector<double> v2 = { 1.0, 2.0, 3.0 };
    std::vector<std::string> v3 = { "a", "bc", "def" };
    std::vector<char> v4 = { 'a', 'b', 'c' };

    std::cout << "sum_all(5): " << sum_all(5) << std::endl;          // 5
    std::cout << "sum_all(3.0): " << sum_all(3.0) << std::endl;      // 3.0
    std::cout << "sum_all(v1): " << sum_all(v1) << std::endl;        // 6
    std::cout << "sum_all(v2): " << sum_all(v2) << std::endl;        // 6.0
    std::cout << "sum_all(v3): " << sum_all(v3) << std::endl;        // "abcdef"
    std::cout << "sum_all(v4): " << sum_all(v4) << std::endl;        // "abc"

    return 0;
}
```

1. Основной шаблон sum_all:
  - Этот шаблон принимает один аргумент и просто возвращает его. Это позволяет обрабатывать аргументы любого типа.
2. Частичная специализация для std::vector<T>:
  - Мы специализируем шаблон для std::vector. В этой специализации используется std::accumulate для суммирования элементов вектора.
  - Используется if constexpr для проверки типа.
    - Если тип T является std::string, мы складываем строки.
    - Если тип T является char, мы также складываем в строки (т.е., преобразуем символы в строку).
    - В остальных случаях (например, для int, double), мы выполняем стандартное суммирование.
3. Функция std::accumulate:
  - Она используется для суммирования элементов вектора. Для векторов целых и дробных чисел она возвращает сумму, для строк — конкатенацию.
4. В main:
  - Проводятся тесты для различных типов вызовов sum_all. Результаты каждой операции выводятся на экран.

При запуске программы будут выведены результаты для различных вызовов sum_all, как указано в задаче. Это позволяет обрабатывать значения различных типов и векторы этих типов.

***Задача «Оптимизированный copy»***

Реализовать собственную версию шаблонной функции copy с ускорением работы для частных случаев.

Мотивация

Шаблонная функция std::copy реализует простой цикл. При копировании, например, массивов байтов (char или unsigned char) процесс будет происходить побайтово. Однако в силу архитектуры памяти и процессора копирование одного байта требует столько же времени, сколько и копирование машинного слова (4 байта для 32-разрядной архитектуры, 8 байт — для 64-разрядной). Функции побайтового копирования из стандартной библиотеки C (memcpy, memmove, strcpy и т.д.) учитывают этот нюанс, и мы могли бы воспользоваться ими.

Описание задачи

Возьмите два частных случая:

1. Итераторы являются указателями.
2. Итераторы относятся к std::vector.

И при условии, что элемент в принципе допускает побайтовое копирование (для проверки можно воспользоваться is_fundamental), реализуйте вызов memmove в соответствующих частных специализациях.

Тестирование

Проверьте, что копирование работает корректно, а также то, что частные специализации достаточно ограничены, чтобы не использовать memmove для более сложных типов данных. Например, для итераторов std::vector<std::string> мы не можем использовать memmove. Здесь должен сработать общий алгоритм копирования.

**Ответ:**

Для реализации собственной версии шаблонной функции copy, которая ускоряет процесс копирования для определенных случаев, мы можем использовать memmove для случаев, когда итераторы являются указателями (например, для массивов) и когда они относятся к std::vector. Мы также будем использовать std::is_fundamental для проверки типов и ограничения применения memmove только на фундаментальные типы.

Пример реализации

```
#include <iostream>
#include <vector>
#include <cstring> // Для memmove
#include <type_traits>

// Общая шаблонная функция copy
template<typename InputIt, typename OutputIt>
OutputIt copy(InputIt first, InputIt last, OutputIt d_first) {
    while (first != last) {
        *d_first++ = *first++;
    }
    return d_first;
}

// Частичная специализация для указателей
template<typename T>
T* copy(T* first, T* last, T* d_first) {
    static_assert(std::is_fundamental<T>::value, "copy for pointers only available
for fundamental types.");
    std::size_t count = last - first; // Количество элементов
    std::memmove(d_first, first, count * sizeof(T)); // Используем memmove
    return d_first + count;
}

// Частичная специализация для std::vector
template<typename T>
typename std::enable_if<std::is_fundamental<T>::value, std::vector<T>&>::type
copy(const std::vector<T>& src, std::vector<T>& dst) {
    if (src.size() > dst.size()) {
        dst.resize(src.size());
    }
    std::memmove(dst.data(), src.data(), src.size() * sizeof(T));
    return dst;
}

int main() {
    // Тестирование для указателей
    int src_arr[] = {1, 2, 3, 4, 5};
    int dst_arr[5];
    copy(src_arr, src_arr + 5, dst_arr);

    std::cout << "Copied array (pointers): ";
    for (const auto& val : dst_arr) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    // Тестирование для std::vector
    std::vector<int> src_vec = {6, 7, 8, 9, 10};
    std::vector<int> dst_vec;
    copy(src_vec, dst_vec);

    std::cout << "Copied vector: ";
    for (const auto& val : dst_vec) {
```

```
        std::cout << val << " ";
    }
    std::cout << std::endl;

    // Тестирование для более сложного типа (не должно использовать memmove)
    std::vector<std::string> src_str_vec = {"Hello", "World"};
    std::vector<std::string> dst_str_vec;

    // Этот вызов будет использовать общий алгоритм копирования
    copy(src_str_vec.begin(), src_str_vec.end(), std::back_inserter(dst_str_vec));

    std::cout << "Copied vector of strings: ";
    for (const auto& val : dst_str_vec) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

1. Общая шаблонная функция copy:
   - Это базовая реализация, которая перебирает элементы между двумя итераторами и копирует их один за другим.
2. Частичная специализация для указателей:
   - Здесь мы используем статическое утверждение (static_assert) с std::is_fundamental для того, чтобы гарантировать, что типы, переданные в функцию, являются фундаментальными.
   - Используется memmove для копирования данных, если итераторы являются указателями.
3. Частичная специализация для std::vector:
   - Эта специализация для std::vector также использует memmove, но перед вызовом выполняет проверку, чтобы изменить размер целевого вектора, если это необходимо.
   - Опять же, мы убеждаемся, что тип T является фундаментальным.
4. Тестирование:
   - Первая часть тестирования показывает, как работает копирование обычных массивов (src_arr) с использованием указателей.
   - Вторая часть тестирует копирование значений из одного вектора (src_vec) в другой, используя memmove.
- Третья часть тестирует копирование строковых классов вектора, показывая, что для сложных типов используется базовая (общая) версия copy.

При запуске этой программы будут скопированы массивы и векторы, и результаты будут выведены на экран, демонстрируя корректную работу обоих подходов к копированию.

**Лабораторная работа 4. Графический интерфейс**

Задание

Создать проект приложения с графическим пользовательским интерфейсом.

**Ответ:**

Приведён проект приложения с графическим пользовательским интерфейсом (GUI) с использованием библиотеки Qt, представляющий собой калькулятор, который позволяет выполнять основные арифметические операции (сложение, вычитание, умножение и деление).

Шаги по созданию проекта «Калькулятор»

1. Установка Qt:
   - Qt скачивается и устанавливается с официального сайта [qt.io](https://www.qt.io/download). Установка Qt Creator должна быть завершена.
2. Создание нового проекта:
   - Запускается Qt Creator и создаётся новый проект. Выбор "Qt Widgets Application". Указывается имя проекта и местоположение для сохранения.
3. Настройка интерфейса:
   - В разделе дизайна (Designer) Qt Creator выполняется перетаскивание элементов интерфейса на форму.

Необходимы:
- QLineEdit для отображения выражения и результата.
- QPushButton для цифр (0-9), операций (+, -, *, /) и кнопки "=" для вычисления результата.
- Кнопка "C" для сброса.

Пример реализации основного окна калькулятора

main.cpp

```
#include "calculator.h"
#include "ui_calculator.h"
#include <QPushButton>
```

```
Calculator::Calculator(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::Calculator),
    result(0),
    pendingOperation("") {
    ui->setupUi(this);

    // Подключение сигналов кнопок к слоту
    for (QPushButton* button : findChildren<QPushButton*>()) {
        connect(button, &QPushButton::clicked, this,
&Calculator::on_buttonClicked);
    }
}

Calculator::~Calculator() {
    delete ui;
}

void Calculator::on_buttonClicked() {
    QPushButton *button = qobject_cast<QPushButton *>(sender());
    QString buttonText = button->text();

    if (buttonText == "C") {
        ui->lineEdit->clear();
        result = 0;
        pendingOperation.clear();
    } else if (buttonText == "=") {
        if (!pendingOperation.isEmpty()) {
            double secondOperand = ui->lineEdit->text().toDouble();
            if (pendingOperation == "+") result += secondOperand;
            else if (pendingOperation == "-") result -= secondOperand;
            else if (pendingOperation == "*") result *= secondOperand;
            else if (pendingOperation == "/") result /= secondOperand;
            ui->lineEdit->setText(QString::number(result));
            pendingOperation.clear();
        }
    } else if (buttonText == "+" || buttonText == "-" || buttonText == "*" ||
buttonText == "/") {
        if (!pendingOperation.isEmpty()) {
            // Вычисление предыдущей операции
            double secondOperand = ui->lineEdit->text().toDouble();
            if (pendingOperation == "+") result += secondOperand;
            else if (pendingOperation == "-") result -= secondOperand;
            else if (pendingOperation == "*") result *= secondOperand;
            else if (pendingOperation == "/") result /= secondOperand;
        } else {
            result = ui->lineEdit->text().toDouble();
        }
        pendingOperation = buttonText;
        ui->lineEdit->clear();
    } else {
        // Добавление цифры
        ui->lineEdit->setText(ui->lineEdit->text() + buttonText);
    }
}
```

calculator.ui

- В этом файле выполняется визуальное проектирование интерфейса с использованием Qt Designer. Добавляется QLineEdit для отображения результата и несколько QPushButton для цифр и операций. Оставляется кнопка "C" для сброса, а также кнопка "=" для вычисления результата.

Описание пользовательского интерфейса

- QLineEdit используется для отображения текущего ввода и результатов.
- Кнопки для цифр от 0 до 9.
- Кнопки для арифметических операций: +, -, *, /.

- Кнопка "C" предназначена для сброса ввода.
- Кнопка "=" используется для вычисления результата.
Компиляция и запуск
- Все файлы сохраняются, проект строится в Qt Creator. После завершения процесса появляется интерфейс калькулятора.

**4.1.2. Компьютерная программа**

Тема 1. Основные концепции C++.
Тема 2. Базовые и продвинутые методы управления ресурсами на C++.
Тема 3. Применение шаблонов C++ для проектирования программ. Разработка и проектирование многопоточных программ.
Тема 4. Разработка и проектирование графического пользовательского интерфейса.

**4.1.2.1. Порядок проведения.**

На основе изученного программного обеспечения построения интеллектуальных информационных систем студентам предлагается написать компьютерную программу по вариантам.

**4.1.2.2 Критерии оценивания**

**17-20 баллов ставится, если обучающийся:**

Код компьютерной программы правильный. Компьютерная программа выполняется без ошибок. Результат компьютерной программы полностью соответствует её целям.

**14-16 баллов ставится, если обучающийся:**

Код компьютерной программы в основном правильный. Компьютерная программ выполняется без ошибок. Результат компьютерной программы в основном соответствует её целям.

**11-15 баллов ставится, если обучающийся:**

При выполнении компьютерной программы возможны ошибки. Код компьютерной программы частично правильный. Результат компьютерной программы частично соответствует её целям.

**0—10 баллов ставится, если обучающийся:**

Компьютерная программа не выполняется. Код компьютерной программы выполнен с ошибками. Результат программы не соответствует её целям.

**4.1.2.3. Содержание оценочного средства**

**1. Разработка программы управления ИТ-проектами: задача нахождения себестоимости ИТ-проекта.**

Разработка программы для управления ИТ-проектами является важным шагом в оптимизации процессов управления расходами и ресурсами. Одной из ключевых задач программы является расчет себестоимости ИТ-проекта, включая затраты на трудозатраты, оборудование, программное обеспечение и другие ресурсы.

Определим шаги для разработки программы

1. Сбор требований. Необходимо определить ключевые параметры, которые необходимо учитывать при вычислении себестоимости проекта. Это могут быть затраты на работу специалистов, покупки оборудования, расходы на лицензии и административные затраты.
2. Определение архитектуры. Программа может быть реализована как консольное приложение с возможностью ввода данных пользователем для расчета себестоимости.
3. Выбор технологий. Мы будем использовать язык C++ для разработки программы и стандартную библиотеку для обработки ввода-вывода.

4. Реализация. На этом этапе происходит написание кода. Пример программы на C++ для расчета себестоимости ИТ-проекта:

```
#include <iostream>
#include <vector>
#include <string>

class CostItem {
public:
    std::string name;
    double cost;

    CostItem(const std::string& name, double cost) : name(name), cost(cost) {}
};

class ProjectCostCalculator {
private:
    std::vector<CostItem> costItems;
```

```
public:
    void addCostItem(const CostItem& item) {
        costItems.push_back(item);
    }

    double calculateTotalCost() {
        double totalCost = 0.0;
        for (const auto& item : costItems) {
            totalCost += item.cost;
        }
        return totalCost;
    }

    void printCostBreakdown() {
        std::cout << "Себестоимость ИТ-проекта:\n";
        for (const auto& item : costItems) {
            std::cout << " - " << item.name << ": " << item.cost << " руб.\n";
        }
        std::cout << "---------------------------------\n";
        std::cout << "Общая себестоимость: " << calculateTotalCost() << " руб.\n";
    }
};

int main() {
    ProjectCostCalculator calculator;

    // Добавление элементов себестоимости
    calculator.addCostItem(CostItem("Трудозатраты (разработчики)", 500000));
    calculator.addCostItem(CostItem("Оборудование", 100000));
    calculator.addCostItem(CostItem("Лицензии на ПО", 30000));
    calculator.addCostItem(CostItem("Административные расходы", 20000));

    // Вывод разбивки себестоимости проекта
    calculator.printCostBreakdown();

    return 0;
}
```

В приведенном примере реализована базовая программа, которая позволяет добавлять элементы себестоимости и вычислять общую стоимость ИТ-проекта. Каждый элемент имеет название и стоимость, и программа выводит подробную разбивку всех затрат.

### 2. Программное обеспечение системы резервирования билетов

Разработка системы резервирования билетов является важным аспектом для любого предприятия, предоставляющего услуги в области транспорта, кино, театра и других событий. Система должна позволять пользователям просматривать доступные билеты, а также осуществлять их бронирование.

Определим основные шаги для разработки системы:
1. Сбор требований. Необходимо определить основные функции системы. Важные функции могут включать просмотр доступных мест, бронирование билетов, отмену бронирования и просмотр истории бронирований.
2. Определение архитектуры. Мы можем реализовать систему как консольное приложение с использованием структур данных для хранения информации о мероприятиях и забронированных билетах.
3. Выбор технологий. Мы будем использовать язык C++ для разработки и стандартную библиотеку для обработки ввода-вывода.
4. Реализация. На этом этапе написан код. Пример программы на C++ для системы резервирования билетов:

```
#include <iostream>
#include <vector>
#include <string>
#include <map>

class Ticket {
public:
```

```
    std::string event;
    int seatNumber;
    bool reserved;

    Ticket(std::string eventName, int seat)
        : event(eventName), seatNumber(seat), reserved(false) {}
};

class TicketBookingSystem {
private:
    std::vector<Ticket> tickets;

public:
    TicketBookingSystem() {
        // Инициализация примера с мероприятиями
        tickets.emplace_back("Концерт", 1);
        tickets.emplace_back("Концерт", 2);
        tickets.emplace_back("Театр", 1);
        tickets.emplace_back("Театр", 2);
    }

    void viewTickets() {
        std::cout << "Доступные билеты:\n";
        for (const auto& ticket : tickets) {
            if (!ticket.reserved) {
                std::cout << "Событие: " << ticket.event << ", Место: " <<
ticket.seatNumber << "\n";
            }
        }
    }

    void reserveTicket(int seatNumber, const std::string& eventName) {
        for (auto& ticket : tickets) {
            if (ticket.seatNumber == seatNumber && ticket.event == eventName &&
!ticket.reserved) {
                ticket.reserved = true;
                std::cout << "Билет на " << eventName << " с местом " <<
seatNumber << " успешно забронирован.\n";
                return;
            }
        }
        std::cout << "Не удалось зарезервировать билет. Возможно, он уже
забронирован или не существует.\n";
    }

    void cancelReservation(int seatNumber, const std::string& eventName) {
        for (auto& ticket : tickets) {
            if (ticket.seatNumber == seatNumber && ticket.event == eventName &&
ticket.reserved) {
                ticket.reserved = false;
                std::cout << "Бронирование на " << eventName << " с местом " <<
seatNumber << " отменено.\n";
                return;
            }
        }
        std::cout << "Не удалось отменить бронирование. Возможно, оно не
существует.\n";
    }
};

int main() {
    TicketBookingSystem system;
    int choice;
```

```
    while (true) {
        std::cout << "\n1. Просмотреть доступные билеты\n";
        std::cout << "2. Забронировать билет\n";
        std::cout << "3. Отменить бронь\n";
        std::cout << "4. Выход\n";
        std::cout << "Выберите действие: ";
        std::cin >> choice;

        if (choice == 1) {
            system.viewTickets();
        } else if (choice == 2) {
            std::string eventName;
            int seatNumber;
            std::cout << "Введите название события: ";
            std::cin >> eventName;
            std::cout << "Введите номер места: ";
            std::cin >> seatNumber;
            system.reserveTicket(seatNumber, eventName);
        } else if (choice == 3) {
            std::string eventName;
            int seatNumber;
            std::cout << "Введите название события: ";
            std::cin >> eventName;
            std::cout << "Введите номер места: ";
            std::cin >> seatNumber;
            system.cancelReservation(seatNumber, eventName);
        } else if (choice == 4) {
            break;
        } else {
            std::cout << "Неверный выбор, попробуйте еще раз.\n";
        }
    }

    return 0;
}
```

В приведенном примере реализована простая система резервирования билетов, которая позволяет пользователю:
- Просматривать доступные билеты.
- Бронировать билеты по номеру места и названию события.
- Отменять бронирование.

Система использует структуру данных для хранения информации о билетах и их статусе (забронирован или свободен).

### 3. Разработка программного модуля «Складской комплекс»

Разработка модуля для управления складом является важным аспектом для оптимизации процессов учета и контроля товарных запасов. Модуль должен позволять пользователям добавлять, удалять и отслеживать товары на складе.

Определим основные шаги для разработки модуля:
1. Сбор требований. Необходимо определить функции, которые должен выполнять модуль. Основные функции могут включать добавление товаров, удаление товаров, отображение текущего состояния склада и поиск по товарам.
2. Определение архитектуры. Мы можем реализовать модуль как консольное приложение, которое будет взаимодействовать с пользователем через командный интерфейс.
3. Выбор технологий. Мы будем использовать язык C++ и стандартные библиотеки для обработки ввода-вывода.
4. Реализация. Пример кода на C++ для модуля «Складской комплекс»:

```
#include <iostream>
#include <vector>
#include <string>

class Product {
```

```
public:
    std::string name;
    int quantity;

    Product(const std::string& name, int qty) : name(name), quantity(qty) {}
};

class Warehouse {
private:
    std::vector<Product> products;

public:
    void addProduct(const std::string& name, int qty) {
        for (auto& product : products) {
            if (product.name == name) {
                product.quantity += qty; // Увеличиваем количество, если товар уже
существует
                std::cout << "Товар \"" << name << "\" добавлен. Текущая запасы: "
<< product.quantity << std::endl;
                return;
            }
        }
        products.emplace_back(name, qty);
        std::cout << "Товар \"" << name << "\" добавлен с количеством: " << qty <<
std::endl;
    }

    void removeProduct(const std::string& name, int qty) {
        for (auto& product : products) {
            if (product.name == name) {
                if (product.quantity >= qty) {
                    product.quantity -= qty;
                    std::cout << "Товар \"" << name << "\" удалён. Оставшееся
количество: " << product.quantity << std::endl;
                    if (product.quantity == 0) {
                        // Если количество товара стало 0, удаляем его из склада
                        products.erase(std::remove(products.begin(),
products.end(), product), products.end());
                        std::cout << "Товар \"" << name << "\" полностью уничтожен
из склада." << std::endl;
                    }
                } else {
                    std::cout << "Недостаточно товара \"" << name << "\" для
удаления." << std::endl;
                }
                return;
            }
        }
        std::cout << "Товар \"" << name << "\" не найден на складе." << std::endl;
    }

    void viewProducts() const {
        std::cout << "\nТекущие товары на складе:\n";
        for (const auto& product : products) {
            std::cout << "Товар: " << product.name << ", Количество: " <<
product.quantity << std::endl;
        }
        if (products.empty()) {
            std::cout << "Склад пуст." << std::endl;
        }
    }
};

int main() {
```

```
    Warehouse warehouse;
    int choice;

    while (true) {
        std::cout << "\n1. Добавить товар\n";
        std::cout << "2. Удалить товар\n";
        std::cout << "3. Просмотреть товары\n";
        std::cout << "4. Выход\n";
        std::cout << "Выберите действие: ";
        std::cin >> choice;

        if (choice == 1) {
            std::string name;
            int qty;
            std::cout << "Введите название товара: ";
            std::cin >> name;
            std::cout << "Введите количество: ";
            std::cin >> qty;
            warehouse.addProduct(name, qty);
        } else if (choice == 2) {
            std::string name;
            int qty;
            std::cout << "Введите название товара: ";
            std::cin >> name;
            std::cout << "Введите количество для удаления: ";
            std::cin >> qty;
            warehouse.removeProduct(name, qty);
        } else if (choice == 3) {
            warehouse.viewProducts();
        } else if (choice == 4) {
            break;
        } else {
            std::cout << "Неверный выбор, попробуйте еще раз.\n";
        }
    }

    return 0;
}
```

В приведенном примере реализован простой модуль для управления складом, который позволяет пользователю:
- Добавлять товары на склад.
- Удалять товары со склада.
- Просматривать текущие товары и их количество.
Модуль использует вектор для хранения информации о товарах.

### 4. Программирование оценки кредитоспособности физических лиц

Оценка кредитоспособности физических лиц является важным инструментом для банков и финансовых учреждений, позволяющим принимать решения о выдаче кредита. Программа должна учитывать различные параметры для вычисления оценки.

Определим основные шаги для разработки программы:
1. Сбор требований. Необходимо определить параметры, которые нужно учитывать при оценке кредитоспособности. Основными параметрами могут быть: ежемесячный доход, сумма задолженности, уровень накоплений и возраст.
2. Определение формул для оценки. Мы можем использовать простую формулу для расчета кредитоспособности, учитывающую соотношение дохода к задолженности (например, отношение дохода к долга).
3. Выбор технологий. Мы будем использовать язык C++ для создания консольного приложения, которое будет взаимодействовать с пользователем через командный интерфейс.
4. Реализация. Пример кода на C++ для оценки кредитоспособности физических лиц:

```
#include <iostream>
#include <string>
```

```
class CreditScoreCalculator {
private:
    double monthlyIncome; // Ежемесячный доход
    double totalDebt; // Общая задолженность
    double savings; // Накопления
    int age; // Возраст

public:
    CreditScoreCalculator(double income, double debt, double savings, int age)
        : monthlyIncome(income), totalDebt(debt), savings(savings), age(age) {}

    double calculateDebtToIncomeRatio() {
        if (monthlyIncome == 0) {
            return 0; // Избегаем деления на ноль
        }
        return totalDebt / monthlyIncome;
    }

    double evaluateCreditworthiness() {
        double score = 100.0; // Максимальный балл
        double debtToIncomeRatio = calculateDebtToIncomeRatio();

        if (debtToIncomeRatio > 0.4) {
            score -= (debtToIncomeRatio - 0.4) * 100; // Штраф за высокую
задолженность
        }

        if (savings < 10000) {
            score -= (10000 - savings) / 100; // Штраф за низкие накопления
        }

        if (age < 18) {
            return 0; // Неприемлемый возраст
        }

        if (age < 25) {
            score -= 10; // Штраф за низкий возраст
        }

        return (score < 0) ? 0 : score; // Убедимся, что балл не отрицательный
    }

    void printCreditworthiness() {
        double creditScore = evaluateCreditworthiness();
        std::cout << "Ваш оценочный балл кредитоспособности: " << creditScore <<
std::endl;
        if (creditScore >= 70) {
            std::cout << "Вы имеете высокий уровень кредитоспособности." <<
std::endl;
        } else if (creditScore >= 40) {
            std::cout << "Вы имеете средний уровень кредитоспособности." <<
std::endl;
        } else {
            std::cout << "Вы имеете низкий уровень кредитоспособности." <<
std::endl;
        }
    }
};

int main() {
    double income;
    double debt;
    double savings;
```

```
    int age;

    std::cout << "Введите ваш ежемесячный доход: ";
    std::cin >> income;

    std::cout << "Введите общую сумму задолженности: ";
    std::cin >> debt;

    std::cout << "Введите уровень накоплений: ";
    std::cin >> savings;

    std::cout << "Введите ваш возраст: ";
    std::cin >> age;

    CreditScoreCalculator calculator(income, debt, savings, age);
    calculator.printCreditworthiness();

    return 0;
}
```

В приведенном примере реализована простая программа для оценки кредитоспособности, которая рассчитывает «оценочный балл» на основе входных данных. Основные параметры включают ежемесячный доход, общую задолженность, уровень накоплений и возраст, которые влияют на оценку кредитоспособности.

### 5. Разработка программного обеспечения информационного киоска торговой компании

Разработка программного обеспечения для информационного киоска торговой компании является важным шагом в повышении уровня обслуживания клиентов и улучшении взаимодействия с ними. Киоск должен предоставлять информацию о товарах, акциях, а также обеспечивать возможность навигации по магазину.

Определим шаги для создания программного обеспечения
1. Сбор требований. Необходимо определить ключевые функции, которые должен выполнять киоск. Важные функции могут включать отображение каталога товаров, информацию о наличии, актуальные акции, карты магазина и возможность поиска.
2. Определение архитектуры. Определим архитектуру приложения. Мы можем использовать архитектуру клиент-сервер, где сервер предоставляет данные, а клиент (киоск) отображает их пользователю. Для простоты мы сосредоточимся на локальном приложении без сетевого взаимодействия.
3. Выбор технологий. Мы будем использовать язык C++ для разработки. Для создания простого графического интерфейса мы можем использовать библиотеку Qt, что облегчит процесс работы с пользовательским интерфейсом.
4. Реализация. На этом этапе происходит написание кода. Пример кода для простого информационного киоска на C++ с использованием библиотеки Qt:
Код на C++ с использованием библиотеки Qt:

```
#include <QApplication>
#include <QWidget>
#include <QListView>
#include <QVBoxLayout>
#include <QStringListModel>
#include <QPushButton>
#include <QMessageBox>

class InfoKiosk : public QWidget {
public:
    InfoKiosk(QWidget *parent = nullptr) : QWidget(parent) {
        auto *layout = new QVBoxLayout(this);

        model = new QStringListModel(this);
        QStringList productList;
        productList << "Товар 1 - 100 руб. (В наличии)"
                    << "Товар 2 - 200 руб. (Нет в наличии)"
                    << "Товар 3 - 150 руб. (В наличии)";
```

```
        model->setStringList(productList);

        listView = new QListView(this);
        listView->setModel(model);

        QPushButton *showDetailButton = new QPushButton("Показать детали", this);
        connect(showDetailButton, &QPushButton::clicked, this,
&InfoKiosk::showDetails);

        layout->addWidget(listView);
        layout->addWidget(showDetailButton);
        setLayout(layout);
    }

private:
    QStringListModel *model;
    QListView *listView;

    void showDetails() {
        QModelIndex index = listView->currentIndex();
        if (index.isValid()) {
            QString product = model->data(index, Qt::DisplayRole).toString();
            QMessageBox::information(this, "Детали товара", product);
        } else {
            QMessageBox::warning(this, "Предупреждение", "Выберите товар из
списка");
        }
    }
};

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    InfoKiosk kiosk;
    kiosk.setWindowTitle("Информационный киоск");
    kiosk.resize(400, 300);
    kiosk.show();

    return app.exec();
}
```

В приведенном примере реализована базовая функциональность, которая позволяет отображать список товаров и показывать детали выбранного товара через диалоговое окно. Для полноценной работы киоска целесообразно добавить функционал поиска и фильтрации товаров, а также возможность отображения акций и специального контента.

Примеры возможных вариантов заданий:

1. Разработка программы автоматизации работы деканата вуза
2. Разработка программы автоматизации учета изделий на предприятии
3. Разработка программы автоматизированного рабочего места операциониста библиотеки
4. Разработка программы автоматизации предприятий автосервиса
5. Разработка программы автоматизации учета кадров на предприятии
6. Разработка программы автоматизации строительной организации
7. Разработка программы автоматизации гостиничного комплекса
8. Разработка программы автоматизации аптеки
9. Разработка программы автоматизации туристической фирмы
10. Программные комплексы поддержки принятия управленческих решений (разработка системы учета договоров и расчетов с субподрядчиками)
11. Разработка программного обеспечения, решающего задачу распределения готовой продукции между складами
12. Программный учёт материально-технических средств на примере предприятия
13. Программирование выдачи справок клиентам строительной компании

**4.2. Оценочные средства промежуточной аттестации**

По дисциплине предусмотрен экзамен. Он проходит по билетам. В каждом билете содержится один теоретический вопрос и одна задача. Экзамен проводится в устной, письменной или компьютерной форме. Оценивается владение материалом, его системное освоение, способность применять нужные знания, навыки и умения при анализе проблемных ситуаций и решении практических заданий.

**4.2.1. Устный ответ**

**4.2.1.1. Порядок проведения.**

Промежуточная аттестация нацелена на комплексную проверку освоения дисциплины. Обучающийся получает вопрос(ы)/задание(я) и время на подготовку. Промежуточная аттестация проводится в устной, письменной или компьютерной форме. Оценивается владение материалом, его системное освоение, способность применять нужные знания, навыки и умения при анализе проблемных ситуаций и решении практических заданий.

**4.2.1.2. Критерии оценивания.**

**18-20 баллов ставится, если обучающийся:**

В ответе качественно раскрыл содержание темы. Ответ хорошо структурирован. Прекрасно освоен понятийный аппарат. Продемонстрирован высокий уровень понимания материала. Превосходное умение формулировать свои мысли, обсуждать дискуссионные положения.

**14-17 баллов ставится, если обучающийся:**

Основные вопросы темы раскрыл. Структура ответа в целом адекватна теме. Хорошо освоен понятийный аппарат. Продемонстрирован хороший уровень понимания материала. Хорошее умение формулировать свои мысли, обсуждать дискуссионные положения.

**11-13 баллов ставится, если обучающийся:**

Тему частично раскрыл. Ответ слабо структурирован. Понятийный аппарат освоен частично. Понимание отдельных положений из материала по теме. Удовлетворительное умение формулировать свои мысли, обсуждать дискуссионные положения.

**0--10 баллов ставится, если обучающийся:**

Тему не раскрыл. Понятийный аппарат освоен неудовлетворительно. Понимание материала фрагментарное или отсутствует. Неумение формулировать свои мысли, обсуждать дискуссионные положения.

**4.2.1.3. Оценочные средства.**

**Вопросы для устного или письменного ответа**

1. Развитие и стандартизация языка C++.
2. Типы данных.
3. Наследование.
4. Полиморфизм.
5. Инкапсуляция.
6. Специальные функции.
7. Шаблоны.
8. Стандартная библиотека.
9. Концепции C++.
10. Основные стратегии копирования-владения.
11. Стратегия запрета копирования.
12. Стратегия исключительного владения.
13. Стратегия глубокого копирования.
14. Стратегия совместного владения.
15. Определение функции обмена состояниями для класса.
16. Удаление промежуточных копий компилятором.
17. Реализация семантики перемещения.
18. Возможные варианты реализации стратегии совместного владения.
19. Стратегия исключительного владения и семантика перемещения.
20. Жизненный цикл ресурса и объекта-владельца ресурса
21. Шаблон C++.
22. Обзор промышленных программных библиотек.
23. Многопоточное программирование.
24. Управление потоками.
25. Разделение данных между потоками.
26. Синхронизация параллельных операций.
27. Проектирование параллельных программ.
28. Продвинутое управление потоками.
29. Тестирование и отладка многопоточных приложений.
30. Графический пользовательский интерфейс.
31. Проектирование графического пользовательского интерфейса на C++.
32. Инструментарий разработчика.
33. Разбор существующего проекта с открытым исходным кодом.

**4.2.1.4. Ключи к вопросам для устного или письменного ответа**

1. Развитие и стандартизация языка C++

C++ был разработан Бьёрн Страуструпом в начале 1980-х годов как расширение языка C для поддержки объектно-ориентированного программирования. Первая версия, C++, появилась в 1985 году. С тех пор язык значительно эволюционировал, и его функциональные возможности расширялись через ряд стандартов. В 1998 году был принят первый стандарт ISO C++ (C++98), который включал основные принципы объектно-ориентированного программирования, такие как классы и наследование. В 2003 году вышла небольшая редакция, C++03, которая устраняла некоторые ошибки и недостатки. Основные изменения были внедрены в стандарте C++11, который добавил такие концепции, как автоматическое выведение типов, лямбда-выражения и умные указатели. Последующие стандарты, такие как C++14, C++17 и C++20, продолжают улучшать язык, добавляя новые функции и повышая производительность. Стандартизация C++ осуществляется комитетом ISO/IEC JTC1/SC22/WG21, который регулярно собирается для обсуждения улучшений и обновлений языка.

2. Типы данных

В языке C++ существует несколько основных типов данных, которые можно разделить на примитивные и составные. Примитивные типы включают: int (целые числа), float (числа с плавающей точкой), double (двойные числа с плавающей точкой), char (символы) и bool (логические значения). Эти типы позволяют представлять различные виды данных. Составные типы данных включают массивы, структуры и классы. Массивы представляют собой наборы элементов одного типа, структуры позволяют объединять разные типы данных, а классы служат основой объектно-ориентированного программирования, предоставляя возможность создавать объекты с собственными атрибутами и методами. Также в C++ существует возможность создавать указатели, которые хранят адреса других переменных, а также ссылки, которые представляют собой альтернативные имена для существующих переменных. Правильный выбор типов данных является ключевым для эффективного использования памяти и производительности программы.

3. Наследование

Наследование — это один из основных принципов объектно-ориентированного программирования, позволяющий создавать новые классы на основе существующих. В C++ наследование реализуется путем создания производного класса, который унаследует характеристики базового класса. Существует несколько типов наследования: публичное, защищенное и частное, определяющие уровень доступа к членам базового класса. Публичное наследование позволяет пользователю производного класса получить доступ к публичным и защищенным членам базового класса, тогда как защищенное и частное наследование ограничивает этот доступ. Наследование способствует повторному использованию кода, упрощает его поддержку и логически структурирует иерархию классов. Кроме того, C++ поддерживает множественное наследование, позволяющее производным классам унаследовать свойства сразу от нескольких базовых классов. Однако множественное наследование также может привести к проблемам, таким как "алмазная проблема", требующей специального разрешения.

4. Полиморфизм

Полиморфизм — это концепция объектно-ориентированного программирования, позволяющая объектам разных классов обрабатывать один и тот же интерфейс по-разному. В C++ полиморфизм достигается через механизмы виртуальных функций и переопределения методов. С помощью виртуальных функций можно определить методы в базовом классе, которые затем можно переопределить в производных классах, предоставляя каждому из них свою реализацию. Такой подход позволяет использовать базовый класс как интерфейс для работы с объектами разных производных классов. Полиморфизм способствует гибкости и расширяемости кода, так как программы могут работать с объектами различных классов, не зная их точного типа. Существует два вида полиморфизма: компиляционный (включая перегрузку функций и операторов) и рантаймовый (через виртуальные функции). Это делает C++ мощным инструментом для разработки сложных программных систем.

5. Инкапсуляция

Инкапсуляция — это один из ключевых принципов объектно-ориентированного программирования, который заключается в объединении данных и методов, работающих с этими данными, в одной сущности, называемой классом. Этот принцип позволяет скрывать внутренние детали реализации объекта и защищать его состояние от неконтролируемого доступа и изменений. В C++ инкапсуляция достигается с помощью спецификаторов доступа: public, protected и private. Публичные члены класса доступны извне, защищенные — только для производных классов и самого класса, а приватные — доступны только внутри класса. Таким образом, инкапсуляция снижает связанность кода, улучшает управление изменениями в нем и способствует созданию устойчивых и надежных программных систем, упрощая взаимодействие между объектами и предотвращая случайные ошибки.

6. Специальные функции

Специальные функции в C++ — это функции, которые имеют особое значение и предназначение для классов. К ним относятся конструкторы, деструкторы, копирующие и перемещающие операции. Конструктор — это специальный метод, который вызывается при создании объекта класса, инициализируя его данные. Деструктор — это метод, который вызывается при уничтожении объекта и используется для освобождения ресурсов. Копирующие конструкторы позволяют создавать новый объект как копию существующего, обеспечивая

правильное копирование ресурсов, особенно при работе с динамической памятью. Перемещающие операции (конструктор перемещения и оператор перемещения) предназначены для оптимизации работы с временными объектами и ресурсами, позволяя переносить данные из одного объекта в другой, вместо их копирования. Эти специальные функции являются важной частью управления ресурсами и обеспечивают правильное поведение объектов в C++.

7. Шаблоны

Шаблоны в C++ — это мощный механизм, позволяющий создавать обобщенные функции и классы, которые могут работать с различными типами данных. Они позволяют писать код, который автоматически адаптируется к различным типам, не дублируя его для каждого из них. Есть два основных вида шаблонов: шаблоны функций и шаблоны классов. Шаблоны функций позволяют создавать функции, которые принимают параметры любого типа, а шаблоны классов создают классы, работающие с обобщенными типами. При использовании шаблонов компилятор генерирует код для конкретных типов на этапе компиляции, что обеспечивает высокую производительность. Также шаблоны играют важную роль в стандартной библиотеке STL, которая включает в себя контейнеры (например, вектор, список) и алгоритмы, работающие на принципах обобщенного программирования. Шаблоны способствуют сокращению объема кода, улучшению читаемости и повторному использованию.

5. Инкапсуляция

Инкапсуляция — это один из ключевых принципов объектно-ориентированного программирования, который заключается в объединении данных и методов, работающих с этими данными, в одной сущности, называемой классом. Этот принцип позволяет скрывать внутренние детали реализации объекта и защищать его состояние от неконтролируемого доступа и изменений. В C++ инкапсуляция достигается с помощью спецификаторов доступа: public, protected и private. Публичные члены класса доступны извне, защищенные — только для производных классов и самого класса, а приватные — доступны только внутри класса. Таким образом, инкапсуляция снижает связанность кода, улучшает управление изменениями в нем и способствует созданию устойчивых и надежных программных систем, упрощая взаимодействие между объектами и предотвращая случайные ошибки.

6. Специальные функции

Специальные функции в C++ — это функции, которые имеют особое значение и предназначение для классов. К ним относятся конструкторы, деструкторы, копирующие и перемещающие операции. Конструктор — это специальный метод, который вызывается при создании объекта класса, инициализируя его данные. Деструктор — это метод, который вызывается при уничтожении объекта и используется для освобождения ресурсов. Копирующие конструкторы позволяют создавать новый объект как копию существующего, обеспечивая правильное копирование ресурсов, особенно при работе с динамической памятью. Перемещающие операции (конструктор перемещения и оператор перемещения) предназначены для оптимизации работы с временными объектами и ресурсами, позволяя переносить данные из одного объекта в другой, вместо их копирования. Эти специальные функции являются важной частью управления ресурсами и обеспечивают правильное поведение объектов в C++.

7. Шаблоны

Шаблоны в C++ — это мощный механизм, позволяющий создавать обобщенные функции и классы, которые могут работать с различными типами данных. Они позволяют писать код, который автоматически адаптируется к различным типам, не дублируя его для каждого из них. Есть два основных вида шаблонов: шаблоны функций и шаблоны классов. Шаблоны функций позволяют создавать функции, которые принимают параметры любого типа, а шаблоны классов создают классы, работающие с обобщенными типами. При использовании шаблонов компилятор генерирует код для конкретных типов на этапе компиляции, что обеспечивает высокую производительность. Также шаблоны играют важную роль в стандартной библиотеке STL, которая включает в себя контейнеры (например, вектор, список) и алгоритмы, работающие на принципах обобщенного программирования. Шаблоны способствуют сокращению объема кода, улучшению читаемости и повторному использованию, что делает их важным инструментом в C++.

8. Стандартная библиотека

Стандартная библиотека C++ (STL) представляет собой набор готовых классов и функций, предоставляющих мощные инструменты для работы с данными, алгоритмами и контейнерами. Она включает контейнеры, такие как массивы (массивы), векторы, списки, множества, карты и очереди, которые позволяют удобно хранить и обрабатывать данные. Алгоритмы STL обеспечивают широкий набор функциональных возможностей, включая сортировку, поиск, манипуляции с контейнерами и другие операции, которые можно применять к данным, хранящимся в контейнерах. STL также предоставляет инструменты для итерации по элементам контейнеров, что значительно упрощает их использование. Наиболее важные компоненты STL включают итераторы, которые служат абстракцией для доступа к элементам контейнеров. Использование

стандартной библиотеки C++ облегчает разработку программ, повышает производительность и улучшает читаемость кода, позволяя программистам сосредоточиться на решении задач, а не на реализации основных структур данных и алгоритмов.

9. Концепции C++

Концепции C++ представляют собой способ определения требований к шаблонам, позволяющий задавать ограничения для типов, используемых в шаблонах, и тем самым повышать безопасность и читаемость кода. Они были введены в стандарт C++20 и обеспечивают более строгую типизацию, чем традиционные инструменты, такие как static_assert и SFINAE (Substitution Failure Is Not An Error). Концепции позволяют задавать предикаты, которые описывают допустимые свойства типов (например, наличие определенных методов или операций). Это позволяет разработчику определять шаблоны, принимающие только те типы, которые соответствуют заданным условиям. Концепции повышают выразительность кода, делая его более понятным и упрощая отладку, так как ошибки типов могут быть выявлены на этапе компиляции. С помощью концепций можно успешно решать задачи, связанные с обобщенным программированием, улучшая интерфейсы функций и классов.

10. Основные стратегии копирования-владения

Стратегии копирования-владения в C++ определяют, как объекты управляют ресурсами, которыми они владеют, в ходе копирования и перемещения. Основные стратегии включают «копирование значения», «передача владения» и «перемещение». При копировании значения создается полная копия объекта, что может быть затратным в отношении производительности, особенно для объектов с динамическим выделением памяти. В «передаче владения» ресурсы передаются от одного объекта к другому, что подразумевает освобождение ресурсов в исходном объекте, а значит обеспечивает эффективное управление памятью. Перемещение, введенное в C++11, позволяет передавать владение ресурсами от временных объектов без их копирования, используя семантику перемещения (перемещающие конструкторы и операторы). Каждая из этих стратегий подходит для разных сценариев использования и зависит от требований к производительности и безопасности управления памятью в приложении.

11. Стратегия запрета копирования

Стратегия запрета копирования в C++ используется для предотвращения неявного или явного копирования объектов, что особенно важно, когда класс управляет уникальным ресурсом, который не может быть скопирован безопасно, например, в случае использования низкоуровневых системных ресурсов (файловые дескрипторы, указатели на выделенную память и т. д.). Чтобы реализовать это, можно объявить недоступными копирующий конструктор и оператор присваивания, сделав их приватными или удаленными, начиная с C++11. Это позволяет избежать ситуаций, когда копия объекта может привести к двойному освобождению ресурса или некорректной работе программы. Классы, использующие стратегию запрета копирования, обычно предоставляют возможность перемещения, что позволяет эффективно передавать владение ресурсами между объектами без копирования, сохраняя при этом контроль за управлением памятью. Эта стратегия часто применяется в современных библиотеках и фреймворках для обеспечения безопасного и эффективного управления ресурсами.

12. Стратегия исключительного владения

Стратегия исключительного владения (exclusive ownership) в C++ подразумевает, что каждый объект несет полную ответственность за управляемый ресурс, и только один объект может владеть этим ресурсом в любой момент времени. Это важный принцип в контексте работы с динамически выделенной памятью или другими ресурсами, такими как файлы и сетевые подключения. В реализации исключительного владения, когда объект уничтожается или выходит из области видимости, соответствующие ресурсы освобождаются, что предотвращает утечки памяти. C++11 ввел умные указатели, такие как std::unique_ptr, которые реализуют эту стратегию, позволяя автоматически управлять временем жизни объектов и избегать ошибок, связанных с ручным управлением памятью. Исключительное владение требует реализации перемещающих конструкторов и операторов, чтобы гарантировать, что передача владения происходит безопасно и эффективно, не допуская несанкционированного копирования объектов.

13. Стратегия глубокого копирования

Стратегия глубокого копирования (deep copy) заключается в создании полной независимой копии объекта, включая все ресурсы, на которые он ссылается. Это особенно актуально для объектов, которые содержат указатели на динамически выделенную память или другие ресурсы, чтобы избежать ситуации, когда несколько объектов ссылаются на один и тот же ресурс, что может привести к проблемам, таким как двойное освобождение памяти или изменение состояния одного объекта, влияющее на другие. В C++ глубокое копирование обычно реализуется в пользовательских копирующих конструкторах и операторах присваивания, которые выделяют новую память для каждого указателя и копируют данные из оригинала. Таким образом, каждый объект будет иметь свои отдельные копии ресурсов, обеспечивая по-прежнему корректное и независимое поведение. Глубокое копирование требует больше времени и ресурсов, чем поверхностное копирование, поэтому при проектировании классов важно понимать, когда именно следует применять эту стратегию.

14. Стратегия совместного владения

Стратегия совместного владения (shared ownership) позволяет нескольким объектам разделять доступ к одному и тому же ресурсу. Это подразумевает, что несколько объектов могут владеть одним ресурсом, и ресурс будет освобожден только тогда, когда все владельцы перестанут ссылаться на него. В C++ эта стратегия реализуется с помощью std::shared_ptr, который ведет учет количества ссылок на ресурс и автоматически освобождает его, когда ссылка достигает нуля. Это позволяет избежать проблем, связанных с двойным освобождением памяти, и упрощает управление жизненным циклом объектов, особенно в сложных иерархиях и при использовании многопоточности. Однако важно также учитывать, что частое использование совместного владения может привести к неизбежной фрагментации памяти и повышенной вероятности циклических ссылок, когда два или более объекта ссылаются друг на друга, что может привести к утечкам памяти. Поэтому при использовании этой стратегии необходимо следить за дизайном и структурой кода, применяя, при необходимости, механизмы, которые смогут предотвратить проблемы с управлением памятью.

15. Определение функции обмена состояниями для класса

Функция обмена состояниями (swap function) в C++ предназначена для обмена значениями между двумя объектами одного и того же класса. Эта функция позволяет эффективно обменивать состояния объектов, избегая необходимости создания временных объектов или копирования данных, что особенно полезно при использовании семантики перемещения. Определение функции swap обычно включает в себя использование стандартного механизма обмена, такого как std::swap, и улучшает производительность операций с классами в условии, что данная функция имеет низкую стоимость выполнения. Пример реализации функции обмена для класса может выглядеть следующим образом:

```
class MyClass {
public:
    int* data;
    size_t size;

    MyClass(size_t s) : size(s), data(new int[s]) {}

    // Деструктор
    ~MyClass() { delete[] data; }

    // Определение функции swap
    void swap(MyClass& other) noexcept {
        std::swap(data, other.data);
        std::swap(size, other.size);
    }

    // Можно определить free swap для удобства
    friend void swap(MyClass& a, MyClass& b) noexcept {
        a.swap(b);
    }
};
```

Определение swap позволяет повысить эффективность операций, таких как перемещение и присваивание, особенно когда объекты управляют динамически выделенной памятью.

16. Удаление промежуточных копий компилятором

Удаление промежуточных копий (copy elision) в C++ — это оптимизация, которая позволяет компилятору избегать создания временных объектов в процессе выполнения программы. Это может произойти, например, при возврате объекта из функции или при передаче объекта по значению. Компилятор может "элиминировать" создание временного объекта, возвращая экземпляр напрямую. Эта оптимизация стала частью стандарта C++ с C++11 и является обязательной для компиляторов при реализации возвращаемого значения для функции, использующей return value optimization (RVO) или named return value optimization (NRVO). Удаление промежуточных копий не только уменьшает потребление памяти, но и повышает скорость выполнения программы, позволяя избежать не только дополнительных конструкций объектов, но и расходов на их копирование и разрушение. Тем не менее, в некоторых случаях, когда копирование объектов может быть необходимым, поведение программы будет изменяться в зависимости от того, зависит ли функция от таких семантик.

17. Реализация семантики перемещения

Семантика перемещения в C++ была введена в стандарте C++11 и предназначена для эффективного управления ресурсами, позволяя передавать владение ресурсами от временных объектов без необходимости их копирования. Реализация семантики перемещения требует создания перемещающего конструктора и перемещающего оператора присваивания. Перемещающий конструктор принимает "правый" объект,

перенастраивая указатели ресурсов и освобождая старые ресурсы. При реализации следует учитывать, чтобы после переноса объекта его состояние оставалось валидным (но не обязательно полным). Пример может выглядеть следующим образом:

```
class MyClass {
public:
    int* data;
    size_t size;

    MyClass(size_t s) : size(s), data(new int[s]) {}

    // Деструктор
    ~MyClass() { delete[] data; }

    // Перемещающий конструктор
    MyClass(MyClass&& other) noexcept
        : data(other.data), size(other.size) {
        other.data = nullptr; // Устанавливаем указатель "other" в nullptr
        other.size = 0;
    }

    // Перемещающий оператор присваивания
    MyClass& operator=(MyClass&& other) noexcept {
        if (this != &other) {
            delete[] data; // Освобождаем текущие ресурсы
            data = other.data;
            size = other.size;
                other.data = nullptr; // Устанавливаем указатель "other" в nullptr
            other.size = 0;
        }
        return *this;
    }
};
```

Эта реализация позволяет благодаря перемещениям значительно повысить производительность, особенно при работе с временными объектами или большими объемами данных, благодаря уменьшению количества операций копирования.

18. Возможные варианты реализации стратегии совместного владения
Стратегия совместного владения (shared ownership) позволяет нескольким объектам совместно владеть ресурсами, при этом обеспечивая корректное управление временем жизни ресурсов. В C++ существует несколько вариантов реализации этой стратегии:
1. Умные указатели:
- std::shared_ptr: стандартный класс, который управляет временем жизни объекта с помощью подсчета ссылок. Когда объект создается через std::shared_ptr, он инкрементирует счетчик ссылок. Когда std::shared_ptr больше не используется (разрушается или переназначается), счетчик уменьшается. Когда счетчик достигает нуля, объект освобождается.
2. Собственные реализации подсчета ссылок:
- Разработка собственного класса, аналогичного std::shared_ptr, где вы реализуете подсчет ссылок и управление памятью. Это требует реализации конструктора копирования, деструктора и корректного управления памятью при создании и уничтожении объектов.
3. Weak Pointer (слабые указатели):
- Использование std::weak_ptr в сочетании с std::shared_ptr для предотвращения циклических ссылок. std::weak_ptr позволяет создать ссылку на объект, но не увеличивает счетчик ссылок. Это может быть полезно, когда вы хотите иметь доступ к объекту, но заботиться о его уничтожении, если он не используется другими std::shared_ptr.
4. Статическая или глобальная память:
- В каких-то случаях можно реализовать совместное владение ресурсами через статические или глобальные переменные. Это позволяет различным объектам ссылаться на одни и те же ресурсы без необходимости управления динамической памятью, хотя такой подход может привести к проблемам с потокобезопасностью и управлением временем жизни ресурсов.
5. Системы сигналов и слотов:

- Использование паттерна проектирования "Сигналы и Слоты", который позволяет нескольким объектам подписываться на события, обеспечивая совместное владение ресурсами, связанными с этими событиями.

Другие подходы могут включать использование различных структур данных или встроенных механизмов для управления временем жизни ресурсов, основываясь на потребностях конкретного проекта.

19. Стратегия исключительного владения и семантика перемещения

Стратегия исключительного владения и семантика перемещения в C++ тесно связаны между собой. Исключительное владение подразумевает, что только один объект может владеть ресурсом в любой момент времени, в то время как семантика перемещения предоставляет возможность эффективно передавать владельца ресурса между объектами без необходимости копирования. Основные аспекты взаимодействия этих двух концепций:

1. Перемещение:
   - Перемещающий конструктор и оператор присваивания, основанные на исключительном владении, помогают эффективно "перемещать" ресурсы от одного объекта к другому. При перемещении происходит передача владения указателем на ресурс, а исходный объект сбрасывается в безопасное состояние.
2. Никакой копии:
   - В отличие от копирования объекта, которое находится под контролем создания новых экземпляров с их собственными копиями ресурсов, перемещение позволяет избежать избыточной работы, что особенно полезно для объектов, управляющих редкими или сложными ресурсами (например, большими массивами или сетевыми соединениями).
3. Безопасность управления памятью:
   - При правильной реализации перемещения не возникает риска двойного освобождения памяти, поскольку после перемещения исходный объект больше не владеет ресурсом. Это позволяет избежать утечек памяти и повышает безопасность кода.
4. Использование умных указателей:

- std::unique_ptr четко реализует стратегию исключительного владения и поддерживает семантику перемещения, автоматически обеспечивая правильное освобождение ресурсов при уходе объекта из области видимости или перемещении.

Сбалансированное использование исключительного владения и семантики перемещения в C++ обеспечивает эффективное и безопасное управление ресурсами, избегая производительных издержек, связанных с глубокой копией ресурсов.

20. Жизненный цикл ресурса и объекта-владельца ресурса

Жизненный цикл ресурса и объекта-владельца ресурса — это концепция, описывающая, как создаются, управляются и освобождаются ресурсы в процессе работы с объектами. Эта концепция становится особенно важной в C++, где управление памятью и ресурсами реализация зависит от выбранной стратегии владения. Основные этапы жизненного цикла включают:

1. Создание ресурса:
   - Ресурс (например, динамически выделенная память, файлы или сетевые сокеты) выделяется и инициализируется. Это может происходить либо в конструкторе объекта, который управляет этим ресурсом, либо в другой функции, которая создает ресурс и передает его объекту-владельцу.
2. Владение ресурсом:
   - Объект становится владельцем ресурса и несет полную ответственность за его корректное использование и освобождение. В зависимости от стратегии владения (исключительное или совместное) в разные моменты времени может существовать один или несколько владельцев.
3. Использование ресурса:
   - Объект взаимодействует с ресурсом — читает, записывает или производит другие операции, в зависимости от требований программы. На этом этапе важно обеспечить уникальный доступ к ресурсу (для исключительного владения) или синхронизированный доступ (для совместного владения).
4. Освобождение ресурса:
   - В момент, когда объект уничтожается или выходит из области видимости, он должен освободить управляемый ресурс. Для объектов с исключительным владением это происходит в деструкторе объекта, а для объектов с совместным владением освобождение происходит, когда все владельцы теряют ссылку на ресурс.
5. Безопасность и управление:
   - Важно разработать стратегии, которые обеспечивают корректное освобождение ресурсов и предотвращают утечки памяти и двойное освобождение. Это можно достичь с помощью использования умных указателей, следования правилам копирования и перемещения и контроля состояния объектов.

Понимание жизненного цикла ресурсов и объектов, управляющих этими ресурсами, критично для разработчиков C++, так как это помогает создавать более надежные и эффективные программы, минимизируя ошибки и издержки, связанные с управлением памятью.

21. Шаблон C++

Шаблоны в C++ представляют собой мощный механизм, позволяющий создавать обобщенные функции и классы, способные работать с различными типами данных без необходимости переписывать код для каждого нового типа. Шаблоны используют механизм подстановки типов во время компиляции и позволяют разработчикам создавать обобщенные алгоритмы и структуры данных. В C++ имеются два основных вида шаблонов:
1. Шаблоны функций:
- Шаблон функции определяет функцию, которая может принимать аргументы различного типа. Когда функция вызывается с конкретным типом, компилятор генерирует соответствующую версию функции. Пример шаблона функции:

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

2. Шаблоны классов:
- Шаблон класса позволяет создавать классы, которые могут работать с разными типами данных. Как и с функциями, компилятор создает конкретный класс на основе переданных типов. Пример шаблона класса:

```
template <typename T>
class Container {
private:
    T* elements;
    size_t size;
public:
    Container(size_t s) : size(s) { elements = new T[s]; }
    ~Container() { delete[] elements; }
    T& operator[](size_t index) { return elements[index]; }
};
```

Шаблоны также поддерживают специализацию, что позволяет создавать конкретные реализации для определенных типов, если требуется изменить поведение по сравнению с общей реализацией. Шаблоны позволяют создавать обобщенные алгоритмы, которые могут работать с различными типами данных, и обеспечивают безопасность типов во время компиляции.

22. Обзор промышленных программных библиотек
Существуют множество промышленных программных библиотек, которые упрощают разработку программного обеспечения, предоставляя готовые решения для распространенных задач. Вот некоторые из наиболее популярных библиотек в области разработки на C++:
1. Boost:
- Одна из самых известных библиотек, предоставляющая множество модулей для работы с потоками, контейнерами, файлами, алгоритмами, регулярными выражениями и многими другими задачами. Boost многие считают "золотым стандартом" для библиотек C++.
2. Qt:
- Мощная библиотека для разработки кроссплатформенных графических пользовательских интерфейсов (GUI) и приложений. Она включает поддержку сетевых протоколов, работы с базами данных и многопоточности.
3. OpenCV:
- Библиотека для компьютерного зрения и обработки изображений. OpenCV предоставляет функции для работы с изображениями, видео и компьютерным зрением, и широко используется в приложениях, связанных с анализом изображений и видео.
4. Poco:
- Кроссплатформенная библиотека для разработки сетевых приложений, которая включает в себя библиотеки для работы с HTTP, FTP, базами данных и многопоточности.
5. Eigen:
- Библиотека для линейной алгебры, работающая с матрицами и векторами, оптимизированная для выполнения математических операций и численного анализа.
6. TensorFlow и PyTorch (C++ интерфейсы):
- Хотя эти библиотеки в основном известны в контексте Python, они также предоставляют C++ API для выполнения задач машинного обучения и нейронных сетей.
7. fmt:
- Современная и высокоэффективная библиотека для форматирования строк. Она часто используется в проектах вместо стандартной c-style функции вывода, обеспечивая безопасность типов и лучшую производительность.
Еще одним важным аспектом является наличие библиотек для управления многопоточностью, такие как std::thread из стандартной библиотеки C++, и сторонние решения, такие как Intel Threading Building Blocks (TBB) для параллельного программирования.

23. Многопоточное программирование

Многопоточное программирование в C++ позволяет создавать приложения, которые могут выполнять несколько потоков выполнения одновременно, что приводит к улучшению производительности, особенно на многопроцессорных и многопоточных системах. В C++ поддержка многопоточности реализована через стандартную библиотеку, начиная с C++11. Основные концепты многопоточного программирования включают:
1. Потоки:
- Создание потока в C++ можно осуществить через класс std::thread. Поток может быть создан с функцией или лямбда-выражением, и каждый поток может выполняться независимо от других.

```
void threadFunction() {
    // Код выполнения потока
}

int main() {
    std::thread t(threadFunction);
    t.join(); // Ожидание завершения потока
    return 0;
}
```

2. Синхронизация:
- Для управления доступом к общим ресурсам между потоками необходимо использовать механизмы синхронизации. Это включает мьютексы (std::mutex), условия (std::condition_variable) и блокировки (std::unique_lock) для предотвращения гонок за ресурсами.
3. Обмен данными:
- Важно правильно организовать структуру обмена данными между потоками, чтобы избежать состязаний и обеспечить безопасность данных во время выполнения. Для проектов, использующих совместное владение, как std::shared_ptr, обязательно учитывать состояние и время жизни данных.
4. Параллельные алгоритмы:
- С C++17 появились возможности для реализации параллельных алгоритмов через стандартную библиотеку, такие как std::for_each с флагом std::execution::par, что позволяет выполнить операции параллельно.
5. Асинхронность:
- Асинхронное программирование позволяет создавать приложения, которые могут выполнять операции, не блокируя основной поток выполнения. Для этой цели можно использовать std::async и std::future для выполнения задач асинхронно и получения их результатов позже.
Многопоточное программирование эффективно для решения задач, требующих высокой производительности, таких как обработка больших объемов данных, веб-сервисы, игровые приложения и другие области, где параллельное выполнение задач критически важно. Однако с многопоточностью также связаны сложности, такие как управление состоянием, синхронизация и отладка, требующие тщательного планирования и понимания механик параллельного выполнения.

24. Управление потоками
Управление потоками в C++ включает в себя создание, взаимодействие и завершение потоков. В стандартной библиотеке C++11 и более поздних версиях предоставляются различные инструменты для работы с потоками. Основные аспекты управления потоками:
1. Создание потоков:
Поток можно создать через класс std::thread, который принимает функцию или объект класса с оператором () (оператор вызова) в качестве аргумента.

```
#include <thread>
#include <iostream>

void hello() {
    std::cout << "Hello from thread!" << std::endl;
}

int main() {
    std::thread t(hello); // Создание потока
    t.join(); // Ожидание завершения потока
    return 0;
}
```

2. Управление жизненным циклом потоков:

- join(): Блокирует вызывающий поток до тех пор, пока поток, к которому он обращается, не завершится. Ожидание завершения потока гарантирует, что все ресурсы будут правильно освобождены.
- detach(): Осуществляет отсоединение потока от основного потока, позволяя ему работать независимо. После вызова detach() нельзя ожидать, что поток завершится.

```
std::thread t(hello);
t.detach(); // Поток работает независимо
```

3. Проверка состояния потока:
Можно проверять, находится ли поток в рабочем состоянии с помощью функции, которая возвращает true, если поток активен и присоединен.

```
if (t.joinable()) {
    t.join();
}
```

4. Завершение потоков:
Завершение потока происходит автоматически при выходе из функции, на которую ссылается поток. Если поток выполняет бесконечный цикл или блокируется, может потребоваться другой способ завершения. Обычно рекомендуется использовать флаги или условия завершения.
5. Управление потоком на более высоком уровне:
В зависимости от задачи можно использовать более высокоуровневые библиотеки, такие как Intel Threading Building Blocks (TBB) или OpenMP, которые предоставляют более мощные параллельные конструкты.

25. Разделение данных между потоками
Разделение данных между потоками – это задача, требующая внимания при проектировании параллельных программ. Эффективное разделение данных улучшает производительность и уменьшает необходимость в синхронизации. Вот основные стратегии разделения данных между потоками:
1. Разделение на блоки:
Разделение данных на блоки, которые могут обрабатываться параллельно. Например, в алгоритме обработки массива можно разбить массив на несколько подмассивов и дать каждому потоку обрабатывать свою часть.

```
void process(int* data, size_t start, size_t end) {
    for (size_t i = start; i < end; ++i) {
        // Обработка элемента
    }
}

int main() {
    const size_t size = 100;
    int data[size];
    std::thread t1(process, data, 0, size / 2);
    std::thread t2(process, data, size / 2, size);
    t1.join();
    t2.join();
}
```

2. Стратегия "разделяй и властвуй":
Использование алгоритмов, которые делят задачу на меньшие части. Это позволяет каждому потоку работать над своей частью задачи, что может эффективно использовать многопоточность.
3. Локальные копии:
Каждому потоку можно предоставить локальные копии данных, следующих по определенному правилу. Это исключает необходимость синхронизации и снижает конкуренцию за общий ресурс.
4. Работа с коллекциями:
Использование потокобезопасных структур данных, таких как concurrent_vector из TBB или ThreadSafeQueue, которые могут безопасно разделять данные и управлять доступом к ним.
5. Стратегия "пул задач":
При использовании пулов задач потоки могут забирать задачи из очереди, что позволяет эффективно разделять нагрузку между потоками, с минимальным вредом от блокировки.

26. Синхронизация параллельных операций
Синхронизация параллельных операций — важный аспект многопоточного программирования, предотвращающий проблемы состояния гонки, обеспечивая согласованность данных. В C++ стандартная библиотека предоставляет различные механизмы для синхронизации:

1. Мьютексы:
Мьютексы (std::mutex) используют для контроля доступа к разделяемым ресурсам. Перед доступом к ресурсу поток должен заблокировать мьютекс, а после работы — разблокировать.

```
std::mutex mtx;

void safe_function() {
    mtx.lock();
    // Доступ к общему ресурсу
    mtx.unlock();
}
```

2. Локальные блокировки:
Использование std::lock_guard или std::unique_lock для автоматического управления захватом и освобождением мьютекса, что упрощает защиту областей критической секции.

```
void safe_function() {
    std::lock_guard<std::mutex> lock(mtx); // Автоматическая разблокировка
    // Доступ к общему ресурсу
}
```

3. Условные переменные:
Условные переменные (std::condition_variable) позволяют потокам ожидать, пока не будут выполнены определенные условия. Это полезно в ситуациях, когда нужно подождать, например, пока данные не будут доступны.

```
std::condition_variable cv;
bool ready = false;

void wait_for_signal() {
    std::unique_lock<std::mutex> lock(mtx);
    cv.wait(lock, [] { return ready; });
    // Дальнейшая обработка
}

void signal() {
    std::unique_lock<std::mutex> lock(mtx);
    ready = true;
    cv.notify_all(); // Пробуждение ожидающих потоков
}
```

4. Семафоры и атомарные операции:
Несмотря на то, что стандартные семафоры не включены в C++, многие библиотеки предлагают свои реализации семафоров. Атомарные операции (std::atomic) используются для обеспечения корректного чтения и записи в переменные, предотвращая состояние гонки без блокировок.
5. Избегание мертвых блокировок:
Один из важных аспектов синхронизации — избегать ситуации, когда два или более потока блокируются, ожидая друг друга. Некоторые стратегии включают правильное определение порядка блокировок и использование тайм-аутов.
Синхронизация между потоками является критически важным аспектом многопоточного программирования, требующим тщательного планирования и реализации для достижения оптимальной производительности и безопасности данных.

27. Проектирование параллельных программ
Проектирование параллельных программ требует особого внимания к архитектуре и структуре приложения для эффективного использования многопоточности. Важные аспекты проектирования включают:
1. Определение задач:
- Начните с анализа задачи и выделения отдельных частей, которые могут быть выполнены параллельно. Разделите работу на независимые задачи, каждая из которых может выполняться в своем потоке.
2. Выбор модели параллелизма:
- Определите подходящую модель параллелизма для вашей задачи. Это может быть:
  - Параллельные задачи: различные потоки выполняют отдельные функции (например, обработка данных).
  - Параллельные данные: один и тот же алгоритм применяется к различным частям данных параллельно.

- Потокобезопасные операции: использование общей структуры данных, к которой одновременно обращаются несколько потоков с механизмами синхронизации.

3. Управление данными:
- Спроектируйте доступ к данным так, чтобы минимизировать количество конфликтов и необходимых блокировок. Используйте разделение данных и локальные копии, чтобы избежать конкуренции за ресурсы. Подумайте о потокобезопасных контейнерах или алгоритмах, которые уменьшают необходимость блокировок.

4. Правило минимизации блокировок:
- По возможности используйте как можно меньше блокировок, чтобы избежать потенциальных проблем с мертвыми блокировками и снизить накладные расходы на синхронизацию.

5. Тестирование на масштабируемость:
- Убедитесь, что ваше решение масштабируемо и может эффективно использовать ресурсы системы. Проектируйте алгоритмы и структуры так, чтобы они могли адаптироваться к увеличению числа потоков или количеству данных.

6. Обработка ошибок и исключений:
- Проектируйте приложение с учетом обработки ошибок и исключений в каждом потоке. Обеспечьте механизм для безопасности в случае сбоя, чтобы избежать неконсистентного состояния приложения.

7. Сложность отладки:
- Понимайте, что отладка многопоточных программ может быть более сложной из-за сложного взаимодействия потоков. Поэтому инженерные проекты должны включать возможности для журналирования и мониторинга, чтобы облегчить диагностику проблем.

28. Продвинутое управление потоками

Продвинутое управление потоками охватывает более сложные аспекты многопоточного программирования. Некоторые из ключевых областей, связанных с передовыми подходами, включают:

1. Пулы потоков:
- Использование пулов потоков (thread pools) для управления потоками может существенно снизить накладные расходы, связанные с созданием и уничтожением потоков. Пул потоков управляет безопасным набором потоков, которые могут обрабатывать несколько задач.

2. Синхронные и асинхронные задачи:
- Использование как синхронных, так и асинхронных подходов для выполнения задач. Например, std::async позволяет выполнять задачи асинхронно и получить результат позднее через std::future.

3. Распределенное программирование:
- Исследование применения потоков в распределенных системах. Когда данные или задачи распределены между несколькими машинами, необходимо учитывать управление потоками и синхронизацию на уровне сети.

4. Функциональные подходы:
- Использование функционального программирования и imutability для уменьшения проблем с изменяемым состоянием. Это позволяет проще управлять параллельными вычислениями и минимизировать необходимость синхронизации.

5. Управление приоритетами потоков:
- Некоторые платформы могут поддерживать управление приоритетами для потоков, что позволяет изменять порядок выполнения потоков в зависимости от важности их задач.

6. Смарт-блокировки (lock-free):
- Использование алгоритмов и структур данных без блокировок, чтобы избежать стандартных проблем, связанных с блокировками. Это может улучшить производительность и отзывчивость.

7. Мониторинг и профилирование:
- Инструменты мониторинга и профилирования позволяют отслеживать производительность многопоточных приложений, выявлять узкие места и определять области для оптимизации.

29. Тестирование и отладка многопоточных приложений

Тестирование и отладка многопоточных приложений — это критически важные этапы разработки, которые могут быть сложными из-за непредсказуемого поведения потоков. Рассматриваемые методы включают:

1. Задание ясных целей тестирования:
- Определите, что именно вы хотите протестировать: корректность выполнения, производительность, устойчивость к нагрузке и т. д.

2. Использование встроенных средств отладки:
- Многие IDE и инструменты отладки поддерживают многопоточность и позволяют наблюдать за действиями потоков. Используйте точки останова и трассировку для отслеживания выполнения программы.

3. Проверка состояния гонки:
- Используйте инструменты статического и динамического анализа, такие как Thread Sanitizer, для выявления состояний гонки, ошибок синхронизации и мертвых блокировок.

4. Симуляция нагрузки:

- Создание тестов на нагрузку, которые симулируют сценарии использования с высоким уровнем параллелизма. Это помогает выявить узкие места и проблемы, которые могут возникнуть при работе приложения под нагрузкой.

5. Логирование:
   - Включите механизм логирования, который фиксирует время и источник событий в приложении. Это поможет в отладке, когда возникают ошибки или неконсистентность данных.
6. Модульное тестирование:
   - Разработка отдельных тестов для функций потоков. Работайте с потоками в изолированном окружении, чтобы протестировать их поведение отдельно от остального приложения.
7. Изоляция потока:
   - В некоторых случаях может быть полезно изолировать потоки для тестирования, чтобы минимизировать влияние других потоков на результаты тестов.
8. Ручное тестирование:
   - Проведение ручных тестов для проверки многопоточных сценариев может быть полезным, особенно для проверки успешных и неуспешных сценариев, таких как обработка ошибок.

Комплексный подход к тестированию и отладке многопоточных приложений позволит выявить и исправить проблемы до развертывания, что обеспечит высокую производительность и надежность в производственной среде.

30. Графический пользовательский интерфейс

Графический пользовательский интерфейс (GUI) – это способ взаимодействия пользователя с компьютерными программами с помощью графических элементов, таких как окна, кнопки, меню и другие контролы. Основные компоненты графического интерфейса включают:

1. Элементы управления:
   - Кнопки, текстовые поля, флажки, выпадающие списки и другие элементы, позволяющие пользователю взаимодействовать с приложением.
2. Окна:
   - Основная контейнерная единица GUI, во многих случаях включает заголовок, меню, область взаимодействия и статусную строку.
3. События:
   - Механизм обработки пользовательского ввода, например клик мышью, нажатие клавиш и другие действия, которые вызывают выполнение определенных функций.
4. Меню и панель инструментов:
   - Предоставляют доступ к функциям программы и действию пользователю через содержимое меню, которое может открываться или быть постоянно видно.
5. Отрисовка и графика:
   - Компьютерные графические API, такие как OpenGL или DirectX, могут использоваться для создания сложной визуализации, а также для обработки изображений и анимации.

31. Проектирование графического пользовательского интерфейса на C++

Проектирование графического пользовательского интерфейса на C++ требует осознания выбора библиотек и инструментов, которые помогут создать мощный и удобный интерфейс. Основные шаги для проектирования GUI на C++:

1. Выбор библиотеки:
   - Наиболее популярные библиотеки для создания GUI на C++ включают:
     - Qt: мощная кроссплатформенная библиотека с обширным набором компонентов и средств разработки.
     - wxWidgets: кроссплатформенная библиотека, позволяющая создавать нативные GUI для различных платформ.
     - GTK+: библиотека для создания графических интерфейсов, которая часто используется в приложениях для Linux.
2. Создание макета интерфейса:
   - Используйте средства проектирования интерфейсов, такие как Qt Designer или wxFormBuilder, чтобы визуально спроектировать макет вашего приложения.
3. Разработка логики приложения:
   - Связывайте элементы управления с логикой приложения, определяя обработчики событий для реакций на действия пользователя.

```
// Пример обработки события кнопки в Qt
connect(button, &QPushButton::clicked, this, &MyClass::onButtonClicked);
```

4. Адаптация интерфейса:
   - Проектируйте интерфейс с учетом различных разрешений и размеров экранов, обеспечьте его отзывчивость и удобство в использовании на всех устройствах.
5. Тестирование UI:
   - Используйте методы юзабилити-тестирования для сбора отзывов от пользователей и внесения улучшений в интерфейс.

32. Инструментарий разработчика
Правильный инструментарий разработчика решает множество задач, связанных с разработкой, от создания и компиляции кода до отладки и тестирования. Основные инструменты, которые могут быть полезны при разработке приложений на C++, включают:
1. IDE (интегрированные среды разработки):
- Популярные IDE, такие как Visual Studio, Qt Creator, Code::Blocks, CLion, предоставляют мощные инструменты для написания и отладки кода.
2. Компиляторы:
- Компиляторы, такие как GCC, Clang или MSVC, необходимы для компиляции вашего C++ кода и создания исполняемых файлов.
3. Системы контроля версий:
- Git и другие системы контроля версий являются неотъемлемой частью современного разработки, обеспечивая управление версиями и совместную работу.
4. Инструменты отладки:
- GDB для отладки приложений на C++, а также встроенные отладчики в IDE, которые позволяют проверять состояние программы во время выполнения и отслеживать ошибки.
5. Профайлеры:
- Инструменты профилирования, такие как Valgrind, gprof или встроенные инструменты в IDE, помогают анализировать производительность и находить узкие места.
6. Библиотеки:
- Используйте внешние и стандартные библиотеки (например, Boost, STL), которые предоставляют готовые функции и структуры для облегчения разработки.
7. Документация:
- Используйте инструменты, такие как Doxygen, для генерации документации на основе вашего кода, чтобы облегчить понимание для будущих разработчиков.

33. Разбор существующего проекта с открытым исходным кодом
Разбор существующего проекта с открытым исходным кодом — это отличная возможность изучить архитектуру, кодовую базу и применяемые практики. Основные шаги для исследования проекта с открытым исходным кодом:
1. Выбор проекта:
- Начните с выбора проекта, который вас интересует и соответствует вашим целям обучения, будьте внимательны к языку программирования, стилю и структуре.
2. Изучение документации:
- Начните с чтения файлов документации, таких как README, CONTRIBUTING и Wiki, чтобы понять цель проекта и его архитектуру.
3. Запуск проекта:
- Склонируйте репозиторий и следуйте инструкциям по установке и запуску проекта на своем компьютере. Это позволит вам увидеть, как проект работает на практике.
4. Изучение архитектуры:
- Анализируйте структуру проекта, файловую систему и основные модули. Обратите внимание на то, как организована кодовая база и как различные компоненты взаимодействуют друг с другом.
5. Понимание кода:
- Читайте код, внимательно изучая его логику, структуру и используемые паттерны проектирования. Попробуйте написать комментарии или модифицировать код, чтобы лучше понять, как все работает.
6. Контрибьюция:
- Если вы чувствуете себя готовым, попробуйте внести свой вклад в проект. Это может быть исправление ошибок, добавление функций или улучшение документации.
7. Общение с сообществом:
- Присоединяйтесь к сообществу проекта, участвуйте в обсуждениях и задавайте вопросы. Это не только поможет вам лучше понять проект, но и расширит вашу сеть контактов.

**4.2.2. Практическое задание**

**4.2.2.1. Порядок проведения.**

Практические навыки проверяются путём выполнения обучающимися практических заданий в условиях, полностью или частично приближенных к условиям профессиональной деятельности. Проверяется знание теоретического материала, необходимое для правильного совершения необходимых действий, умение выстроить последовательность действий, практическое владение приёмами и методами решения профессиональных задач.

**4.2.2.2. Критерии оценивания**

**27-30 баллов ставится, если обучающимся:**
Задание выполнено полностью и правильно.
**22-26 баллов ставится, если обучающимся:**
Задание выполнено полностью, но нет достаточного обоснования. Или при верном решении допущена ошибка или недочет, не влияющий на правильную последовательность рассуждений.

**18-21 баллов ставится, если обучающимся:**
Задание выполнено частично или с фактическими ошибками.
**0-17 баллов ставится, если обучающимся:**
Задание не выполнено или выполнено с большим количеством фактических ошибок.

**4.2.3.3. Содержание оценочного средства**

1) Задание
Используя шаблоны функций и частичную специализацию шаблонов добиться корректного использования пользовательского класса в коллекциях unordered_map, unordered_set.
Описание
Неупорядоченные коллекции реализованы на базе хеш-таблиц. Для манипулирования элементами (ключами) коллекции необходимо уметь строить хеш от элемента и сравнивать два элемента коллекции. Для встроенных или стандартных типов данных хеш-функция и операция сравнения уже определены. Для пользовательских классов хеш-функцию и сравнение необходимо реализовать самостоятельно.
Пример

```
#include <unordered_set>
class Point {
    int x, y;
};
// ...
// Point p, p1, p2;
// хеш от p: p.x^p.y,
// сравнение p1 и p2:  (p1.x==p2.x) && (p1.y==p2.y)
// ...

std::unordered_set < Point > s;
```

2) Задание. Реализовать обобщённый итерационный алгоритм вычисления квадратного корня (и не только), в котором все детали вынесены в функторы, в виде функции iterate.
Потребуется три функтора:
initial – функтор, вычисляющий начальное приближение. Для квадратного корня из x можно взять x/2.
next – функтор, строящий следующее приближение. Для квадратного корня: (y + x/y)/2, где x – аргумент функции корня, а y – текущее приближение.
enough – предикат, оценивающий точность. Если он возвращает true, итерационную процедуру можно завершить. Для квадратного корня можно взять fabs(x - y*y) < 1e-8, где x и y имеют тот же смысл, что и выше.
Таким образом, шаблонная функция iterate будет иметь четыре параметра шаблона: тип значения и три функтора.

3) Задание
В директории лежат входные текстовые файлы, поименованные следующим образом: in_<N>.dat, где N - натуральное число. Каждый файл состоит из двух строк. В первой строке - число, обозначающее действие, а во второй - числа с плавающей точкой, разделенные пробелом.
Действия могут быть следующими:
1 - сложение
2 - умножение
3 - сумма квадратов
Необходимо написать многопоточное приложение, которое выполнит требуемые действия над числами и сумму результатов запишет в файл out.dat. Название рабочей директории передается в виде аргумента рабочей строки. В реализации приветствуется использование полиморфизма и паттернов проектирования.

4) Задание
Разработайте многопоточное приложение, выполняющее вычисление произведения матриц A (m×n) и B (n×k). Элементы $c_{ij}$ матрицы произведения C = A×B вычисляются параллельно p однотипными потоками. Если некоторый поток уже вычисляет элемент $c_{ij}$ матрицы C, то следующий приступающий к вычислению поток выбирает для расчета элемент $c_{ij+1}$, если j<k, и $c_{i+1k}$, если j=k. Выполнив вычисление элемента матрицы-произведения, поток проверяет, нет ли элемента, который еще не рассчитывается. Если такой элемент есть, то приступает к его расчету. В противном случае отправляет (пользовательское) сообщение о завершении своей работы и приостанавливает своё выполнение. Главный поток, получив сообщения о завершении вычислений от всех потоков, выводит результат на экран и запускает поток, записывающий результат в конец файла-протокола. В каждом потоке должна быть задержка в выполнении вычислений (чтобы дать возможность поработать всем потокам). Синхронизацию потоков между собой организуйте через критическую секцию или мьютекс.

5) Задание.

При нажатии кнопки "Start" диалоговое приложение запускает консольное приложение. Последующие нажатия кнопки "Start" должны привести к созданию в консольном приложении N новых рабочих потоков, где N - значение из поля с числовым счетчиком.

Нажатие кнопки "Stop" должно привести к закрытию последнего созданного рабочего потока в консольном приложении, а в случае отсутствия рабочих потоков - к его завершению. Консольное приложение также должно завершиться и при завершении диалогового. Диалоговое приложение должно отслеживать, закрыл ли пользователь консольное приложение самостоятельно. Следующее после закрытия консоли нажатие "Start" возобновляет рабочий цикл.

После запуска консоли выпадающий список содержит строки "Все потоки" и "Главный поток", по мере создания новых потоков в него добавляются строки, содержащие их номера (разумеется, при удалении потоков удаляются и соответствующие им строки). Корректировка выпадающего списка осуществляется лишь после получения подтверждения от консоли об успешном создании потоков.

Взаимодействие между приложениями реализовать с помощью объектов ядра "события" (events).

Добавить в приложения передачу информации с помощью файла, отображаемого в память (memory mapped file). Реализовать транспортные функции в виде динамически подключаемой библиотеки с неявной загрузкой.

При нажатии кнопки "Send" диалоговое приложение пересылает консоли текст, введенный в поле ввода. Выпадающий список задает поток-адресат. При выборе строки "Все потоки" сообщение отправляется всем потокам, каждый поток считывает информацию независимо.

Главный поток выводит полученный текст на стандартный вывод (stdout), рабочие потоки создают текстовые файлы с именами <свой номер>.txt и дописывают в них полученный текст. Файл, отображаемый в память, защищается от совместного использования с помощью объекта синхронизации mutex.

Все действия подтверждаются сообщениями от консоли.

**4.2.3.4. Ключи к практическим заданиям**

Задание 1

Для корректного использования пользовательского класса Point в неупорядоченных коллекциях, таких как unordered_set и unordered_map, необходимо определить хеш-функцию и оператор сравнения. Ниже приведен пример реализации:

```
#include <iostream>
#include <unordered set>
#include <functional>

class Point {
public:
    int x, y;
    Point(int xCoord, int yCoord) : x(xCoord), y(yCoord) {}

    // Оператор сравнения
    bool operator==(const Point& other) const {
        return (x == other.x && y == other.y);
    }
};

namespace std {
    template <>
    struct hash<Point> {
        size_t operator()(const Point& p) const {
            return hash<int>()(p.x) ^ hash<int>()(p.y); // Комбинирование хешей
        }
    };
}

int main() {
    std::unordered_set<Point> points;

    points.insert(Point(1, 2));
    points.insert(Point(3, 4));
```

```
    // Проверка наличия точки
    if (points.find(Point(1, 2)) != points.end()) {
        std::cout << "Point (1, 2) exists in the set." << std::endl;
    }

    return 0;
}
```

В этом примере в классе Point переопределен оператор == для сравнения объектов Point по координатам. В специальной структуре hash<Point> реализуется метод operator(), который генерирует хеш-значение для объекта Point. Для этого используется стандартная хеш-функция hash<int>() для координат x и y, которые комбинируются с помощью побитовой операции XOR. Теперь объекты Point можно добавлять в unordered_set и выполнять стандартные операции, такие как поиск.

Задание 2

ля реализации обобщенного итерационного алгоритма вычисления квадратного корня с использованием функторов, необходимо создать три функтора, а также шаблонную функцию iterate, которая будет принимать эти функторы в качестве параметров. Ниже представлен пример реализации.

```
#include <iostream>
#include <cmath>
#include <functional>

// Функтор для вычисления начального приближения
struct Initial {
    template<typename T>
    T operator()(T x) const {
        return x / 2; // Начальное приближение
    }
};

// Функтор для вычисления следующего приближения
struct Next {
    template<typename T>
    T operator()(T x, T y) const {
        return (y + x / y) / 2; // Следующее приближение
    }
};

// Функтор для оценки точности
struct Enough {
    template<typename T>
    bool operator()(T x, T y) const {
        return fabs(x - y * y) < 1e-8; // Условие завершения
    }
};

// Шаблонная функция для итерации
template<typename T, typename InitialFunctor, typename NextFunctor, typename
EnoughFunctor>
T iterate(T x, InitialFunctor init, NextFunctor next, EnoughFunctor enough) {
    T y = init(x); // Начальное приближение
    while (!enough(x, y)) { // Проверка на завершение
        y = next(x, y); // Обновление приближения
    }
    return y; // Возврат результата
}

int main() {
    double x = 25.0;

    // Использование обобщённого итерационного алгоритма
    double result = iterate(x, Initial(), Next(), Enough());

    std::cout << "Квадратный корень из " << x << " = " << result << std::endl;
```

```
    return 0;
}
```

В этом примере представлены следующие компоненты:

1. Функтор Initial: Вычисляет начальное приближение для квадратного корня (x/2).
2. Функтор Next: Вычисляет следующее приближение с использованием формулы (y + x/y)/2.
3. Функтор Enough: Проверяет, достигнуто ли достаточное приближение, с использованием условия fabs(x - y*y) < 1e-8.
4. Шаблонная функция iterate: Принимает аргумент x, а также три функтора. Она использует функторы для вычисления начального значения, итеративного обновления и проверки условия завершения.

В функции main демонстрируется использование алгоритма для вычисления квадратного корня из числа 25. Результат выводится на экран.

Таким образом, приспособление к пользовательским функтором позволяет адаптировать алгоритм под различные задачи, не ограничиваясь только вычислением квадратного корня.

Задание 3
Для реализации многопоточного приложения, выполняющего указанные действия над числами из текстовых файлов, мы можем воспользоваться полиморфизмом и применить паттерн проектирования "Стратегия". Вот пример реализации на C++:
1. Определяем интерфейс и стратегии
Создадим базовый класс Operation, который будет иметь виртуальный метод для выполнения операции, а также три подкласса для каждой из операций.

```
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <vector>
#include <thread>
#include <mutex>
#include <filesystem>
#include <memory>

// Интерфейс для операций
class Operation {
public:
    virtual double execute(const std::vector<double>& numbers) = 0;
    virtual ~Operation() = default;
};

// Стратегия для сложения
class Addition : public Operation {
public:
    double execute(const std::vector<double>& numbers) override {
        double result = 0.0;
        for (double num : numbers) {
            result += num;
        }
        return result;
    }
};

// Стратегия для умножения
class Multiplication : public Operation {
public:
    double execute(const std::vector<double>& numbers) override {
```

```
        double result = 1.0;
        for (double num : numbers) {
            result *= num;
        }
        return result;
    }
};

// Стратегия для суммы квадратов
class SumOfSquares : public Operation {
public:
    double execute(const std::vector<double>& numbers) override {
        double result = 0.0;
        for (double num : numbers) {
            result += num * num;
        }
        return result;
    }
};

// Мьютекс для защиты доступа к общей переменной
std::mutex mtx;

// Функция обработки файла
void process_file(const std::string& filename, double& total) {
    std::ifstream input_file(filename);
    if (!input_file.is_open()) {
        std::cerr << "Не удалось открыть файл: " << filename << std::endl;
        return;
    }

    int action;
    std::string line;
    std::getline(input_file, line);
    std::istringstream(line) >> action;

    std::getline(input_file, line);
    std::istringstream iss(line);
    std::vector<double> numbers;
    double num;
    while (iss >> num) {
        numbers.push_back(num);
    }

    std::unique_ptr<Operation> operation;
    switch (action) {
        case 1:
            operation = std::make_unique<Addition>();
            break;
        case 2:
            operation = std::make_unique<Multiplication>();
            break;
        case 3:
            operation = std::make_unique<SumOfSquares>();
            break;
        default:
            std::cerr << "Неизвестное действие в файле: " << filename <<
std::endl;
            return;
    }

    double result = operation->execute(numbers);

    // Обновляем общую сумму в защищенной области
```

```
    std::lock_guard<std::mutex> lock(mtx);
    total += result;
}

// Главная функция
int main(int argc, char* argv[]) {
    if (argc != 2) {
        std::cerr << "Использование: " << argv[0] << " <directory_path>" <<
std::endl;
        return 1;
    }

    std::string directory_path = argv[1];
    double total = 0.0;
    std::vector<std::thread> threads;

    // Перебор файлов в указенной директории
    for (const auto& entry : std::filesystem::directory_iterator(directory_path))
{
        const std::string filename = entry.path().string();
        if (std::filesystem::is_regular_file(entry) && filename.ends_with(".dat"))
{
            threads.emplace_back(process_file, filename, std::ref(total));
        }
    }

    // Ожидание завершения всех потоков
    for (auto& t : threads) {
        t.join();
    }

    // Запись итоговой суммы в файл
    std::ofstream output_file("out.dat");
    if (output_file.is_open()) {
        output_file << "Итоговая сумма: " << total << std::endl;
    } else {
        std::cerr << "Не удалось открыть файл out.dat для записи." << std::endl;
    }

    return 0;
}
```

Объяснение кода:

1. Интерфейс и стратегии: Создан базовый класс Operation с виртуальным методом execute. Созданы три класса производных (Addition, Multiplication, SumOfSquares), которые реализуют конкретные математические операции.

2. Функция process_file: Читает файл, определяет, какая операция должна быть выполнена, и использует соответствующий класс стратегии для выполнения операции. Результат сохраняется в общей переменной, защищаемой мьютексом.

3. Основная логика: Программа принимает путь к директории через аргументы командной строки, перебирает файлы, соответствующие шаблону in_<N>.dat, и создает потоки для обработки каждого файла. После завершения всех потоков результат сохраняется в файл out.dat.

Таким образом, данная реализация демонстрирует использование полиморфизма и многопоточности в обработке файлов, обеспечивая эффективное и безопасное выполнение различных операций.

Задание 4
Для разработки многопоточного приложения, вычисляющего произведение матриц A и B, мы будем использовать C++ с поддержкой многопоточности и синхронизации. Программа будет распараллеливать вычисления элементов

матрицы C, а потоки будут работать с задержкой для моделирования реального времени выполнения операций.

```
#include <iostream>
#include <vector>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <chrono>
#include <atomic>
#include <iomanip>
#include <fstream>

// Глобальные переменные
std::mutex mtx; // Мьютекс для синхронизации
std::condition_variable cv; // Условная переменная
std::atomic<int> current_index{0}; // Текущий индекс элемента матрицы C
std::vector<std::vector<int>> C; // Результирующая матрица

// Функция для генерации случайных матриц
std::vector<std::vector<int>> generate_matrix(int rows, int cols) {
    std::vector<std::vector<int>> matrix(rows, std::vector<int>(cols));
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            matrix[i][j] = rand() % 10; // Элементы от 0 до 9
        }
    }
    return matrix;
}

// Функция для умножения матриц
void multiply_matrices(const std::vector<std::vector<int>>& A,
                       const std::vector<std::vector<int>>& B,
                       int thread_id, int m, int n, int k) {
    while (true) {
        // Захват мьютекса для доступа к текущему индексу
        std::unique_lock<std::mutex> lock(mtx);
        int idx = current_index++;// Получаем текущий индекс и увеличиваем его
        if (idx >= m * k) { // Проверка выхода за границы матрицы
            lock.unlock();
            break; // Если индексы превышают пределы, выходим из цикла
        }

        lock.unlock();

        // Преобразуем индекс в двумерные координаты
        int i = idx / k; // Индекс строки
        int j = idx % k; // Индекс столбца

        // Умножение и задержка для имитации длительной операции
        for (int p = 0; p < n; ++p) {
            C[i][j] += A[i][p] * B[p][j];
        }

        // Имитация задержки в вычислении
        std::this_thread::sleep_for(std::chrono::milliseconds(100));

        // Уведомляем о завершении работы текущего потока
        std::cout << "Поток " << thread_id << " закончил вычисление элемента C["
<< i << "][" << j << "]" << std::endl;
    }
}

void log_result(const std::vector<std::vector<int>>& result, const std::string&
filename) {
    std::ofstream log_file(filename, std::ios::app);
```

```
    if (log_file.is_open()) {
        log_file << "Результат матричного произведения:\n";
        for (const auto& row : result) {
            for (const auto& elem : row) {
                log_file << std::setw(4) << elem; // Форматированный вывод
            }
            log_file << "\n";
        }
        log_file << "\n";
        log_file.close();
    } else {
        std::cerr << "Не удалось открыть файл для записи!" << std::endl;
    }
}

int main() {
    int m = 3, n = 3, k = 3; // Размерности матриц
    // Генерация случайных матриц A и B
    auto A = generate_matrix(m, n);
    auto B = generate_matrix(n, k);

    // Инициализация матрицы C нулями
    C.assign(m, std::vector<int>(k, 0));

    // Количество потоков
    int num_threads = 4;
    std::vector<std::thread> threads;

    // Создание и запуск потоков
    for (int i = 0; i < num_threads; ++i) {
        threads.emplace_back(multiply_matrices, std::cref(A), std::cref(B), i + 1,
m, n, k);
    }

    // Ожидание завершения всех потоков
    for (auto& t : threads) {
        t.join();
    }

    // Вывод результата на экран
    std::cout << "Результат матричного произведения:\n";
    for (const auto& row : C) {
        for (const auto& elem : row) {
            std::cout << std::setw(4) << elem; // Форматированный вывод
        }
        std::cout << "\n";
    }

    // Запись результата в файл
    log_result(C, "log.txt");

    return 0;
}
```

Объяснение программы:

1. Генерация случайной матрицы: Функция generate_matrix создает матрицы с целыми числами от 0 до 9.
2. Умножение матриц: Функция multiply_matrices обрабатывает элементы матрицы C. Параметр current_index отслеживает, какой элемент сейчас рассчитывается. Если индексы выходят за пределы, поток завершает свою работу.
3. Потоки: В main создаются и запускаются несколько потоков, используя std::thread, который выполняет функцию multiply_matrices.
4. Синхронизация: Для синхронизации используется мьютекс, который обеспечивает безопасный доступ к current_index.

5. Запись результата: После завершения всех потоков результат выводится на экран и записывается в файл log.txt.
6. Задержка: После каждого вычисления элемента матрицы, поток приостанавливается на 100 миллисекунд, чтобы имитировать процесс вычисления и позволить другим потокам работать.

Задание 5

Создание описанного многопоточного приложения с использованием Windows API и динамически подключаемых библиотек (DLL) — сложная задача, требующая серьезных усилий. Давайте разобьем ее на основные части.
Общая структура приложения
1. Диалоговое приложение (GUI):
- Интерфейс с кнопками "Start", "Stop", "Send" и полем для ввода текста.
- Выпадающий список для выбора потока.
- Обработчики событий для кнопок, для управления созданием и остановкой потоков, а также отправки сообщений.
2. Консольное приложение:
- Создаёт рабочие потоки.
- Принимает команды на создание/остановку потоков.
- Получает текстовое сообщение и обрабатывает его в зависимости от адресата (все потоки или конкретный поток).
- Создаёт текстовые файлы для хранения полученных сообщений.
3. Динамически подключаемая библиотека (DLL):
- Реализует функции для обмена данными между приложениями и синхронизирует доступ к разделяемым ресурсам.
4. Объекты ядра "события" и "мьютексы" для синхронизации между процессами.
5. Файлы, отображаемые в память, для пересылки данных.
Код реализации
Для наглядности я приведу фрагменты кода для каждого из компонентов.
1. Диалоговое приложение

```
#include <windows.h>
#include <commctrl.h>
#include <string>
#include <vector>
#include <thread>
#include <iostream>
#include <fstream>
#include "Resource.h" // Файл ресурсов для диалогового приложения

// Объявляем функции для работы с DLL
typedef void(*StartThreadsFunc)(int);
typedef void(*StopThreadFunc)();
typedef void(*SendMessageToThreadFunc)(const char*, int);
typedef void(*ShutDownThreadsFunc)();

HINSTANCE hDLL;
StartThreadsFunc StartThreads;
StopThreadFunc StopThread;
SendMessageToThreadFunc SendMessageToThread;
ShutDownThreadsFunc ShutDownThreads;

// Глобальные переменные
HWND hCombo;
HINSTANCE hInst;
std::vector<std::string> threadsList;

// Пользовательский диалог
INT_PTR CALLBACK DialogProc(HWND hWndDlg, UINT message, WPARAM wParam, LPARAM
lParam) {
    switch (message) {
        case WM_INITDIALOG:
```

```
            // Загрузка DLL
            hDLL = LoadLibrary(TEXT("ThreadLibrary.dll"));
            StartThreads = (StartThreadsFunc)GetProcAddress(hDLL, "StartThreads");
            StopThread = (StopThreadFunc)GetProcAddress(hDLL, "StopThread");
            SendMessageToThread = (SendMessageToThreadFunc)GetProcAddress(hDLL,
"SendMessageToThread");
            ShutDownThreads = (ShutDownThreadsFunc)GetProcAddress(hDLL,
"ShutDownThreads");
            hCombo = GetDlgItem(hWndDlg, IDC_COMBO1);
            SendMessage(hCombo, CB_ADDSTRING, 0, (LPARAM)"Все потоки");
            SendMessage(hCombo, CB_ADDSTRING, 0, (LPARAM)"Главный поток");
            return TRUE;

        case WM_COMMAND:
            if (LOWORD(wParam) == IDC_BUTTON_START) {
                // Получаем значение счетчика, установленного в интерфейсе
                int threadCount = GetDlgItemInt(hWndDlg, IDC_NUM_COUNT, NULL,
FALSE);
                StartThreads(threadCount);
            }
            else if (LOWORD(wParam) == IDC_BUTTON_STOP) {
                StopThread();
            }
            else if (LOWORD(wParam) == IDC_BUTTON_SEND) {
                char message[256];
                GetDlgItemText(hWndDlg, IDC_EDIT_MSG, message, sizeof(message));
                int selectedIndex = SendMessage(hCombo, CB_GETCURSEL, 0, 0);
                SendMessageToThread(message, selectedIndex);
            }
            break;

        case WM_CLOSE:
            ShutDownThreads();
            FreeLibrary(hDLL);
            EndDialog(hWndDlg, 0);
            return TRUE;
    }
    return FALSE;
}
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE, LPSTR, int nCmdShow) {
    hInst = hInstance;
    DialogBox(hInst, MAKEINTRESOURCE(IDD_DIALOG1), NULL, DialogProc);
    return 0;
}
```

2. Консольное приложение

```
#include <iostream>
#include <thread>
#include <vector>
#include <windows.h>

HANDLE hEvent;
std::mutex mtx;
std::vector<std::thread> threads;
int threadCount = 0;

void WorkerThread(int id) {
    while (true) {
        // Ожидание сигнала на выполнение
        WaitForSingleObject(hEvent, INFINITE);
        // Выполнить работу потока
        // Например, просто вывод сообщения
        std::cout << "Поток " << id << " запущен." << std::endl;
```

```
        // Имитируем задержку работы потока
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
}

extern "C" __declspec(dllexport) void StartThreads(int count) {
    std::lock_guard<std::mutex> lock(mtx);
    for (int i = 0; i < count; ++i) {
        threads.emplace_back(WorkerThread, threadCount++);
    }
    // Уведомляем потоки, что они могут начинать работу
    SetEvent(hEvent);
}

extern "C" __declspec(dllexport) void StopThread() {
    std::lock_guard<std::mutex> lock(mtx);
    if (!threads.empty()) {
        // Остановка последнего потока
        threads.back().detach(); // Убедимся, что поток успешно завершен
        threads.pop_back();
    }
}

extern "C" __declspec(dllexport) void SendMessageToThread(const char* message, int
recipient) {
    std::lock_guard<std::mutex> lock(mtx);
    // Обработка конкретного потока или всех
    for (int i = 0; i < threads.size(); ++i) {
        if (recipient == 0 || recipient == i + 1) { // Если все потоки или
конкретный
            // Запись в файл с именем <свой номер>.txt
            std::ofstream outFile(std::to_string(i) + ".txt", std::ios::app);
            outFile << message << std::endl;
        }
    }
}

extern "C" __declspec(dllexport) void ShutDownThreads() {
    std::lock_guard<std::mutex> lock(mtx);
    for (auto& thread : threads) {
        if (thread.joinable()) {
            thread.join();
        }
    }
    threads.clear();
}

// Функция для инициализации
void Initialize() {
    hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
}
```

Описание кода

1. Диалоговое приложение:
   - Создается пользовательский интерфейс для управления рабочими потоками.
   - В функции DialogProc обрабатываются нажатия кнопок.
   - Загрузка и использование функций из DLL для управления потоками и отправки сообщений.
2. Консольное приложение:
   - Каждый рабочий поток выполняет функцию WorkerThread, которая имитирует выполнение работы.
   - Используются события и мьютексы для синхронизации.
   - Реализованы функции для обработки создания, остановки потоков и отправки сообщений.
3. DLL:

- Реализуется экспорт функций для взаимодействия между приложениями.
- DLL загружается автоматически и используется в диалоговом приложении.

*Приложение 2*
*к рабочей программе дисциплины (модуля)*
*Б1.В.ДВ.03.01 Промышленное программирование*

**Перечень литературы, необходимой для освоения дисциплины (модуля)**

Направление подготовки: 15.03.06 Мехатроника и робототехника
Профиль подготовки: Физические основы мехатроники и робототехники
Квалификация выпускника: бакалавр
Форма обучения: очная
Язык обучения: русский
Год начала обучения по образовательной программе: 2025

**Основная литература:**

1. Агафонов, Е. Д. Прикладное программирование : учебное пособие / Е. Д. Агафонов, Г. В. Ващенко. - Красноярск: СФУ, 2015. - 112 с. - ISBN 978-5-7638-3165-8. - Текст: электронный. - URL: https://znanium.com/catalog/product/550046 - Режим доступа: по подписке.

2. Медведев, М. А. Программирование на СИ#: Учебное пособие / Медведев М.А., Медведев А.Н., - 2-е изд., стер. - Москва : Флинта, Изд-во Урал. ун-та, 2017. - 64 с. ISBN 978-5-9765-3169-7. - Текст : электронный. - URL: https://znanium.com/catalog/product/948428 - Режим доступа: по подписке.

**Дополнительная литература:**

1. Калиногорский, Н. А. Основы практического применения интернет-технологий : учебное пособие / Н. А. Калиногорский. - 3-е изд., стер. - Москва : ФЛИНТА, 2020. - 182 с. - ISBN 978-5-9765-2302-9. - Текст : электронный. - URL: https://znanium.com/catalog/product/1142475 - Режим доступа: по подписке..

2. Царев, Р.Ю. Информатика и программирование [Электронный ресурс] : учеб. пособие / Р. Ю. Царев, А. Н. Пупков, В. В. Самарин, Е. В. Мыльникова. - Красноярск : Сиб. федер. ун-т, 2014. - 132 с. - ISBN 978-5-7638-3008-8. - Текст : электронный. - URL: https://znanium.com/catalog/product/506203 - Режим доступа: по подписке.

*Приложение 3*
*к рабочей программе дисциплины (модуля)*
*Б1.В.ДВ.03.01 Промышленное программирование*

**Перечень информационных технологий, используемых для освоения дисциплины (модуля), включая перечень программного обеспечения и информационных справочных систем**

Направление подготовки: 15.03.06 Мехатроника и робототехника

Профиль подготовки: Физические основы мехатроники и робототехники

Квалификация выпускника: бакалавр

Форма обучения: очная

Язык обучения: русский

Год начала обучения по образовательной программе: 2025

Освоение дисциплины (модуля) предполагает использование следующего программного обеспечения и информационно-справочных систем:

Программное обеспечение: операционная система Windows, Microsoft office, PyCharm, Kaspersky Free для Windows

Электронная библиотечная система «ZNANIUM.COM»

Электронная библиотечная система Издательства «Лань»

Электронная библиотечная система «Консультант студента»