Экзамен АиП C++

C++

1. Основные отношения между классами: наследование, композиция, наполнение, зависимость. Примеры диаграмм классов.

Наследование - создание производных классов на основе базового, класс-наследник получает свойства и методы базового класса, может переопределять и добавлять новые. На диаграмме классов обозначается незакрашенной стрелкой
Композиция - отношение между классами, когда один из них включает в себя объекты другого (1 или больше) и полностью управляет их жизненным циклом. Включаемый объект может существовать только как часть контейнера. Обозначается на диаграмме стрелочкой с закрашенным ромбиком. Пример: автомобиль и двигатель, без которого он не будет работать.
Наполнение(агрегация) - отношение между классами, при котором один из них содержит или не содержит объекты другого (0 или больше), в отличие от композиции объекты второго класса могут существовать без объекта первого. Пример: автомобиль и колёса, которые могут быть в количестве от 0 до 4.
Зависимость(ассоциация) - объекту одного класса ставится в соответствие некоторое количество объектов другого, на диаграмме обозначается простой стрелкой с указанием с каждой стороны количества объектов. В этом случае один класс содержит или использует объекты другого.

2. Конструкторы: инициализирующий, без параметров (инициализирующий и неинициализирующий), копирующий, перемещения. Примеры.

Конструкторы представляют специальную функцию, которая имеет то же имя, что и класс, которая не возвращает никакого значения и которая позволяет инициализировать объект класса во время его создания и таким образом гарантировать, что поля класса будут иметь определенные значения. При каждом создании нового объекта класса вызывается конструктор класса. Конструкторов может быть не сколько, каждый может иметь любые параметры и вызывать другие конструкторы.
По умолчанию компилятор при компиляции классов генерирует специальный конструктор - **конструктор копирования**, который позволяет создать объект на основе другого объекта (по сути копирует объект). Конструктор копирования по умолчанию копирует

значения полей объекта, в новый объект. Конструктор копирования должен принимать в качестве параметра объект того же класса. Причем параметр лучше принимать по ссылке, потому что при передаче по значению компилятор будет создавать копию объекта. А для создания копия объекта будет вызываться конструктор копирования, что приведет бесконечной рекурсии.

```
Person(const Person &p)
{
    name = p.name;
    age = p.age + 1;
}
// если нужно удалить копирующий конструктор
Person(const Person &p) = delete;
```

Конструктор перемещения (move constructor) представляет альтернативу конструктору копирования в тех ситуациях, когда надо сделать копию объекта, но копирование данных нежелательно - вместо копирования данных они просто перемещаются из одной копии объекта в другую.

```
class Message
{
public:
    Message(const char* data, unsigned count)
    {
        size = count;
        text = new char[size];  // выделяем память
        for(unsigned i = 0; i < size; i++)    // копируем данные
        {
                    text[i] = data[i];
                }
                id = ++counter;
                std::cout << "Create Message " << id << std::endl;
    }
    // конструктор копирования
    Message(const Message& copy) : Message{copy.getText(), copy.size }  //
обращаемся к стандартному конструктору
    {
            std::cout << "Copy  Message " << copy.id << " to " << id <<
std::endl;
        }
    ~Message() // деструктор
    {
        std::cout << "Delete Message " << id << std::endl;
        delete[] text;  // особождаем память
```

```
    }
    char* getText() const { return text; }
    unsigned getSize() const { return size; }
    unsigned getId() const {return id;}
private:
    char* text { };  // текст сообщения
    unsigned size { };    // размер сообщения
    unsigned id { };  // номер сообщения
    static inline unsigned counter{};   // статический счетчик для генерации
номера объекта
};
```

3. Список инициализации. Примеры.

Поля класса, являющиеся константами нельзя инициализировать в конструкторе, потому что они являются неизменяемыми, поэтому используются списки инициализации:

```
class Person
{
    const std::string name;
    unsigned age;
public:
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
    Person(std::string p_name, unsigned p_age) : name(p_name), age(p_age)
    { }
};
```

В этом случае поля `name` и `age` получат значения параметров `p_name` и `p_age` ещё до того, как начнёт выполняться тело конструктора `Person`. При использовании списков инициализации важно учитывать, что передача значений должна идти в том порядке, в котором константы и переменные объявлены в классе.

4. Наследование. Особенности описания конструкторов производных классов. Примеры.

Наследование - создание производных классов на основе базового, класс-наследник получает свойства и методы базового класса, может переопределять и добавлять новые. При наследовании конструкторы **не наследуются**, и если базовый класс содержит только конструкторы с параметрами, то производный класс **должен** вызывать в своем конструкторе один из конструкторов базового класса. После списка параметров

конструктора производного класса через двоеточие идет вызов конструктора базового класса, в который передаются значения параметров.

```
#include <iostream>
class Person
{
public:
    Person(std::string name, unsigned age);
    void print();
private:
    std::string name;
    unsigned age;
};
class Employee : public Person
{
public:
    Employee(std::string name, unsigned age, std::string company):
Person(name, age) // вызов конструктора базового класса
    {
        this->company = company;
    }
private:
    std::string company;
};
```

Если конструктор полностью совпадает с конструктором базового класса его можно просто подключить:

```
class Employee : public Person
{
public:
    using Person::Person;
};
```

5. Композиция. Особенности описания конструкторов классов-агрегатов. Примеры.

Композиция - отношение между классами, когда один из них включает в себя объекты другого (1 или больше) и полностью управляет их жизненным циклом. Включаемый объект может существовать только как часть контейнера. Обозначается на диаграмме стрелочкой с закрашенным ромбиком. Пример: автомобиль и двигатель, без которого он не будет работать. Для включаемых классов обязательным условием является существование конструктора по умолчанию.

```
class Point2D
{
private:
    int m_x;
    int m_y;
public:
    // Конструктор по умолчанию
    Point2D() : m_x(0), m_y(0) {}
    // Специфический конструктор
    Point2D(int x, int y) : m_x(x), m_y(y) {}
    void setPoint(int x, int y)
    {
        m_x = x;
        m_y = y;
    }
};
class Line
{
private:
    std::string name;
    Point2D a, b;
}
```

6. Наполнение. Особенности описания конструкторов и деструкторов классов-агрегатов. Примеры.

Наполнение(агрегация) - отношение между классами, при котором один из них содержит или не содержит объекты другого (0 или больше), в отличие от композиции объекты второго класса могут существовать без объекта первого. Пример: автомобиль и колёса, которые могут быть в количестве от 0 до 4.
Должен быть обязательно конструктор по умолчанию (без параметров).

```
class Wheel {
private:
        int id;
        static int n;
public:
        Wheel() {
                id = ++n;
        }
        int getID() { return id; }
};

int Wheel::n = 0;
```

```
class Car {
private:
        Wheel* wheels;
public:
        Car(Wheel* w) : wheels(w) {};
        ~Car() {
                std::cout << "Объект Car уничтожается\n";
        }
};

int main()
{
        Wheel* wheels = new Wheel[4]; // массив колёс
        {
                Car car(wheels); // автомобиль с колёсами
        } // объект car удаляется при выходе из блока
        for (int i = 0; i < 4; ++i)
                std::cout << wheels[i].getID() << "\n"; // колёса продолжают
существовать
        delete[] wheels;
}
```

7. Полиморфное наследование в языке C++. Раннее и позднее связывание: описание в программе и механизмы реализации. Примеры.

Полиморфизм в языке C++ позволяет работать с разными данными одинаковым образом. Несколько классов наследуют и переопределяют свойства одного общего, в результате чего получают общий интерфейс для работы с ними. В зависимости от способа переопределения происходит раннее или позднее связывание методов с объектами и типами, для которых они будут вызваны.
Раннее связывание - это когда обычный метод явно переопределяется для класса-наследника и вызывается для всех объектов соответствующего типа, в том числе по типу указателя на объект(если явно преобразовать тип указателя, для него будет вызван уже другой метод, независимо от того, объект какого типа реально находится по данному адресу).
Позднее связывание - это когда метод родительского класса объявляется виртуальным и классы-наследники его переопределяют, в этом случае для данной иерархии классов создаётся специальная таблица виртуальных методов и при каждом вызове метода для объекта по указателю происходит получение реального типа хранимого объекта из специального скрытого поля и вызов метода по таблице для него.

В данном примере для метода `PrintInfo()` используется раннее связывание, а для метода `Sound` - позднее связывание:

```
class Animal {
public:
        virtual void Sound()=0; // чистый виртуальный метод, требуется
переопределение в потомках
        void PrintInfo() { // обычный метод
                std::cout << "Class Animal\n";
        };
};

class Dog : public Animal {
public:
        void Sound() override {
                std::cout  << "Ruff-ruff\n";
        }
        void PrintInfo() {
                std::cout << "Class Dog\n";
        }
};

class Cat : public Animal {
public:
        void Sound() override {
                std::cout << "Meow-meow\n";
        }
        void PrintInfo() {
                std::cout << "Class Cat\n";
        }
};

int main() {
        Animal *animals[2];
        animals[0] = new Cat();
        animals[1] = new Dog();
        for (auto i : animals) {
                i->Sound();     // для каждого объекта вызовется свой метод
                i->PrintInfo(); // для всех объектов вызовется метода класса
Animal
        }
        // при позднем связывании для любого преобразования типа указателя
        // всё равно вызовется метода, соответствующий реальному типу объекта
        ((Dog*)animals[0])->Sound();
        ((Dog*)animals[1])->Sound();
        // при раннем связывании и явном преобразовании вызовется метод типа
```

```
указателя
        ((Cat*)animals[0])->PrintInfo();
        ((Dog*)animals[1])->PrintInfo();
}
```

8. Контейнерные классы. Примеры диаграмм классов.

Для управления наборами объектов в стандартной библиотеке C++ определены контейнеры. Контейнер представляет коллекцию объектов определенного типа и позволяет управлять доступом к этим элементам. В C++ есть два типа контейнеров: ассоциативные и последовательные:

Последовательные

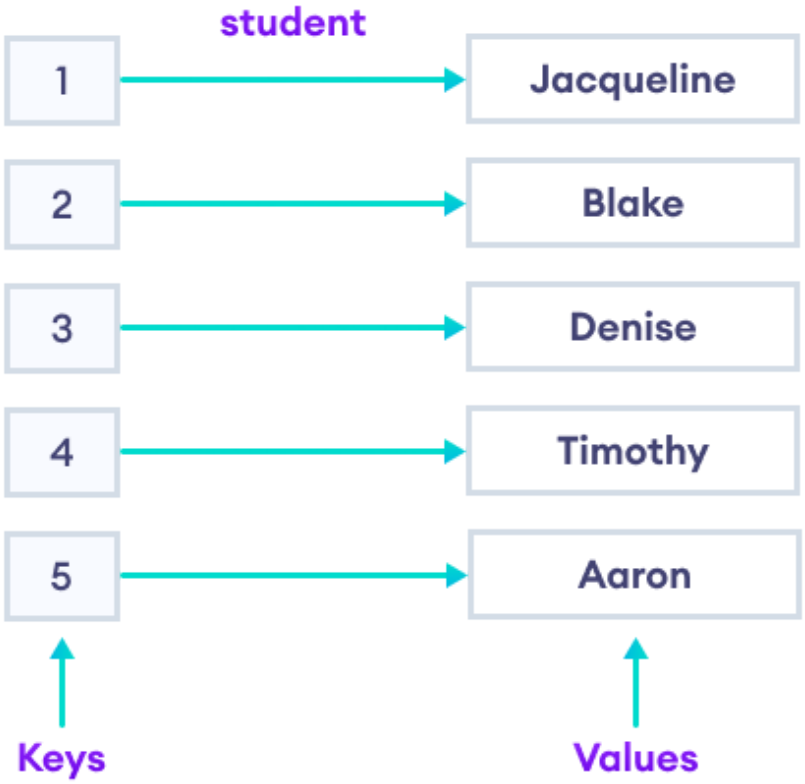
Обеспечивают хранение конечного количества однотипных величин в виде непрерывной последовательности

- `vector` - динамический массив - структура, эффективно реализующая произвольный доступ к элементам, добавление в конец и удаление из конца
- `list` - линейный список - эффективно реализует вставку и удаление элементов в/из произвольного места и не эффективно - произвольный доступ к элементам
- `stack` - стек - эффективно реализует добавление в конец и удаление из конца
- `queue` - очередь - эффективно реализует добавление в конец и удаление из начала
- `deque` - двусторонняя очередь (дек) - эффективно реализует произвольный доступ к элементам, добавление в оба конца и удаление из обоих концов
- `priority_queue` – очередь, сортированная по приоритетам

Ассоциативные

Обеспечивают быстрый доступ к данным по ключу, построены на основе сбалансированных (ветки $\pm$ равны по длине) деревьев

- `map` - словарь с уникальными ключами (ищет по дереву, выбирая нужное поддерево)



- `multimap` - словарь с дубликатами ключей (по одному ключу несколько значений)
- `set` - множество (множество с заданным отношением порядка)
- `multiset` - мультимножество (множество, в котором могут повторяться элементы, поддерживает элементы отсортированными)
- `bitset` - битовое множество (набор битов)

9. Qt. Принципы создания графических интерфейсов. Использование C++ контейнеров.

Qt - это фреймворк и набор инструментов для разработки приложений с открытым исходным кодом, позволяющий создавать приложения с графическим интерфейсом (GUI) для различных платформ. В нём используется язык C++, но существуют привязки и к другим языкам.
Qt использует подход Model-View-Controller (MVC), который разделяет приложение на три части:

- **Модель:** Хранит данные приложения.
- **Вид:** Отображает данные модели пользователю.
- **Контроллер:** Обрабатывает взаимодействие пользователя с видом и обновляет модель соответственно.

Принципы создания графических интерфейсов.

При разработке графического интерфейса стоит учитывать, что работать с программой будет работать пользователь, ещё не знакомый с её устройством и функциями. Поэтому интерфейс не должен быть перегруженным большим количеством элементов, главные функции должны быть на виду, остальные расположены в привычных для пользователя и логичных местах. Например, функции для работы с файлами следует разместить в меню "Файл", для настройки внешнего вида приложения - в меню "Вид" или "Окно", поле поиска поместить сверху, как это сделано в большинстве программ, и т.д. Также стоит адаптировать внешний вид под платформу, на которой запускается приложение (с этим помогает фреймворк Qt). При проектирование интерфейса главного окна можно разделить его на области, каждая из которых будет отвечать за что-то одно, например, блок с кнопками отделить от таблицы с данными. Обо всех действиях, происходящих в программе стоит предупреждать - выводить индикатор загрузки, сообщать о прогрессе выполнения какого-либо действия, предупреждать об ошибках и спрашивать подтверждение важных действий.
То есть интерфейс должен быть таким, чтобы пользователь мог его быстро освоить, в случае сложного интерфейса стоит составить дополнительную документацию по работе с программой.
!!! ВСЕГДА НУЖНА КНОПКА ВЫХОДА !!!

C++ контейнеры

Qt предоставляет набор шаблонизированных классов контейнеров для хранения и управления коллекциями элементов. Эти контейнеры являются аналогами стандартных контейнеров C++, но оптимизированы для работы в Qt и имеют ряд дополнительных функций.
Основные контейнеры Qt:

- **QList:** Динамический список, который может хранить элементы любого типа.
- **QVector:** Быстрый динамический массив, оптимизированный для произвольного доступа к элементам.
- **QSet:** Множество, которое хранит уникальные элементы без дубликатов.
- **QMap:** Ассоциативный массив, который сопоставляет ключи со значениями.
- **QHash:** Хэш-таблица, которая обеспечивает быстрый поиск элементов по ключу.
- **QStack:** Стек LIFO (Last In, First Out), который используется для хранения элементов в порядке их добавления.
- **QQueue:** Очередь FIFO (First In, First Out), которая используется для хранения элементов в порядке их добавления.

10. Qt. Сигналы и слоты.

Механизм сигналов и слотов представляет одну из отличительных особенностей Qt и позволяют сделать приложение отзывчивым, реагировать на действия пользователя, отслеживать различные события в приложении. Так, когда пользователь выполняет какое-либо действие с каким-либо элементом пользовательского интерфейса, должна быть выполнена определённая задача. Например, если пользователь нажимает кнопку "Закрыть" в верхнем правом углу окна, то ожидается, что окно закроется. То есть необходим механизм для отслеживания событий и реагирования на них. В среде Qt такой механизм предоставляют сигналы и слоты.

- **Сигнал** - это сообщение, которое передается, чтобы сообщить об изменении состояния объекта. Сигнал может нести информацию о произошедшем изменении.
- **Слот** - это специальная функция, вызываемая в ответ на определенный сигнал. Поскольку слоты - это функции, они содержат логику для выполнения определенного действия.

```
class MyClass : public QObject
{
  Q_OBJECT
public:
  explicit MyClass(QObject *parent = nullptr);
signals:
  void signalName();  // определяем сигнал
public slots:
  void onButtonClicked() { ... }  // определяем слот
};

MyClass::MyClass(QObject *parent = nullptr) : QObject(parent) {
  // соединение сигнала и слота
  QObject::connect(button, SIGNAL(clicked()), this, SLOT(onButtonClicked()));
}

void MyClass::onButtonClicked() {
  // совершение действия
  emit signalName(); // отправка сигнала
}
```

11. Qt. События и обработчики событий.

Сигналы генерируются, когда происходят какие-либо **события**, и всё, что происходит в приложении, является результатом обработки сигналов от тех или иных событий. Таким событием может быть клик мыши, нажатие клавиши, изменение размера окна, сигнал таймера и т.д.
В модели событий есть 3 участника:

1. **Источник события** - это объект, состояние которого изменяется
2. **Объект события** - это отслеживаемый параметр источника события (например, нажатие клавиши на клавиатуре или изменение размеров виджета)
3. **Цель события** - это объект, который должен быть уведомлён о произошедшем событии

В отличие от сигналов, которые необходимы для организации взаимодействия между виджетами, события необходимы для организации взаимодействия между виджетом и системой.
Пример обработки нажатия клавиши:

```
#include <QWidget>
#include <QApplication>
#include <QKeyEvent>

class Window : public QWidget {

public:
        explicit Window(QWidget* parent = 0) : QWidget(parent) { }
        // обработчик события нажатия клавиши
        void keyPressEvent(QKeyEvent* event) {
                if (event->key() == Qt::Key_Escape) {
                        qApp->quit(); // закрываем приложение
                }
        }
};
```

Пример обработки сигнала таймера:

```
#include <QWindet>

class MainWindow : public QWidget
{
        Q_OBJECT
public:
        explicit MainWindow(QWidget* parent = 0) : QWidget(parent)
        {
                timerId = startTimer(1000); // запускаем таймер с интервалом
1000 мс
        }
        virtual ~MainWindow();

private:
        int timerId;
```

```
protected:
        void timerEvent(QTimerEvent* event);
};

void MainWindow::timerEvent(QTimerEvent* event)
{
        // обрабатываем срабатывание таймера
}

MainWindow::~MainWindow()
{
        killTimer(timerId); // останавливаем таймер в конце
}
```

12. Полиморфное наследование. Сложный полиморфизм. Пример.

Полиморфизм в языке C++ позволяет работать с разными данными одинаковым образом. Несколько классов наследуют и переопределяют свойства одного общего, в результате чего получают общий интерфейс для работы с ними. В зависимости от способа переопределения происходит раннее или позднее связывание методов с объектами и типами, для которых они будут вызваны. Сложный полиморфизм реализуется через механизм позднего связывания и требует описания виртуальных методов. Виртуальными называются методы, которые объявляются с использованием ключевого слова virtual в базовом классе и переопределяются (замещаются) в одном или нескольких производных классах.

```
class Animal {
public:
        virtual void Sound()=0; // чистая виртуальная функция, требуется
переопределение в потомках
};

class Dog : public Animal {
public:
        void Sound() override {
                std::cout  << "Ruff-ruff\n";
        }
};

class Cat : public Animal {
public:
        void Sound() override {
```

```
                std::cout << "Meow-meow\n";
        }
};

int main() {
        Animal *animals[2];
        animals[0] = new Cat();
        animals[1] = new Dog();
        for (auto i : animals)
                i->Sound();     // для каждого объекта вызовется свой метод
}
```

13. C++. Множественное наследование. Виртуальное наследование. Пример.

Множественным наследованием называется наследование одновременно от нескольких классов, производный класс включает несколько базовых классов. При этом необходимо вызывать конструкторы базовых классов с параметрами (при отсутствии конструктора без параметров).

```
#include <iostream>
class Book {   // класс книги
public:
        Book(unsigned pages) : pages(pages) {}
        void printPageCount() {
                std::cout << pages << " pages" << std::endl;
        }
private:
        unsigned pages; // количество страниц
};

class File {   // класс электронного файла
public:
        File(double size) : size(size) {
                std::cout << "File created" << std::endl;
        }
        ~File() {
                std::cout << "File deleted" << std::endl;
        }
        void printSize() {
                std::cout << size << "Mb" << std::endl;
        }
private:
        double size;  // размер файла
};
```

```
// класс электронной книги
class Ebook : public Book, public File
{
public:
        // нужно вызвать конструкторы базовых классов
        Ebook(std::string title, unsigned pages, double size) :
                Book(pages), File(size), title(title) { }
        void printTitle() {
                std::cout << "Title: " << title << std::endl;
        }
private:
        std::string title;  // название книги
};

int main() {
        Ebook cppbook{ "About C++", 320, 5.6 };
        cppbook.printTitle();
        cppbook.printPageCount();
        cppbook.printSize();
}
```

Если бы базовые классы имели методы с одинаковыми названиями, то произошла бы ошибка компиляции. В таком случае необходимо явно указывать, к методу какого из базовых классов нужно обратиться. Например, если бы в предыдущем примере, классы `Book` и `File` имели метод с названием `print()`, то при обращении к ним требовалось бы явно указать, какой именно вызвать:

```
int main() {
        ...
        cppbook.Book::print(); // вызов метода класса Book
        cppbook.File::print(); // вызов метода класса File
}
```

Виртуальное наследование требуется, когда класс наследуется от нескольких классов, которые в свою очередь напрямую или косвенно наследуются от одного базового класса. Например:

```
class Person {
public:
        Person(std::string name) : name(name) {
                std::cout << "Person created" << std::endl;
        }
        ~Person() {
                std::cout << "Person deleted" << std::endl;
```

```
        }
        void print() const {
                std::cout << "Person " << name << std::endl;
        }
private:
        std::string name;
};

class Student : public virtual Person {
public:
        Student(std::string name) : Person{ name } {}
};

class Employee : public virtual Person {
public:
        Employee(std::string name) : Person{ name } {}
};
// работающий студент
class StudentEmployee : public Student, public Employee {
public:
        StudentEmployee(std::string name) : Person{ name }, Student{ name },
Employee{ name } {}
};

int main() {
        StudentEmployee bob{ "Bob" };
        bob.print();
}
```

Здесь классы `Student` и `Employee` виртуально наследуются от класса `Person`, чтобы их общий наследник `StudentEmployee` мог вызывать функцию `print()`, реализованную в классе `Person`. Если не использовать виртуальное наследование, возникнет ошибка компиляции.

14. C++. Статические поля и методы. Пример.

Статические поля и методы применяют для данных и методов, которые должны относиться не к конкретному объекту класса, а ко всему классу в целом. Для их определения используется ключевое слово `static`. Обращение к таким полям и методам происходит через название класса с помощью двойного двоеточия, например: `Counter::count`.
Статические переменные определяются всего один раз и существуют, даже если не было создано ни одного объекта класса. Типичный пример использования статических полей счётчик количества созданных объектов данного класса:

```
class Counter {
private:
        static int count;
public:
        Counter() {
                ++count;
        }
        void print_count() {
                std::cout << count << std::endl;
        }
        ~Counter() {
                --count;
        }
};

int Counter::count = 0;

int main() {
        Counter *a = new Counter();
        a->print_count();
        Counter b;
        b.print_count();
        delete a;
        b.print_count();
}
```

В этом примере значение переменной `count` общее для объектов `a` и `b`, и, когда объект `a` уничтожается и в его деструкторе значение `count` уменьшается, для объекта `b` это значение также уменьшается.
Статические функции также принадлежат классу и не привязаны к конкретному объекту. Статическими принято делать функции, которые не используют и не изменяют данные конкретного объекта. Можно изменить предыдущий пример, сделав метод для вывода текущего значения счётчика статическим, чтобы он не был привязан к объекту:

```
class Counter {
private:
        static int count;
public:
        Counter() {
                ++count;
        }
        static void print_count() {
                std::cout << count << std::endl;
        }
```

```
        ~Counter() {
                --count;
        }
};

int Counter::count = 0;

int main() {
        Counter::print_count(); // теперь можно вывести count даже до создания
первого объекта
        Counter *a = new Counter();
        Counter::print_count();
        Counter b;
        Counter::print_count();
        delete a;
        Counter::print_count();
}
```

15. C++. Переопределение операций. Пример.

Перегрузка операторов (operator overloading) позволяет определить для объектов классов встроенные операторы, такие как +, -, * и т.д. Для определения оператора для объектов своего класса, необходимо определить функцию, название которой содержит слово operator и символ перегружаемого оператора. Перегрузить можно только те операторы, которые уже определены в C++. Создать новые операторы нельзя. Также нельзя изменить количество операндов, их ассоциативность, приоритет. Переопределены операторы могут быть как внутри класса в виде метода (количество параметров на один меньше количества операндов, так как есть доступ к объекту, для которого оператор был вызван), так и в виде отдельной функции вне класса (количество параметров равно количеству операндов). Операторы с двумя параметрами называются бинарными, с одним - унарными.

```
#include <iostream>

class Container {
private:
        int val;
public:
        Container() : val(0) {};
        Container(int v) : val(v) {};
        int get_val() const {
                return val;
        }
        // переопределение оператора сложения внутри класса
```

```
        Container operator+(const Container& right) {
                return Container(val + right.get_val());
        }
        //переопределение унарного оператора внутри класса (префиксный
инкремент, для постфиксного указать "operator++(int)")
        Container& operator++() {
                ++val;
                return *this;
        }
};

// переопределение оператора умножения вне класса
Container operator*(const Container& left, const Container& right) {
        return Container(left.get_val() * right.get_val());
}
// переопределение унарного оператора вне класса
Container& operator--(Container& obj) {
        // в данном случае не получится реализовать оператор, не создавая
нового метода в классе и не изменяя доступность полей
        // решить эту проблему можно с помощью дружественных классов (о них
следующий вопрос)
        // obj.val--;
        return obj;
}

// переопределение оператора вывода
std::ostream& operator<<(std::ostream& stream, const Container& obj) {
        stream << obj.get_val();
        return stream;
}

int main() {
        Container a(2), b(5);
        std::cout << "Результат сложения: "  << a + b << std::endl;
        std::cout << "Результат умножения: " << a * b << std::endl;
        ++a;
        // a++; - сейчас так сделать нельзя, так как не определён этот
оператор
        std::cout << "Результат инкремента: " << a << std::endl;
}
```

16. C++. Дружественные функции, методы и классы. Пример.

Дружественные функции и классы могут получать доступ к скрытым полям и методам класса, при этом не являясь членами класса. Для их определения используется ключевое

слово `friend`. Модифицируем программу из предыдущего вопроса, переопределение операции декремента вне класса реализовано в виде дружественной функции и теперь может изменять значение `private` поля класса `Container`, хотя не является его членом:

```
class Container {
private:
        int val;
public:
        Container() : val(0) {};
        Container(int v) : val(v) {};
        int get_val() const {
                return val;
        }
        friend Container& operator--(Container& obj);
};

// переопределение унарного оператора вне класса
Container& operator--(Container& obj) {
        obj.val--; // в дружественой функции можем получить доступ к скрытым
полям класса
        return obj;
}

int main() {
        Container a(2);
        --a;
        std::cout << "Результат декремента: " << a.get_val() << std::endl;
}
```

Дружественные функции могут также быть методами другого класса, тогда их определение происходит в виде `friend <тип> <имя класса>::<имя метода>(...);` Дружественными могут быть не только функции, но и целые классы, тогда все методы дружественного класса получат доступ к скрытым полям данного класса:

```
#include <iostream>

class Container {
private:
        int val;
public:
        Container() : val(0) {};
        Container(int v) : val(v) {};
        int get_val() const {
                return val;
```

```
        }
        // объявление класса дружественным
        friend class Controller;
};

class Controller {
public:
        // метод дружественного класса получает доступ к скрытым полям
        static void increase(Container& obj) {
                obj.val++;
        }
};

int main() {
        Container a(2);
        std::cout << a.get_val() << std::endl;
        Controller::increase(a);
        std::cout << a.get_val() << std::endl;
}
```

17. C++. Конструкторы перемещения и операция перемещения. Пример.

Конструктор перемещения (move constructor) представляет альтернативу конструктору копирования в тех ситуациях, когда надо сделать копию объекта, но копирование данных нежелательно - вместо копирования данных они просто перемещаются из одной копии объекта в другую. Использование конструкторов и операторов перемещения может значительно ускорить работу с множество объектов, так как сокращает количество операций по работе с памятью.

```
#include <iostream>

class Container {
private:
        int* array;
        int len;
public:
        Container() : array(nullptr), len(0) {}
        Container(int* arr, int n) : len(n){
                // выделяем память при создании объекта
                array = new int[n];
                for (int i = 0; i < n; ++i)
                        array[i] = arr[i];
        }
```

```
        // конструктор перемещения использует rvalue-ссылку в качестве
параметра
        Container(Container&& source) {
                array = source.array;
                len = source.len;
                source.array = nullptr;
                source.len = 0;
        }
        // оператор присваивания перемещением
        Container& operator=(Container&& source) {
                // проверка на присваивание себе
                if (&source == this)
                        return *this;
                // очистка текущих значений
                delete array;
                // переносим значения и передаём право собственности
                array = source.array;
                len = source.len;
                source.array = nullptr;
                source.len = 0;
                return *this;
        }
        void print() {
                for (int i = 0; i < len; ++i)
                        std::cout << array[i] << " ";
                std::cout << std::endl;
        }
};

int main() {
        int arr[] = { 1, 2, 3 };
        Container a(arr, 3);
        // для вызова конструктора перемещения нужно сначала преобразовать
объект в rvalue-ссылку
        // для этого можно использовать стандартную функцию std::move()
        Container b(std::move(a));
        a.print();
        b.print();
        Container c;
        // аналогично для оператора перемещения
        c = std::move(b);
        a.print();
        b.print();
        c.print();
}
```

18. C++. Правило пяти. Пример.

Правило пяти - это расширение правила трёх для современного C++, учитывающее семантику перемещения. Это правило гласит, что если в классе явно определяется один из следующих специальных методов класса, то скорее всего нужно определить и остальные четыре:

1. Деструктор
2. Конструктор копирования
3. Оператор присваивания копирования
4. Конструктор перемещения
5. Оператор присваивания перемещения

Это правило решает проблемы безопасности и производительности при работе с динамической памятью и помогает избежать утечек памяти.

```
class Container {
private:
        int* values;
        int len;
public:
        // инициализирующий конструктор с параметрами
        Container(int* arr, int n) : len(n) {
                values = new int[n];
                for (int i = 0; i < n; ++i)
                        values[i] = arr[i];
        }
        // конструктор копирования
        Container(const Container& other) : Container(other.values, other.len)
{};
        // констуктор перемещения
        Container(Container&& other) {
                len = other.len;
                other.len = 0;
                values = other.values;
                other.values = nullptr;
        }
        // оператор присваивания копирования
        Container& operator=(const Container& other) {
                if (&other == this)
                        return *this;
                delete values;
                len = other.len;
                values = new int[len];
```

```
                for (int i = 0; i < len; ++i)
                        values[i] = other.values[i];
                return *this;
        }
        // оператор присваивания перемещения
        Container& operator=(Container&& other) {
                len = other.len;
                other.len = 0;
                values = other.values;
                other.values = nullptr;
                return *this;
        }
        // деструктор
        ~Container() {
                delete[] values;
        }
};
```

19. C++. Шаблоны функций. Пример.

Механизм шаблонов в языке C++ позволяет создавать универсальные функции, не зависящие от конкретного типа. Это сокращает объём кода и избавляет программиста от необходимости реализовывать одну и ту же подпрограмму для работы с разными типами. Чтобы сделать функцию шаблоном, перед её определением нужно указать `template<typename T1, typename T2>` (вариант для 2 переменных `T1` и `T2` в шаблоне, ключевое слово `typename` также можно заменить на ключевое слово `class`), можно перечислить любое количество типов в треугольных скобках, далее эти имена будут использоваться внутри шаблона функции как типы. На этапе компиляции будут найдены все вызовы шаблонной функции и для каждого сгенерирована функция с соответствующими вызову типами.
Рассмотрим шаблоны на примере функции `max`, которая возвращает максимальный из её аргументов. Без использования шаблонов, реализация могла бы выглядеть следующим образом:

```
int max(int a, int b) {
        return (a > b) ? a : b;
}
long long int max(long long int a, long long int b) {
        return (a > b) ? a : b;
}
float max(float a, float b) {
        return (a > b) ? a : b;
}
double max(double a, double b) {
```

```
        return (a > b) ? a : b;
}
```

Приходится создавать несколько перегрузок функции `max`, чтобы учесть работу с разными типами чисел, но при этом реализация тела функций остаётся неизменной. Используя шаблоны, можно получить тот же результат:

```
#include <iostream>

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

int main() {
        std::cout << max<long long int>(1, 2) << std::endl;
        std::cout << max(2.f, 4.f) << std::endl;
        std::cout << max(122316234567890, 425316234567890) << std::endl;
}
```

В данном случае компилятор сгенерирует 2 версии функции `max`: для работы с типами `long long int` и `float`. Из первого вызова видно, что можно явно указать при вызове шаблонной функции, какой тип использовать: для `max<long long int>(a, b)` будет создана соответствующая функция, а её аргументы неявно приведены к этому типу, если это возможно.
Кроме того, шаблонизацию можно совмещать с перегрузкой функций, а также явно специализировать шаблон, например:

```
// старый вариант для двух аргументов
template<typename T>
T max(T a, T b);
// перегрузка для работы с массивом чисел
template <typename T>
T max(T* array, size_t n);
// явная специализация шаблона для типа int
template<>
int max(int a, int b);
```

20. C++. Шаблоны классов. Пример.

Шаблоны классов, точно так же, как и шаблоны функций, позволяют создавать универсальные классы, используя в них переменные, тип которых будет заранее неизвестен. Это может быть особенно полезно, например, при создании универсального

контейнера, который сможет хранить объекты любого типа. Объявление шаблонов классов аналогично объявлению шаблонов функций, рассмотрим пример класса для хранения пар объектов разных типов:

```
// шаблон с 2 параметрами
// в параметрах также указаны типы по умолчанию (T1=int, T2=int)
// typename можно заменить на class
// template<class T1=int, class T2=int> - работает точно так же
template<typename T1=int, typename T2=int>
class Container {
private:
        std::pair<T1, T2> *values;
        size_t size;
public:
        Container() : values(nullptr), size(0) {}
        Container(std::pair<T1, T2>* vals, size_t n);
        std::pair<T1, T2>& operator[](size_t i) {
                return values[i];
        }
        ~Container() {
                delete[] values;
        }
};

template<typename T1, typename T2>
Container<T1, T2>::Container(std::pair<T1, T2>* vals, size_t n) : size(n) {
        values = new std::pair<T1, T2>[n];
        for (int i = 0; i < n; ++i)
                values[i] = vals[i];
}
```

В зависимости от того, с какими именно наборами параметров будет вызван шаблон далее в программе, компилятор сгенерирует отдельные классы для каждого из них. Это необходимо учитывать при создании программ, содержащих множество больших классов, так как размер скомпилированного файла может многократно возрасти при генерации нескольких копий классов по шаблонам с разными типами.
Определение метода вне шаблона класса также имеет свои особенности - каждый метод должен быть отдельным шаблоном с указанием того, к какому именно наборов типов она относится. В данном выше примере это функция-конструктор.
Тип шаблонов может быть выведен неявно - компилятор сам определит его в зависимости от типов переданных параметров - и указан явно - в треугольных скобках при создании объекта: `Container<int, std::string>`
Для шаблонов классов возможна также полная или частичная специализация:

```
// полная специализация
template<>
class Container<int, char> {
        // реализация для набора типов int и char
}
// частичная специализация
template<typename T>
class Container<int, T> {
        // реализация нового получившегося шаблона с использованием ещё
неизвестного типа T
};
```

21. C++. Организация контейнеров на шаблонах. Пример диаграммы классов.

Механизм шаблонов удобно использовать для реализации универсальных контейнеров для работы с данными разных типов. В том числе все стандартные контейнеры C++ реализованы в виде шаблонов классов: `vector`, `set`, `map`, `stack`, `queue`, `deque` и т.д.
По определению контейнер - это структура данных, которая осуществляет работу с памятью и организацию данных, предлагая при этом удобный интерфейс для работы с ними и реализуя некоторую модель поведения.
Рассмотрим пример реализации стека с помощью шаблонов:

```
#include <iostream>

// структура для реализации элемента односвязного списка
template<typename T>
struct Element {
        Element<T>* prev;
        T value;
        Element(T val, Element<T>* p=nullptr) {
                prev = p;
                value = val;
        }
};

template<typename T>
class Stack {
private:
        Element<T>* first, * last;
        size_t size;
public:
        Stack() : first(nullptr), last(nullptr), size(0) {};
        // выделение памяти под новый элемент и добавление элемента в стек
```

```
        void push(T value) {
                if (first == nullptr) {
                        first = last = new Element<T>(value);
                } else {
                        last = new Element<T>(value, last);
                }
                size++;
        }
        // взять верхний элемент со стека и очистить его память
        T pop() {
                if (size == 0)
                        return 0; // здесь сделать исключение
                T value = last->value;
                Element<T>* t = last;
                if (size > 1)
                        last = last->prev;
                delete t;
                size--;
                return value;
        }
        size_t get_size() {
                return size;
        }
        // очистка памяти
        ~Stack() {
                while (last != nullptr) {
                        Element<T>* t = last;
                        last = last->prev;
                        delete t;
                }
        }
};

int main() {
        Stack<int> s;
        s.push(1);
        s.push(2);
        std::cout << "Size = " << s.get_size() << std::endl;
        std::cout << s.pop() << std::endl;
        std::cout << s.pop() << std::endl;
        std::cout << "Size = "  << s.get_size() << std::endl;
}
```

Вся работа с памятью производится внутри класса `Stack` - выделение памяти при добавлении элемента и очистка памяти при получении последнего элемента или удалении объекта стека. Пользователю этого класса при этом не нужно задумываться, как

и где выделяется и очищается память. Механизм шаблонов в этом случае обеспечивает возможность создавать стеки, которые хранят объекты произвольного класса.

22. C++. Шаблоны стандартных классов. Пример.

Стандартная библиотека шаблонов (**STL**, Standard Template Library) C++ содержит набор обобщённых алгоритмов, контейнеров и различных вспомогательных функций. Все контейнеры из STL являются шаблонами, что обеспечивает их работу с разными типами данных. То есть `vector`, `queue`, `set`, `stack`, `map` и т.д. - шаблоны классов соответствующих контейнеров.

```
#include <iostream>
// для использование этих классов необходимо подключить файлы с ними
#include <vector>
#include <set>
#include <map>

int main() {
        // vector целых чисел
        std::vector<int> int_vec;
        for (int i = 0; i < 10; ++i)
                int_vec.push_back(10 - i);
        std::cout << "Массив целых чисел: ";
        for (int i = 0; i < 10; ++i)
                std::cout << int_vec[i] << " ";
        std::cout << std::endl;
        // vector чисел типа double
        std::vector<double> double_vec;
        for (int i = 0; i < 10; ++i)
                double_vec.push_back((double)i/(i + 1));
        std::cout << "Массив чисел с плавающей точкой: ";
        for (int i = 0; i < 10; ++i)
                std::cout << double_vec[i] << " ";
        std::cout << std::endl;
        // set символов
        std::set<char> char_set;
        char symbols[7] = { 'a', 'a', 'a', 'b', 'b', 'c', 'd' };
        for (int i = 0; i < 7; ++i)
                char_set.insert(symbols[i]);
        std::cout << "Множество символов размера " << char_set.size() << ": ";
        // обход элементов множества с помощью итераторов
        for (std::set<char>::iterator iter = char_set.begin(); iter !=
char_set.end(); ++iter)
                std::cout << *iter << " ";
        std::cout << std::endl;
```

```
        // map с типом ключа std::string и типом элементов long long int
        std::map<std::string, long long int> m;
        m["abc"] = 123;
        m["def"] = 456;
        m["qwe"] = 789;
        std::cout << "Содержимое map: ";
        // обход пар ключ-элемент с помощью итераторов
        for (auto iter = m.begin(); iter != m.end(); ++iter)
                std::cout << "(" << iter->first << ", " << iter->second << ")
";
        std::cout << std::endl;
}
```

23. C++. Исключения C++. Пример.

Возникающие в процессе работы программы ошибки называют исключениями. Например, они возникают при вводе неправильных данных или при нарушении выполнения какого-либо процесса. Исключение представляет собой временный объект, который используется для передачи данных об ошибке. На основании этой информации необходимо обработать эту ошибку, иначе программа прекращает свою работу. Рассмотрим для примера программу, которая целочисленно делит одно введённое число на другое:

```
#include <iostream>

int divide(int a, int b) {
        return a / b;
}

int main() {
        int a, b;
        std::cout << "Введите делимое: ";
        std::cin >> a;
        std::cout << "Введите делитель: ";
        std::cin >> b;
        int res = divide(a, b);
        std::cout << "Результат деления: "  << res << std::endl;
}
```

При вводе в качестве делителя числа 0 программа завершится с сообщением об ошибке. Если добавить проверку на равенство `b` нулю, то функции всё равно придётся возвращать какое-то значение в случае невозможности вычислить значение выражения `a / b`. Обработка исключений позволяет решить эту проблему, сообщив программе, что

внутри функции произошла ошибка, чтобы её можно было обработать.
Работа с исключениями происходит с помощью ключевого слова `throw` и блоков для обработки ошибок `try ... catch ...`
Добавим в программу собственное исключение и его обработку:

```
#include <iostream>
#include <exception>
// объявляем свой класс ошибки, наследуем его от std::exception
class DivisionByZero : public std::exception {};

int divide(int a, int b) {
        if (b == 0)
                throw DivisionByZero(); // выбрасываем исключение, если
делитель равен 0
        return a / b;
}

int main() {
        int a, b;
        std::cout << "Введите делимое: ";
        std::cin >> a;
        std::cout << "Введите делитель: ";
        std::cin >> b;
        try {
                int res = divide(a, b);
                std::cout << "Результат деления: " << res << std::endl;
        } // обрабатываем ошибку деления на 0 в случае её возникновения
        catch (DivisionByZero) {
                std::cout << "Деление на ноль!" << std::endl;
        }
}
```

Теперь если пользователь введёт 0 в качестве делителя, функция `divide` выбросит исключение, которое будет обработано в функции `main` и программа не прервёт своё выполнение, а предупредит пользователя об ошибке в введённых данных. Добиться такого поведения можно было и предварительной валидацией данных, но для более сложных функций обойтись без механизма исключений довольно сложно.
В качестве выбрасываемого исключения может выступать любой объект:

```
// может быть вброшена строка
throw std::string("Деление на 0");
// или код ошибки
```

```
int error_code = 404;
throw error_code;
```

Функция может возвращать несколько разных ошибок. Чтобы обработать такие исключение, нужно указать соответствующие обработчики в блоке `try ... catch ...`:

```
try { ... }
catch (int error_code) { // ошибка типа int
        std::cout << "Произошла ошибка с кодом: " << error_code << std::endl;
}
catch (std;:string str) { // ошибка типа std::string
        std::cout << "Ошибка: " << str << std::endl;
}
catch (...) { // все остальные ошибки
        std::cout << "Неизвестная ошибка" << std::endl;
}
```

24. C++. Умные указатели. Особенности использования. Пример.

Использование обычных указателей C++ может приводить к утечкам памяти по разным причинам - по забывчивости программиста, из-за досрочного выхода из функции, из-за выброса исключения и т.д. Так происходит из-за того, что встроенные указатели не имеют механизма очистки памяти при уничтожении самого объекта указателя.
Один из способов решить эту проблему - создать некоторую "обёртку" для обычных указателей, которая взяла бы на себя очистку памяти, и для этого отлично подходит механизм деструкторов классов, которые вызываются всякий раз при уничтожении объекта класса. Это и легло в основу умных указателей, которые создаются как локальные переменные, которые уничтожаются при выходе из области видимости и очищают память самостоятельно. Реализуем простейший вариант такого указателя:

```
#include <iostream>
// шаблон каласса умного указателя
template<typename T>
class AutoPtr {
private:
        T* ptr; // указатель на хранимый объект
public:
        AutoPtr(T* p = nullptr) : ptr(p) {};
        // реализуем доступ как к обычному указателю
        T& operator*() const {
                return *ptr;
        }
        T* operator->() const {
```

```
                return ptr;
        }
        ~AutoPtr() {
                delete ptr; // очистка памяти в деструкторе
        }
};
// класс для тестирования умного указателя
class Object {
public:
        Object() {
                std::cout << "Объект создан" << std::endl;
        }
        ~Object() {
                std::cout << "Объект удалён" << std::endl;
        }
};
int main() {
        // выделение памяти и сохранение указателя
        AutoPtr<Object> ptr(new Object());
        /* очистки памяти явно здесь уже не производится
        вывод программы:
                Объект создан
                Объект удалён
        */
}
```

Класс `AutoPtr` сохраняет указатель на объект и сам очищает память, выделенную под него в конце. Но с таким указателем возникнут проблемы, если создать несколько объектов, хранящих один и тот же указатель:

```
Object obj = new Object();
AutoPtr<Object> ptr1(&obj);
AutoPtr<Object> ptr2(&obj);
```

Здесь произойдёт повторная попытка очистить данные по одному и тому же адресу. Такая ситуация может привести к попытке обратиться по адресу, который уже был очищен с помощью `delete`. Можно попытаться решить эту ситуацию, переопределив конструкторы копирования (например, использовать глубокое копирование с выделением новой области памяти и полным копированием объекта) и перемещения (передавать право владения второму указателю).

25. C++. Стандартные умные указатели. Отличия.

В стандартную библиотеку C++ уже добавлено несколько типов умных указателей с разными функциями:

- `std::auto_ptr` (удалён в C++ 17) - аналог реализованного выше указателя с семантикой перемещения, был первой попыткой внедрения умных указателей в язык, но имеет ряд проблем, связанных с передачей по значению в функцию, поэтому был удалён
- `std::unique_ptr` - указатель на уникальный объект, то есть не может существовать больше одного указателя на один и тот же адрес в памяти Чтобы выделить память и создать в ней объект, на который будет указывать указатель, применяется функция `std::make_unique<T>`. В качестве параметра в нее передается объект, на который будет указывать указатель.
- `std::shared_ptr` - применяется для создания указателей на объекты, на которые может указывать несколько указателей. Для данных указателей применяется механизм подсчета ссылок. Для инициализация аналогично применяется функция `std::make_shared<T>`
- `std::weak_ptr` - работает аналогично `std::shared_ptr`, но не считается владельцем объекта при подсчёте ссылок, это помогает решить ситуации, когда указатели `std::shared_ptr` указываются друг на друга, что не позволяет им вызвать деструкторы в конце выполнения программы и приводит к утечке памяти. С ним нельзя работать напрямую как с указателем (нет оператора `->`), сначала нужно привести его к `std::shared_ptr`, используя функцию `lock()`