



?

# ООП

## Классы. Отличия классов от структур. Модификаторы доступа. Поля и методы, понятие инкапсуляции.

### Отличия классов от структур

Классы - это абстракция описывающая методы, свойства, ещё не существующих объектов.

В C++ классом называются типы объявленные с помощью `class`, `struct` или `union`.

Отличий `class` от `struct` всего два:

#### *Member access control [class.access]*

Члены класса, определенного с помощью ключевого слова `class`, по умолчанию являются `private`.

Члены класса, определенного с помощью ключевого слова `struct` или `union`, по умолчанию являются `public`.

#### *Accessibility of base classes and base class members [class.access.base]*

При отсутствии спецификатора доступа (т.е. `private/protected/public`) у базового класса, базовый класс будет `public` если класс определен с помощью `struct` и `private` если класс определен с помощью `class`.

### Модификаторы доступа

- `public` - доступ открыт всем другим классам, кто видит определение данного класса.
- `private` - доступ открыт самому классу (т.е. функциям-членам данного класса) и друзьям (friend) данного класса - как функциям, так и классам. Однако производные классы не получают доступа к этим данным совсем. И все другие классы такого доступа не имеют.
- `protected` - доступ открыт классам, производным от данного. То есть, производные классы получают свободный доступ к таким свойствам или метода. Все другие классы такого доступа не имеют.

### Поля и методы, понятие инкапсуляции

Поля — это любые данные, которыми можно характеризовать объект класса.

Методы — это функции, которые могут выполнять какие-либо действия над данными (полями) класса.

Инкапсуляция — это свойство, позволяющее объединить в классе и данные, и методы, работающие с ними и скрыть детали реализации от пользователя.

## **Конструкторы. Конструктор по умолчанию. Конструктор копирования, его сигнатура и схема реализации.**

**Конструктор** — это особый тип метода класса, который автоматически вызывается при создании объекта этого же класса. Конструкторы обычно используются для инициализации переменных-членов класса значениями, которые предоставлены по умолчанию/пользователем.

**Конструкторы имеют определенные правила их именования:** конструкторы всегда должны иметь то же имя, что и класс; конструкторы не имеют типа возврата (даже void).

**Конструкторы по умолчанию.** Конструктор, который не имеет параметров (или содержит параметры, которые все имеют значения по умолчанию). Вызывается, если пользователем не указаны значения для инициализации.

**Конструктор копирования** — это особый тип конструктора, который имеет 1 параметр - ссылка на объект данного типа класса/структуры. Вызывается авто компилятором при необход. создать копию имеющегося объекта (может вызываться и вручную), использует почленную инициализацию.

Существует четыре случая вызова конструктора копирования:

1. Когда объект является возвращаемым значением
2. Когда объект передается (функции) по значению в качестве аргумента
3. Когда объект конструируется на основе другого объекта (того же класса)

Если в классе не объявлен конструктор копирования, то используется конструктор копирования, который автоматически генерируется компилятором. Этот конструктор копирования реализует побитовое копирование для получения копии объекта - это приемлемым для классов, в которых нет динамического выделения памяти. Однако, если в классе есть динамическое выделение памяти (класс использует указатели), то побитовое копирование приведет к тому, что указатели обоих объектов будут указывать на один и тот же участок памяти. А это ошибка.

## **Правила генерации компилятором конструкторов.**

**Деструкторы. Ключевое слово `this` и пример, когда оно нужно.**

**Списки инициализации и пример, когда они необходимы.**

**Явные конструкторы, ключевое слово `explicit`, пример.**

### **Правила генерации компилятором конструкторов.**

1. Конструктор копирования генерируется, если все базовые классы и члены допускают копирование
2. Конструктор по умолчанию без параметров создаётся, если класс не определяет никого конструктора
3. Оператор присваивания `T &T::operator=(const T &)` генерируется, если все базовые классы и члены допускают копирование
4. Деструктор всегда автоматически генерируется, если это возможно

### **Деструкторы.**

Деструктор также является специальной функцией-членом, такой как конструктор. Деструктор уничтожает объекты класса, созданные конструктором. Деструктор имеет то же имя, что и имя их класса, которому предшествует тильда (~).

Невозможно определить более одного деструктора. Деструктор — это единственный с, созданный конструктором. Следовательно, деструктор не может быть перегружен. Деструктор не требует никаких аргументов и не возвращает никакого значения. Он вызывается автоматически, когда объект выходит за пределы области видимости. Деструктор освобождает место в памяти, занятое объектами, созданными конструктором. В деструкторе объекты уничтожаются в обратном порядке создания объекта.

### Ключевое слово `this` и пример, когда оно нужно.

Ключевое слово `this` представляет указатель на текущий объект данного класса. Соответственно через `this` мы можем обращаться внутри класса к любым его членам.

**Пример:**

```
class Point
{
public:
    Point(int x, int y)
    {
        this->x = x;
        this->y = y;
    }
private:
    int x;
    int y;
};
```

### Списки инициализации и пример, когда они необходимы.

Список инициализации членов находится сразу же после параметров конструктора. Он начинается с двоеточия (`:`), а затем значение для каждой переменной указывается в круглых скобках. Больше не нужно выполнять операции присваивания в теле конструктора.

Списки инициализации членов позволяют инициализировать члены, а не присваивать им значения. Это единственный способ инициализации констант и ссылок, которые являются переменными-членами вашего класса. Во многих случаях использование списка инициализации может быть более результативным, чем присваивание значений переменным членам в теле конструктора. Списки инициализации работают как с переменными фундаментальных типов данных, так и с членами, которые сами являются классами.

```
class Values
{
private:
    int m_value1;
    double m_value2;
    char m_value3;

public:
    Values() : m_value1(3), m_value2(4.5), m_value3('d')
    {
        // Нет необходимости использовать присваивание
    }
};
```

### Явные конструкторы, ключевое слово `explicit`, пример.

По умолчанию язык C++ обрабатывает любой конструктор, как оператор неявного преобразования.

Явные конструкторы (с ключевым словом `explicit`) не используются для неявных конвертаций.

```

class MyClass {
    int i;
    public:
        MyClass(int j)
            {i = j;}
        // ...
};

//Объекты этого класса
//могут быть объявлены 2мя способами:
MyClass ob1(1);
MyClass ob2 = 10;

```

```

class MyClass {
    int i;
    public:
        explicit MyClass(int j)
            {i = j;}
        // ...
};

//Теперь допустимы являются
//только конструкции следующего вида:
MyClass ob(integer);

```

## **Статические поля и методы, пример. Локальные статические переменные.**

### **Статические поля и локальные статические переменные.**

**Статические поля** - представляют собой обычную переменную, которая создаётся в момент запуска программы.

Правила видимости такие же, как и для обычных полей.

Переменные-члены класса можно сделать статическими, используя ключевое слово `static`. В отличие от обычных переменных-членов, статические переменные-члены являются общими для всех объектов класса. Они создаются при запуске программы и уничтожаются, когда программа завершает свое выполнение. Следовательно, статические члены принадлежат классу, а не объектам этого класса. Также можно получить доступ к статическим членам через разные объекты класса, а также осуществляется напрямую через имя класса и оператор разрешения области видимости, но статические члены существуют, даже если объекты класса не созданы!

**Определение и инициализация статических переменных-членов класса.** Когда объявляется `static` переменная-член внутри тела класса, то компилятору сообщается о существовании статической переменной-члене, но не о её определении . Поскольку статические переменные-члены не являются частью отдельных объектов класса, то должны явно определить статический член вне тела класса — в глобальной области видимости.

```

#include <iostream>
class Anything

{
public:
    static int s_value; // объявляем статическую переменную-член
};

int Anything::s_value = 3; // определяем статическую переменную-член
int main(){ // Примечание: Мы не создаем здесь никаких объектов класса Anything
    Anything::s_value = 4;
    std::cout << Anything::s_value << '\n';
    return 0;
}

```

Зачем использовать статические переменные-члены внутри классов? Для присваивания уникального идентификатора каждому объекту класса (как вариант).

**Статические методы.** Не привязаны к какому-либо одному объекту класса, а значит они не имеют скрытого указателя `*this` (т.к. он указывает всегда на объект, с которым работает метод, а статич-ие методы могут не работать через объекты) и + могут напрямую обращаться к другим статическим членам (переменным или функциям), но не могут напрямую обращаться к нестатическим членам

(нестатические члены принадлежат объекту класса, а статические методы — нет!). Можно определять вне тела класса. Могут принимать объект, в качестве параметра или создавать объект данного класса.

```
#include <iostream>
class IDGenerator
{
private:
    static int s_nextID; // объявление статической переменной-члена

public:
    static int getNextID(); // объявление статического метода
};

// Определение статической переменной-члена находится вне тела класса. Обратите внимание, мы не используем здесь ключевое слово static
// Начинаем генерировать ID с 1
int IDGenerator::s_nextID = 1;

// Определение статического метода находится вне тела класса. Обратите внимание, мы не используем здесь ключевое слово static
int IDGenerator::getNextID() { return s_nextID++; }

int main()
{
    for (int count=0; count < 4; ++count)
        std::cout << "The next ID is: " << IDGenerator::getNextID() << '\n';
    return 0;
}
```

## Дружественные методы и классы, пример. Константные и не константные методы.

### Дружественные методы и классы, пример.

Дружественная функция — это функция, которая имеет доступ к закрытым членам класса, как если бы она сама была членом этого класса. Во всех других отношениях дружественная функция является обычной функцией. Ею может быть, как обычная функция, так и метод другого класса. Для объявления дружественной функции используется **ключевое слово friend** перед **прототипом функции**, которую вы хотите сделать дружественной классу.

```
class Anything
{
private:
    int m_value;
public:
    friend void reset(Anything &anything);
};

// Функция reset() теперь является другом класса Anything
void reset(Anything &anything)
{
    // И мы имеем доступ к закрытым членам объектов класса Anything
    anything.m_value = 0;
}
```

Один класс может быть дружественным другому классу. Это откроет всем членам первого класса доступ к закрытым членам второго класса.

```

#include <iostream>

class Values
{
private:
    int m_intValue;
public:
    Values(int intValue) { m_intValue = intValue; }

    // Делаем класс Display другом класса Values
    friend class Display;
};

class Display
{
public:
    void displayItem(Values &value)
    {
        std::cout << value.m_intValue;
    }
};

```

### Константные и не константные методы.

**Константный метод** — это метод, который гарантирует, что не будет изменять объект или вызывать неконстантные методы класса (поскольку они могут изменить объект).

```

class Anything
{
public:
    int m_value;

    Anything() { m_value= 0; }
    // ключевое слово const находится после списка параметров, но перед телом функции
    int getValue() const { return m_value; }
};

```

## Перегрузка операторов. Какие операторы нельзя перегружать. Чего можно и чего нельзя добиться перегрузкой операторов. Синтаксис перегрузки арифметических операторов.

### Перегрузка операторов.

Перегрузка операторов позволяет определить действия, которые будет выполнять оператор. Перегрузка подразумевает создание функции, название которой содержит слово operator и символ перегружаемого оператора. Функция оператора может быть определена как член класса, либо вне класса.

### Какие операторы нельзя перегружать.

Следующие операторы перегружать нельзя:

1. ?: (тернарный оператор);
2. :: (доступ к вложенным именам);
3. . (доступ к полям);
4. .\* (доступ к полям по указателю);
5. sizeof, typeid и операторы каста.

### Чего можно и чего нельзя добиться перегрузкой операторов.

Перегружать операторы стоит тогда и только тогда, когда это имеет смысл. То есть если смысл перегрузки очевиден и не несёт в себе скрытых сюрпризов.

## Синтаксис перегрузки арифметических операторов.

```
class Timer
{
public:
    int seconds;

    int operator + (int s)
    {
        return this->seconds + s;
    }
}
```

## Перегрузка префиксного и постфиксного инкремента.

## Перегрузка оператора "квадратные скобки", ее особенности.

## Перегрузка оператора "круглые скобки", пример использования.

**Перегрузка операторов инкремента и декремента.** Поскольку операторы инкремента и декремента являются унарными и изменяют свои operandы, то перегрузку следует выполнять через методы класса. **Версии префикс:** аналогична перегрузке любых других унарных операторов

**Версии постфикс:** что язык C++ использует **фиктивную переменную** (или «**фиктивный параметр**») для операторов версии постфикс. Этот фиктивный целочисленный параметр используется только с одной целью: отличить версию постфикс операторов инкремента/декремента от версии префикс.

```

#include <iostream>

class Number
{
private:
    int m_number;
public:
    Number(int number=0) : m_number(number)
    { }

    Number& operator++(); // версия префикс
    Number& operator--(); // версия префикс

    Number operator++(int); // версия постфикс
    Number operator--(int); // версия постфикс

    friend std::ostream& operator<< (std::ostream &out, const Number &n);
};

Number& Number::operator++()
{
    if (m_number == 8) // Если значением переменной m_number является 8, то выполняем сброс значения m_number на 0
        m_number = 0;
    else // В противном случае, просто увеличиваем m_number на единицу
        ++m_number;
    return *this;
}

Number& Number::operator--()
{
    if (m_number == 0) // Если значением переменной m_number является 0, то присваиваем m_number значение 8
        m_number = 8;
    else // В противном случае, просто уменьшаем m_number на единицу
        --m_number;
    return *this;
}

Number Number::operator++(int)
{
    Number temp(m_number); // Создаем временный объект класса Number с текущим значением переменной m_number

    // Используем оператор инкремента версии префикс для реализации перегрузки оператора инкремента версии постфикс
    ++(*this); // реализация перегрузки
    return temp; // Возвращаем временный объект
}

Number Number::operator--(int)
{
    Number temp(m_number); // Создаем временный объект класса Number с текущим значением переменной m_number

    // Используем оператор декремента версии префикс для реализации перегрузки оператора декремента версии постфикс
    --(*this); // реализация перегрузки
    return temp; // Возвращаем временный объект
}

std::ostream& operator<< (std::ostream &out, const Number &n)
{
    out << n.m_number;
    return out;
}

int main()
{
    Number number(7);
    std::cout << number;
    std::cout << ++number; // вызывается Number::operator++();
    std::cout << number++; // вызывается Number::operator++(int);
    std::cout << number;
    std::cout << --number; // вызывается Number::operator--();
    std::cout << number--; // вызывается Number::operator--(int);
    std::cout << number;

    return 0;
}

```

Результат выполнения программы: [6778776](#)

**Перегрузка оператора индексации []**, которого должна выполняться через метод класса и всегда будет принимать один параметр: значение индекса (элемент массива, к которому требуется доступ). Можно определить отдельно неконстантную и константную версии operator. Неконстантная версия будет использоваться с неконстантными объектами, а версия const — с объектами const + преимуществом

ещё в том, что мы можем выполнять проверку передаваемых значений индекса. **Передаваемый аргумент не обязательно должен быть целым числом.**

```
#include <iostream>

class IntArray
{
private:
    int m_array[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }; // указываем начальные значения

public:
    int& operator[] (const int index);
    const int& operator[] (const int index) const;
};

int& IntArray::operator[] (const int index) // для неконстантных объектов: может использоваться как для присваивания значений элемента
{
    assert(index >= 0 && index < 10);
    return m_array[index];
}

const int& IntArray::operator[] (const int index) const // для константных объектов: используется только для просмотра (вывода) элемента
{
    assert(index >= 0 && index < 10);
    return m_array[index];
}

int main()
{
    IntArray array;
    array[4] = 5; // хорошо: вызывается неконстантная версия operator[]()
    std::cout << array[4];

    const IntArray carray;
    //carray[4] = 5; // ошибка компиляции: вызывается константная версия operator[](), которая возвращает константную ссылку. Выполнят
    std::cout << carray[4];

    return 0;
}
```

**Указатели на объекты:**

```
int main()
{
    IntArray *array = new IntArray;
    (*array)[4] = 5; // сначала разыменовываем указатель для получения объекта array, а затем вызываем operator[]
    delete array;
    return 0;
}
```

**Перегрузка оператора ()**. Оператор **()** позволяет изменять как тип параметров, так и их количество. Перегрузка () должна осуществляться через метод класса.

Перегрузка оператора **( )** с двумя параметрами используется для получения доступа к двумерным массивам или для возврата подмножеств одномерного массива.

```

#include <iostream>
#include <cassert> // для assert()

class Matrix
{
private:
    double data[5][5];
public:
    Matrix()
    {
        // Присваиваем всем элементам массива значение 0.0
        for (int row=0; row < 5; ++row)
            for (int col=0; col < 5; ++col)
                data[row][col] = 0.0;
    }

    double& operator()(int row, int col);
    const double& operator()(int row, int col) const;
    void operator()();

};

double& Matrix::operator()(int row, int col)
{
    assert(col >= 0 && col < 5);
    assert(row >= 0 && row < 5);

    return data[row][col];
}

const double& Matrix::operator()(int row, int col) const
{
    assert(col >= 0 && col < 5);
    assert(row >= 0 && row < 5);

    return data[row][col];
}

void Matrix::operator(){}
{
    // Сбрасываем значения всех элементов массива на 0.0
    for (int row=0; row < 5; ++row)
        for (int col=0; col < 5; ++col)
            data[row][col] = 0.0;
}

int main()
{
    Matrix matrix;
    matrix(2, 3) = 3.6;
    matrix(); // выполняем сброс
    std::cout << matrix(2, 3);

    return 0;
}

```

Результат выполнения программы: 

Перегрузка оператора () используется в реализации функторов (или «функциональных объектов») — классы, которые работают как функции (+ в том, что могут хранить данные в переменных-членах (поскольку они сами являются классами))

```

#include <iostream>

class Accumulator
{
private:
    int m_counter = 0;

public:
    Accumulator()
    {
    }

    int operator() (int i) { return (m_counter += i); }

};

int main()
{
    Accumulator accum;
    std::cout << accum(30) << std::endl; // выведется 30
    std::cout << accum(40) << std::endl; // выведется 70
}

```

```
    return 0;
}
```

Перегрузка операторов приведения типа, ее особенности и недостатки, пример. Особенности перегрузки логических операторов и оператора "запятая".

## Перегрузка операторов приведения типа, ее особенности и недостатки, пример.

Операторы преобразования (conversion operator) определяют преобразование объекта одного типа в

```
class Dollars
{
private:
    int m_dollars;
public:
    Dollars(int dollars=0)
    {
        m_dollars = dollars;
    }

    // Перегрузка операции преобразования значений типа Dollars в значения типа int
    operator int() { return m_dollars; }
};
```

Здесь есть две вещи, на которые следует обратить внимание

другой. Они имеют следующий общий синтаксис: `operator тип() const ;`

1. В качестве функции перегрузки используется метод `operator int()`. Обратите внимание, между словом `operator` и типом, в который мы хотим выполнить конвертацию (в данном случае, тип `int`), находится пробел.
2. Функция перегрузки не имеет типа возврата. Язык C++ предполагает, что вы будете возвращать корректный тип.

## Особенности перегрузки логических операторов и оператора "запятая".

Принципы перегрузки **операторов сравнения** те же, что и в перегрузке других операторов, которые мы рассматривали на предыдущих уроках. Поскольку все операторы сравнения являются бинарными и не изменяют свои левые operandы, то выполнять перегрузку следует через дружественные функции.

```

class Car
{
private:
    std::string m_company;
    std::string m_model;

public:
    Car(std::string company, std::string model)
        : m_company(company), m_model(model)
    {
    }

    friend bool operator==(const Car &c1, const Car &c2);
    friend bool operator!=(const Car &c1, const Car &c2);
};

bool operator==(const Car &c1, const Car &c2)
{
    return (c1.m_company == c2.m_company &&
            c1.m_model == c2.m_model);
}

bool operator!=(const Car &c1, const Car &c2)
{
    return !(c1 == c2);
}

```

В языке C++ оператор ‘,’ может быть перегружен. При перегрузке оператора ‘,’ в классе должна быть объявлена операторная функция `operator, ()`. В тело операторной функции можно поместить любой код. При желании оператор ‘,’ может выполнять любые нестандартные операции над объектами класса.

В случае стандартного использования оператора ‘,’ нужно учесть следующие особенности:

- оператор ‘,’ считается бинарным. Поэтому операторная функция `operator,()` получает один параметр;
- при использовании перегруженного оператора ‘,’ в операции присваивания принимается во внимание последний аргумент (этот аргумент есть результатом операции). Все другие аргументы игнорируются.

```

class ClassName
{
    // операторная функция, которая перегружает оператор ','
    ClassName operator,(ClassName obj) { }
};

```

## **Понятие наследования. Синтаксис наследования, правила сокрытия имен при наследовании. Схема использования объектов производного класса в местах, где требуются объекты базового класса.**

### **Понятие наследования.**

**Наследование** — это инструмент, позволяющий описать новый класс на основе уже существующего

### **Синтаксис наследования, правила сокрытия имен при наследовании.**

```
class A {...};  
class B : public A {...};
```

Спецификатор доступа в родительском классе	Спецификатор доступа при наследовании типа public в дочернем классе	Спецификатор доступа при наследовании типа private в дочернем классе	Спецификатор доступа при наследовании типа protected в дочернем классе
public	public	private	protected
private	Недоступен	Недоступен	Недоступен
protected	protected	private	protected

### Схема использования объектов производного класса в местах, где требуются объекты базового класса.

В метод, принимающий экземпляр базового класса можно передавать экземпляры производного. В обратную сторону это не работает

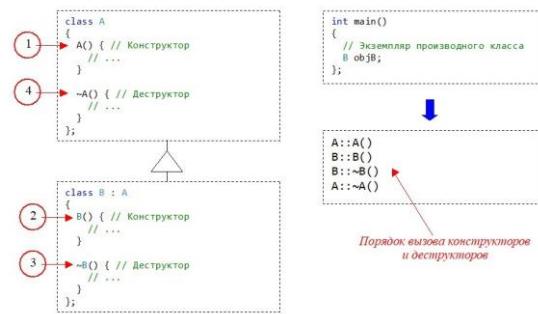
```
class A {...};  
class B : public A {...};  
  
void Foo(A a) {...};  
  
int main() {  
    B b();  
    Foo(b); // допустимо  
}
```

### Порядок выполнения конструкторов и деструкторов при наследовании, вызов конструктора родителя из конструктора потомка. Виды наследования: публичное, приватное, защищенное. Пример использования приватного наследования.

#### Порядок выполнения конструкторов и деструкторов при наследовании.

Если два класса образуют иерархию наследования, то при создании экземпляра производного класса сначала вызывается конструктор базового класса конструирующий объект производного класса. Затем этот конструктор становится недоступен и дополняется кодом конструктора производного класса (сначала происходит инициализация данных базового класса, затем инициализация данных производного класса)

Если классы образуют иерархию, деструкторы этих классов вызываются в обратном порядке по отношению к вызову конструкторов. Сначала вызывается деструктор производного класса, затем вызывается деструктор базового класса.



### Вызов конструктора родителя из конструктора потомка.

```
#include <iostream>

class Parent
{
private: // наш m_id теперь закрытый
    int m_id;

public:
    Parent(int id=0)
        : m_id(id)
    {
    }

    int getId() const { return m_id; }
};

class Child: public Parent
{
private: // наш m_value теперь закрытый
    double m_value;

public:
    Child(double value=0.0, int id=0)
        //Несработает: m_value(value), m_id(id)
        : Parent(id), // вызывается конструктор Parent(int) со значением id!
        m_value(value)
    {
    }

    double getValue() const { return m_value; }
};

int main()
{
    Child child(1.5, 7); // вызывается конструктор Child(double, int)
    std::cout << "ID: " << child.getId() << '\n';
    std::cout << "Value: " << child.getValue() << '\n';

    return 0;
}
```

**Виды наследования: публичное (открытое), приватное (закрытое), защищенное.** Для определения типа наследования нужно просто указать нужное ключевое слово возле наследуемого класса:

```
// Открытое наследование
class Pub: public Parent
{};

// Закрытое наследование
class Pri: private Parent
{};

// Защищенное наследование
class Pro: protected Parent
{};

class Def: Parent // по умолчанию язык C++ устанавливает закрытое наследование
{}
```

Нужно помнить **следующие правила**: класс всегда имеет доступ к своим (не наследуемым) членам; доступ к члену класса основывается на его спецификаторе доступа; дочерний класс имеет доступ к унаследованным членам родительского класса на основе спецификатора доступа этих членов в родительском классе.

**Публичное (открытое)**: когда вы открыто наследуете родительский класс, то унаследованные public-члены остаются public, унаследованные protected-члены остаются protected, а унаследованные private-члены остаются недоступными для дочернего класса. Ничего не меняется.

**Приватное (закрытое)**: все члены родительского класса наследуются как закрытые. Это означает, что private-члены остаются недоступными, а protected- и public-члены становятся private в дочернем классе.

**Внимание:** это не влияет на то, как дочерний класс получает доступ к членам родительского класса! Это влияет только на то, как другими объектами осуществляется доступ к этим членам через дочерний класс.

Спецификатор доступа в родительском классе	Спецификатор доступа при наследовании типа public в дочернем классе	Спецификатор доступа при наследовании типа private в дочернем классе	Спецификатор доступа при наследовании типа protected в дочернем классе
public	public	private	protected
private	Недоступен	Недоступен	Недоступен
protected	protected	private	protected

#### Пример использования приватного наследования:

```

class Base
{
    private:
        int privateBase;
    protected:
        int protBase;
    public:
        int pubBase;
};

class Derived : private Base
{
    //унаследованные данные класса
    //недоступно:
    // int privateBase;
    //private:
    // int protBase;
    //private:
    // int pubBase;

    public:
        void updateDerived()
        {
            //privateBase=0; //нельзя получить доступ
                           //к private данным Base
            protBase=0;
            pubBase=0;
        }
};

class Derived1 : public Derived
{
    public:
        void updateDerived1()
        {
            //privateBase=1; //нельзя получить доступ
                           //к private данным Base
            //protBase=1;//protBase недоступно,
                           //потому что Derived использовал
                           //private при наследовании от Base
            //pubBase=1; //pubBase недоступно, потому что
                           //Derived использовал private
        }
};

```

```

        //при наследовании от Base
    }

};

int main()
{
    //При вызове извне, получить доступ не получится ни к чему
    Derived dd;
    //dd.privateBase=3; недоступно
    //dd.protBase=3; недоступно
    //dd.pubBase=3; недоступно
}

```

Закрытое наследование может быть полезно, когда дочерний класс не имеет очевидной связи с родительским классом, но использует его в своей реализации. В таком случае мы не хотим, чтобы открытый интерфейс родительского класса был доступен через объекты дочернего класса (как это было, когда мы использовали открытый тип наследования).

**Защищенное:** с защищенным наследованием, public- и protected-члены становятся protected, а private-члены остаются недоступными.

## Множественное наследование. Проблема неоднозначности вызова метода из базовых классов и ее решение. Проблема ромбовидного наследования и ее решение.

### Множественное наследование.

Множественное наследование позволяет одному дочернему классу иметь несколько родителей.

### Проблема неоднозначности вызова метода из базовых классов и ее решение.

**Суть проблемы:** Несколько родительских классов имеют метод с одним и тем же именем

**Решение:** Явно указывать какую версию метода мы хотим вызвать

```

class A {
public:
    void Foo() {...}
}

class B {
public:
    void Foo() {...}
}

class MyClass : public A, public B { ... }

int main() {
    MyClass obj();

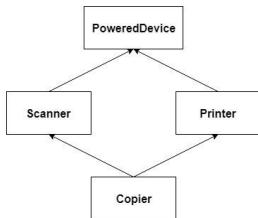
    // явно указываем какую версию метода вызвать
    obj.A::Foo();
    obj.B::Foo();

    return 0;
}

```

### Проблема ромбовидного наследования и ее решение.

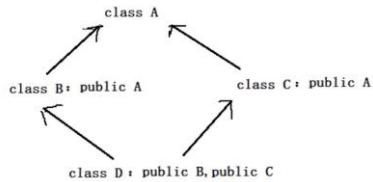
**Суть проблемы:** Это ситуация, когда один класс имеет 2 родительских класса, каждый из которых, в свою очередь, наследует свойства одного и того же родительского класса. Иллюстративно мы получаем форму ромба.



**Решение:** Использование виртуального наследования

```
class Copier: public Scanner, virtual public Printer
```

## Виртуальное наследование, его особенности.



**Виртуальное наследование** решает проблему избыточности данных и проблему неоднозначности.

**Почему существует избыточность данных?** - Прежде всего, класс В и класс С наследуют класс A, поэтому класс В и класс С должны содержать переменную-член класса A, то есть есть одна в классе В `int a` и еще одну в классе С `int a`. Затем, когда класс D наследует класс В и класс С, тогда класс D должен содержать две `int a`. Это так называемая избыточность данных. **Почему это вызывает двусмысленность?** Существует два типа `int a`. Итак, когда мы хотим использовать объект класса D для доступа `int a`, тогда компилятор не знает, к какому из них осуществляется доступ `a`. То, к чему осуществляется доступ, унаследовано от класса В `a` или доступ наследуется от класса С `a` это так называемая двусмысленность. (Конечно, для двусмысленности мы можем использовать "Название класса ::" Чтобы решить проблему, но этот метод не может решить проблему избыточности данных, поэтому существует виртуальное наследование)

C++ по умолчанию не создает ромбовидного наследования: компилятор обрабатывает каждый путь наследования отдельно, в результате чего объект `D` будет на самом деле содержать два разных подобъекта `A`, и при использовании членов `A` потребуется указать путь наследования (`B::A` или `C::A`). Чтобы сгенерировать ромбовидную структуру наследования, необходимо воспользоваться **виртуальным** наследованием класса `A` на нескольких путях наследования: если оба наследования от `A` к `B` и от `A` к `C` помечаются спецификатором `virtual` (например, `class B : virtual public A`), C++ специальным образом проследит за созданием только одного подобъекта `A`, и использование членов `A` будет работать корректно. Если виртуальное и невиртуальное наследования смешиваются, то получается один виртуальный подобъект `A` и по одному невиртуальному подобъекту `A` для каждого пути невиртуального наследования к `A`. При виртуальном вызове метода виртуального базового класса используется так называемое правило доминирования: компилятор запрещает виртуальный вызов метода, который был перегружен на нескольких путях наследования.

## Виртуальные функции, их применение. Понятие полиморфизма. Разница в поведении виртуальных и не виртуальных функций при наследовании.

## Виртуальные функции, их применение.

Виртуальная функция в языке — это особый тип функции, которая, при её вызове, выполняет «наиболее» дочерний метод, который существует между родительским и дочерними классами.

```
#include <iostream>

class Parent
{
public:
    virtual const char* getName() { return "Parent"; } // добавили ключевое слово virtual
};

class Child: public Parent
{
public:
    virtual const char* getName() { return "Child"; }
};

int main()
{
    Child child;
    Parent &rParent = child;

    // результат: rParent is a Child
    std::cout << "rParent is a " << rParent.getName() << '\n';

    return 0;
}
```

## Понятие полиморфизма.

Полиморфизм — это способность объекта использовать методы производного класса, который не существует на момент создания базового.

## Разница в поведении виртуальных и не виртуальных функций при наследовании.

## Чисто виртуальные функции. Понятие абстрактного класса, особенность абстрактных классов.

### Чисто виртуальные функции.

Часто в самом базовом классе сами виртуальные функции фиктивны и имеют пустое тело.

Определенное значение им придается лишь в порожденных классах. Такие функции называются **чистыми виртуальными функциями**.

**Чистая виртуальная функция** — это метод класса, тело которого не определено.

При создании чистой виртуальной функции, вместо определения (написания тела) виртуальной функции, мы просто присваиваем ей значение 0.

```
class Parent {
public:
    virtual void Foo() = 0;
}
```

Виртуальная таблица в языке C++ — это таблица поиска функций для выполнения вызовов функций в режиме позднего (динамического) связывания. Виртуальную таблицу еще называют «**vtable**», «**таблицей виртуальных функций**» или «**таблицей виртуальных методов**».

Во-первых, любой класс, который использует виртуальные функции (или дочерний класс, родительский класс которого использует виртуальные функции), имеет свою собственную виртуальную таблицу. Это обычный статический **массив**, который создается компилятором во время компиляции. Виртуальная таблица содержит по одной записи на каждую виртуальную функцию, которая может быть вызвана

объектами класса. Каждая запись в этой таблице — это **указатель на функцию**, указывающий на наиболее дочерний метод, доступный объекту этого класса.

Во-вторых, компилятор также **добавляет скрытый указатель на родительский класс, который мы будем называть `*__vptr`**. Этот указатель автоматически создается при создании объекта класса и указывает на виртуальную таблицу этого класса.

### Понятие абстрактного класса, особенность абстрактных классов.

Любой класс с одной и более чистыми виртуальными функциями становится **абстрактным классом**, объекты которого создавать нельзя.

## Виртуальные деструкторы, чисто виртуальные деструкторы, их назначение. Проблема с вызовами виртуальных функций в конструкторах.

**Виртуальные деструкторы.** Основное правило: **если в классе присутствует хотя бы одна виртуальная функция, деструктор также следует сделать виртуальным.** Деструктор по умолчанию виртуальным не будет, поэтому следует объявить его явно.

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

using std::cout;
using std::endl;

class A {
public:
    A() { cout << "A()" << endl; }
    virtual ~A() { cout << "~A()" << endl; }

};

class B : public A {
public:
    B() { cout << "B()" << endl; }
    ~B() { cout << "~B()" << endl; }
};

int main()
{
    A * pA = new B;
    delete pA;
    return EXIT_SUCCESS;
}
```

Если базовый деструктор указать как не `virtual`, то при разрушении происходит утечка памяти, потому как деструктор производного класса не вызывается: `A() B() ~A()`. Происходит это потому, что удаление производится через указатель на базовый класс и для вызова деструктора компилятор использует раннее связывание. Деструктор базового класса не может вызвать деструктор производного, потому что он о нем ничего не знает. В итоге часть памяти, выделенная под производный класс, безвозвратно теряется. Чтобы этого избежать, деструктор в базовом классе должен быть объявлен как виртуальный. Теперь порядок вызовов: `A() B() ~B()~A()`. Происходит так потому, что для вызова деструктора используется позднее связывание, то есть при разрушении объекта берется указатель на класс, затем из таблицы виртуальных функций определяется адрес нужного нам деструктора, а это деструктор производного класса, который после своей работы, как и полагается, вызывает деструктор базового. Итог: объект разрушен, память освобождена.

**Чисто виртуальные деструкторы.** Деструкторам разрешено быть «чистыми». Объявляются такие деструкторы точно так же, как чистые виртуальные функции, например:

```
virtual ~VBase() = 0;
```

Класс, в котором определен чистый виртуальный деструктор **является абстрактным**, и создавать объекты этого класса запрещено. Класс-наследник не является абстрактным классом! При объявлении чисто виртуального деструктора нужно написать и его определение. Объясняется тем, что деструктор наследника обязательно вызывает деструктор базового класса, поэтому чистый деструктор все же должен быть определен.

```
class Abstract // абстрактный класс
{ public:
    virtual ~Abstract()=0; // чистый виртуальный деструктор
};

Abstract::~Abstract() // определение чистого деструктора
{ cout << "MAbstract"<< endl; }

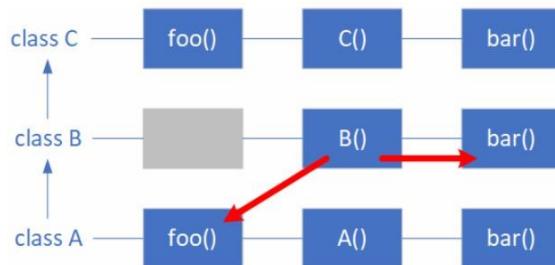
// наследник - не абстрактный класс
class NotAbstract: public Abstract { };
```

Для объекта класса `Abstract` компилятор немедленно выдает сообщение о том, что нельзя создавать объекты абстрактного класса.

**Объект класса-наследника создается без проблем. Определим в наследнике явный деструктор:**

```
virtual ~NotAbstract() // явный деструктор
{ cout << "NotAbstract"<< endl; }
```

**Проблема с вызовами виртуальных функций в конструкторах.** Когда в конструкторе вызывается виртуальная функция, она работает только в пределах базовых или создаваемого в данный момент классов. Конструкторы в классах-наследниках ещё не вызывались, и поэтому реализованные в них виртуальные функции не будут вызваны.



Пояснения: от класса *A* наследуется класс *B*; от класса *B* наследуется класс *C*; функции *foo* и *bar* являются виртуальными; у функции *foo* нет реализации в классе *B*.

```

#include <iostream>

class A
{
public:
    A() { std::cout << "A()\n"; }
    virtual void foo() { std::cout << "A::foo()\n"; }
    virtual void bar() { std::cout << "A::bar()\n"; }
};

class B : public A
{
public:
    B() {
        std::cout << "B()\n";
        foo();
        bar();
    }
    void bar() { std::cout << "B::bar()\n"; }
};

class C : public B
{
public:
    C() { std::cout << "C()\n"; }
    void foo() { std::cout << "C::foo()\n"; }
    void bar() { std::cout << "C::bar()\n"; }
};

int main()
{
    C x;
    return 0;
}

```

Результат: A() B() A::foo() B::bar() C(). Можно легко забыть и считать, что функции *foo* и *bar* будут вызваны из крайнего наследника, т.е. из класса *C*.

## Понятие шаблонов. Виды полиморфизма: статический и динамический.

### Понятие шаблонов.

Существуют шаблоны функций и шаблоны классов.

*Шаблоны функций* – это обобщенное описание поведения функций, которые могут вызываться для объектов разных типов.

```

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

```

*Шаблоны классов* – обобщенное описание пользовательского типа, в котором могут быть параметризованы атрибуты и операции типа.

```

template<class T>
class Array {
    T *_m_data;
    // ...
}

```

### Виды полиморфизма: статический и динамический.

Статический полиморфизм - Полиморфизм во время компиляции.

Компиляции, компилятор завершает работу во время компиляции. Компилятор может определить, какую функцию вызывать, основываясь на типе аргумента функции (который может быть неявно преобразован по типу). Функция вызывается, в противном случае возникает ошибка компиляции.

Существует два способа реализации статического полиморфизма:

- Перегрузка функций: включая перегрузку обычных функций и перегрузку функций-членов
- Использование шаблонов функций

#### **Динамический полиморфизм - Динамическое связывание**

Полиморфизм во время выполнения, во время выполнения программы (период без компиляции), чтобы определить фактический тип ссылочного объекта и вызвать соответствующий метод в соответствии с его фактическим типом.

## **Шаблонные классы, синтаксис их определения. Специализация шаблонов, синтаксис определения, пример. Параметры шаблонов, не являющиеся типами, пример.**

### **Шаблонные классы, синтаксис их определения.**

**Шаблоны классов** – обобщенное описание пользовательского типа, в котором могут быть параметризованы атрибуты и операции типа.

```
template<class T>
class Array {
    T *m_data;
    // ...
}
```

### **Специализация шаблонов, синтаксис определения, пример.**

Иногда может понадобиться, чтобы реализация шаблона функции для одного типа данных отличалась от реализации шаблона функции для другого типа данных.

Специализация шаблонов именно для этого и предназначена.

```
template <class T>
class Repository
{
private:
    T m_value;
public:
    Repository(T value)
    {
        m_value = value;
    }

    void print()
    {
        std::cout << m_value << '\n';
    }
};

template <>
void Repository<double>::print()
{

    std::cout << std::scientific << m_value << '\n';
}
```

Когда компилятору нужно будет создать экземпляр `Repository<double>::print()`, он увидит, что мы уже явно определили эту функцию, и поэтому он будет использовать именно этот экземпляр, а не копировать общую для всех типов данных версию шаблона функции `print()`.

## **Параметры шаблонов, не являющиеся типами, пример.**

Шаблонный параметр, не являющийся типом, – это параметр шаблона, в котором тип параметра предопределен, и он заменяется значением `constexpr`, переданным в качестве аргумента.

```
#include <iostream>

// size является целочисленным параметром, не являющимся типом
template <typename T, int size>
class StaticArray
{
private:
    // Параметр, не являющийся типом, управляет размером массива
    T m_array[size] {};
};

// использование: StaticArray<int, 12>
```

## **Шаблонные функции, синтаксис объявления и использования.**

### **Шаблонные функции, синтаксис объявления и использования.**

**Шаблоны функций** – это обобщенное описание поведения функций, которые могут вызываться для объектов разных типов.

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

// использование: max(4, 8) или max<int>(4, 8)
```

## **Приведение типов: static\_cast, dynamic\_cast, разница между ними, примеры применения.**

### **Приведение типов: static\_cast, dynamic\_cast, разница между ними, примеры применения.**

#### **static\_cast**

Оператор `static_cast` используется для явного приведения типов следующим образом:

```
static_cast<int>(char);
```

Основным преимуществом оператора `static_cast` является проверка его выполнения компилятором во время компиляции, что усложняет возможность возникновения непреднамеренных проблем.

**dynamic\_cast** `dynamic_cast` это ещё один оператор явного преобразования типов. Он используется для преобразования из базового типа к производному. Этот процесс называется **приведением к дочернему типу** (или «**понижжающим приведением типа**»).

```
class A {...}
class B : public A {...}

A* a = new B();
B* b = dynamic_cast<B*>(a);
```

Если `dynamic_cast` не может выполнить конвертацию, то он возвращает нулевой указатель.



## ⊕ Приведение типов `const_cast`, пример применения. Приведение типов `reinterpret_cast`, пример применения.

### Приведение типов `const_cast`, пример применения.

Явное приведение типа `const_cast` используют для того, чтобы отбросить квалификатор `const` у изначально не константных данных или добавить квалификатор `const`. `const_cast` не влияет на оригинальную переменную: не убирает её `const` и не добавляет ей `const`

```
#include <iostream>

using namespace std;

int main()
{
    const int i = 100;
    const int *ptr = &i;

    int * pi = const_cast<int*>(&i); //Снятие const с переменной, на которую указывает pi
    (*pi) = 200; //Изменение переменной, которая обозначена const

    cout << (void*)&i << '\t' << i << '\n'; //На экране можно увидеть один адрес
    cout << (void*)pi << '\t' << *pi << '\n'; //Но разные значения
}
```

### Приведение типов `reinterpret_cast`, пример применения.

Приведение типов без проверки. `reinterpret_cast` — непосредственное указание компилятору.

Применяется только в случае полной уверенности программиста в собственных действиях.

```
int* a = new int();
void* b = reinterpret_cast<void*>(a);
int* c = reinterpret_cast<int*>(b);
// a и c содержат одно и тоже значение
```

## Ключевое слово `typeid`, динамическая проверка типа, вывод имени типа, пример.

В языке C++ оператор `typeid` возвращает ссылку на объект `type_info`, описывающий тип объекта, к которому принадлежит оператор `typeid`. общая форма записи оператора `typeid`

такова: `typeid( объект )`. Оператор `typeid` поддерживает в языке C++ возможность идентификации динамической информации о типе (RTTI). Класс `type_info` определяет следующие открытые члены.

```
bool operator == (const type_info &ob) const;
bool operator != (const type_info &ob) const;
bool before (const type_info &ob) const;
const char *name() const;
```

Перегруженные операторы `==` и `!=` служат для сравнения типов. Функция `before()` возвращает значение `true`, если возвращающий объект в порядке сопоставления стоит перед объектом, используемым в качестве параметра. Эта функция предназначена в основном для внутреннего использования. Её значение возврата не имеет ничего общего с наследованием или иерархией классов. Функция `name()` возвращает указатель на имя типа.

Если оператор `typeid` применяется к указателю полиморфного класса, он автоматически возвращает тип объекта, на который он указывает. (полиморфный класс — это класс который содержит хотя бы одну виртуальную функцию.) Следовательно, оператор `typeid` можно использовать для определения типа объекта, адресуемого указателем на базовый класс.

```

#include <cxxabi.h>
#define typeid_name(names) abi::__cxa_demangle(typeid(names).name(), 0, 0, NULL)
class A {};

class B : public A {};

int main()
{
    A a, *pa;
    B pb;
    pa = &pb;
    cout << typeid_name(A) << endl; // A
    cout << typeid_name(a) << endl; // A
    cout << typeid_name(pa) << endl; // A*
    cout << typeid_name(*pa) << endl; // A
    cout << typeid_name(pb) << endl; // B

    exit(EXIT_SUCCESS);
}

```

```

class A
{
public:
    virtual void fun() {}
};

class B : public A {};

int main()
{
    A a, *pa;
    B pb;
    pa = &pb;
    cout << typeid_name(A) << endl; // A
    cout << typeid_name(a) << endl; // A
    cout << typeid_name(pa) << endl; // A*

    cout << typeid_name(*pa) << endl; // B

    cout << typeid_name(pb) << endl; // B

    exit(EXIT_SUCCESS);
}

```

**Если операнд оператора typeid является типом класса, который содержит хотя бы одну виртуальную функцию, а выражение является ссылкой на базовый класс, оператор typeid указывает тип производного класса базового объекта.**

## ?] Понятие итераторов. Классификация итераторов: input, output, односторонние, двунаправленные, произвольного доступа. Операции, поддерживаемые итераторами.

### Понятие итераторов.

**Итератор** — это объект, который способен перебирать элементы контейнерного класса без необходимости пользователю знать реализацию определенного контейнерного класса. Во многих контейнерах (особенно в списках и в ассоциативных контейнерах) итераторы являются основным способом доступа к элементам этих контейнеров.

### Классификация итераторов: input, output, односторонние, двунаправленные, произвольного доступа.

Итератор	Описание
для чтения	Читают значения с движением вперед. Могут быть инкрементированы, сравниены и разыменованы.
для записи	Пишут значения с движением вперед. Могут быть инкрементированы и разыменованы.
однонаправленные	Читают или пишут значения с движением вперед. Комбинируют функциональность предыдущих двух типов с возможностью сохранять значение итератора.
дву направленные	Читают и пишут значения с движением вперед или назад. Похожи на однонаправленные, но их также можно инкрементировать и декрементировать.
с произвольным доступом	Читают и пишут значения с произвольным доступом. Самые мощные итераторы, сочетающие функциональность двунаправленных итераторов и возможность выполнения арифметики указателей и сравнений указателей.
обратные	Или итераторы с произвольным доступом, или двунаправленные, движущиеся в обратном направлении.

### Операции, поддерживаемые итераторами.

Об итераторе можно думать, как об **указателе** на определенный элемент контейнерного класса с дополнительным набором перегруженных операторов для выполнения четко определенных функций:

1. **Оператор `*`** возвращает элемент, на который в данный момент указывает итератор.
2. **Оператор `++`** перемещает итератор к следующему элементу контейнера. Большинство итераторов также предоставляют оператор `--` для перехода к предыдущему элементу.
3. **Операторы `==` и `!=`** используются для определения того, указывают ли оба итератора на один и тот же элемент или нет. Для сравнения значений, на которые указывают оба итератора, нужно сначала разыменовать эти итераторы, а затем использовать оператор `==` или оператор `!=`.
4. **Оператор `=`** присваивает итератору новую позицию (обычно начальный или конечный элемент контейнера). Чтобы присвоить значение элемента, на который указывает итератор, другому объекту, нужно сначала разыменовать итератор, а затем использовать оператор `=`.

Каждый контейнерный класс имеет **4 основных метода для работы с оператором `=`**:

1. **метод `begin()`** возвращает итератор, представляющий начальный элемент контейнера;
2. **метод `end()`** возвращает итератор, представляющий элемент, который находится после последнего элемента в контейнере;
3. **метод `cbegin()`** возвращает константный (только для чтения) итератор, представляющий начальный элемент контейнера;
4. **метод `cend()`** возвращает константный (только для чтения) итератор, представляющий элемент, который находится после последнего элемента в контейнере.

### Константные и неконстантные итераторы. Класс `std::reverse_iterator`, его примерная реализация.

#### Константные и неконстантные итераторы.

Константные итераторы, в отличие от неконстантных, не позволяют изменять значения на которые они указывают

## **Класс std::reverse\_iterator, его примерная реализация.**

std::reverse\_iterator это адаптер для итераторов, который меняет направление данного итератора

```
template<typename Itr>
class reverse_iterator {
    Itr itr;
public:
    constexpr explicit reverse_iterator(Itr itr): itr(itr) {}
    constexpr auto& operator*() {
        return *std::prev(itr); // <== returns the content of prev
    }
    constexpr auto& operator++() {
        --itr;
        return *this;
    }
    constexpr friend bool operator!=(reverse_iterator<Itr> a, reverse_iterator<Itr> b) {
        return a.itr != b.itr;
    }
}
```

## **Классы итераторов для вставок (insert\_iterator), их схема реализации и использования. Функции std::inserter, std::front\_inserter и std::back\_inserter, схема их реализации и пример использования.**

Чтобы было возможно иметь дело с вставкой таким же образом, как с записью в массив, в библиотеке обеспечивается специальный вид адаптеров итераторов, называемых *итераторами вставки* (*insert iterators*). (могли вставлять данные, а не перезаписывать их)

**Итератор вставки** создаётся из контейнера и, возможно, одного из его итераторов, указывающих, где вставка происходит, если это ни в начале, ни в конце контейнера. Итераторы вставки удовлетворяют требованиям итераторов вывода. *operator\** возвращает непосредственно сам итератор вставки. Присваивание *operator=(const T& x)* определено для итераторов вставки, чтобы разрешить запись в них, оно вставляет *x* прямо перед позицией, куда итератор вставки указывает. Другими словами, итератор вставки подобен курсору, указывающему в контейнер, где происходит вставка. **back\_insert\_iterator** вставляет элементы в конце контейнера, **front\_insert\_iterator** вставляет элементы в начале контейнера, а **insert\_iterator** вставляет элементы, куда итератор указывает в контейнере. **back\_inserter**, **front\_inserter** и **inserter** - три функции, создающие итераторы вставки из контейнера.

## **Понятие исключений, общая идея, аргументы за и против применения исключений. Оператор throw и конструкция try...catch, схема ее работы.**

## **Правила генерации и перехвата исключений, производные исключения.**

## **Понятие исключений, общая идея, аргументы за и против применения исключений.**

Исключения - ошибки, которые могут вызывать как вручную, так и непосредственно системой. Они требуют от вызывающего кода обработки или завершают выполнение программы.

В C++ в качестве сигнала об исключении может использоваться любой тип данных.

Минусы использования исключений:

### ▼ Очистка памят

Одной из самых больших проблем используя исключения, является проблема очистки выделенных ресурсов после генерации исключения.

### ▼ Исключения и деструкторы

В отличие от **конструкторов**, где генерация исключений может быть полезным способом указать, что создать объект не удалось, исключения **никогда не должны генерироваться в деструкторах**.

Проблема возникает, когда исключение генерируется в деструкторе во время **раскручивания стека**. Если это происходит, то компилятор оказывается в ситуации, когда он не знает, продолжать ли процесс раскручивания стека или обработать новое исключение. Конечным результатом будет немедленное прекращение выполнения вашей программы.

### ▼ Проблемы с производительностью

Исключения имеют свою небольшую цену производительности. Они увеличивают размер вашего исполняемого файла и также могут заставить его выполнять медленнее из-за дополнительной проверки, которая должна быть выполнена.

## **Оператор throw и конструкция try...catch, схема ее работы.**

В языке C++ оператор **throw** используется для сигнализирования о возникновении исключения или ошибки. Сигнализирование о том, что произошло исключение, называется **генерацией исключения** (или «**выбрасыванием исключения**»).

Для использования оператора **throw** применяется ключевое слово **throw**, а за ним указывается значение любого типа данных, которое вы хотите задействовать, чтобы сигнализировать об ошибке. Как правило, этим значением является код ошибки, описание проблемы или настраиваемый класс-исключение.

В языке C++ мы используем **ключевое слово try** для определения блока стейтментов (так называемого «**блока try**»). Блок **try** действует как наблюдатель в поисках исключений, которые были выброшены каким-либо из операторов в этом же блоке **try**.

**Ключевое слово catch** используется для определения блока кода (так называемого «**блока catch**»), который обрабатывает исключения определенного типа данных.

При выбрасывании исключения (оператор **throw**), точка выполнения программы немедленно переходит к ближайшему блоку **try**. Если какой-либо из обработчиков **catch**, прикрепленных к блоку **try**, обрабатывает этот тип исключения, то точка выполнения переходит в этот обработчик и, после выполнения кода блока **catch**, исключение считается обработанным.

Если подходящих обработчиков **catch** не существует, то выполнение программы переходит к следующему блоку **try**. Если до конца программы не найдены соответствующие обработчики **catch**, то программа завершает свое выполнение с ошибкой исключения.

## **Правила генерации и перехвата исключений, производные исключения.**

При наличии нескольких блоков catch вызывается только один, тот, который подходит по типу исключения и находится выше остальных по коду.

## Проброс и повторная генерация исключений, перехват любых исключений. Особенности передачи исключений по указателю, по ссылке и по значению.

**Проброс генерации исключений.** В некоторых случаях возникает необходимость поймать исключение в обработчике, а затем передать это исключение дальше в следующий обработчик (если он есть). Речь идёт о ситуации, когда в некотором внешнем блоке try написана группа внутренних блоков try/catch. Для проброса исключения из внутреннего обработчика во внешний нужно написать конструкцию throw; (без аргумента) в соответствующем внутреннем обработчике.

```
try {
    try {
        int i = 0;
        std::cin >> i;
        if (i == 0) {
            throw -101;
        }
    } catch (int code) {
        std::cout << "Got an exception for the first time. Code = " << code << std::endl;
        /* Пробрасываем исключение во внешний обработчик. */
        throw;
    }
}
catch(int code) {
    /* Ловим исключение второй раз. */
    std::cout << "The same exception. Again! Code = " << code << std::endl;
}
```

**Повторная генерация исключений.** При повторной генерации исключения используйте ключевое слово throw без указания какого-либо идентификатора.

```
void runProgram ()

{
    try {
        ManagedIntegerArray a( 10 );
        a[ rand() ] = 25;
    }
    catch ( ManagedIntegerArray::IndexOutOfRangeException & e )
    // Локальная обработка
    std::cout << "Program has a problem with array index" << std::endl;
    // Повторная генерация того же исключения
    throw;
}
```

Результат выполнения программы:

```
Caught Parent p, which is actually a Child
Caught Parent p, which is actually a Child
```

**Перехват Исключений.** Блок catch, следующий за блоком try, ловит любое исключение. Вы можете указать, какой тип исключения вы хотите перехватить, и это определяется объявлением исключения, которое появляется в круглых скобках после ключевого слова. Если нужно, чтобы блок catch обрабатывал любой тип исключения, вызванного в блоке try, нужно поставить многоточие,..., между скобками, заключающими объявление исключения следующим образом:

```
try {
// protected code
```

```
} catch(...) {  
    // code to handle any exception  
}
```

#### **Особенности передачи исключений по указателю, по ссылке и по значению.**

Если не удалять объекты-исключения, перехватываемые по указателю, то это может привести к утечкам памяти, когда на стороне выброса использовалось динамическое выделение памяти. Если удалять все объекты-исключения, можно получить фатальный сбой при попытке удаления объекта, не хранящегося в динамической памяти, такого как `globalException`.

Перехватывать объекты-исключения можно по значению либо по ссылке. Если принимать исключение по значению, будет выполняться копирование его содержимого. Если же принимать исключение по ссылке, копирования происходить не будет, обработчик будет работать с содержимым объекта-исключения в специальной служебной памяти. Обычно используют константные ссылки, чтобы избежать случайной модификации объекта-исключения.

## **Неожиданные исключения, функция `unexpected`. Функции `terminate` и `set_terminate`. Особенности исключений в конструкторах и деструкторах, функция `uncaught_exception`.**

### **Неожиданные исключения, функция `unexpected`.**

Когда не один из блоков `catch` не готов обработать возникшее исключение среда выполнения C++ делает следующее:

1. Вызывается функция `unexpected`
2. Функция `unexpected` вызывает функцию, указанную в `unexpected_handler`

Чтобы зарегистрировать `unexpected_handler` используется функция `set_unexpected(handler)`

### **Функции `terminate` и `set_terminate`.**

Функция `terminate` вызывается в следующих случаях:

1. Исключение выбрасывается и не перехватывается (определяется реализацией, выполняется ли в этом случае какая-либо раскрутка стека)
2. Функция, непосредственно вызываемая механизмом обработки исключений при обработке исключения, которое еще не было перехвачено, завершается через исключение (например, деструктор некоторого локального объекта или конструктор копирования, создающий параметр `catch-clause`)
3. Конструктор или деструктор статического или локального (начиная с C++11) объекта выдает исключение
4. Функция, зарегистрированная с помощью `std::atexit` или `std::at_quick_exit` (начиная с C++11), выдает исключение

Функция `set_terminate` устанавливает новый `terminate_handler` и возвращает предыдущий `terminate_handler`

### **Особенности исключений в конструкторах и деструкторах, функция `uncaught_exception`.**

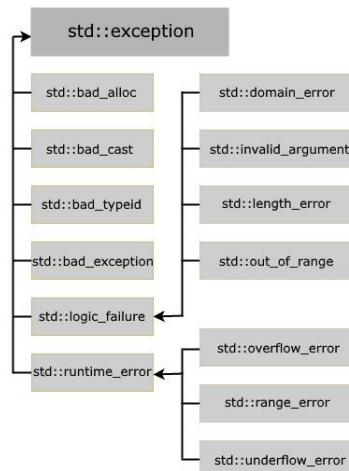
Если конструктор класса завершает работу исключением, значит он не завершает свою работу — следовательно объект не будет создан. Из-за этого могут возникать утечки памяти, т.к. для не полностью сконструированных объектов не будет вызван деструктор.

Если деструктор выбросит исключение, также произойдёт утечка памяти, в памяти останутся части текущего и базового классов.

Стандартная библиотека предоставляет функцию `std::uncaught_exception`, которая в деструкторе позволяет узнать, почему уничтожается объект, из-за выброшенного исключения, или же по какой-либо другой причине.

## Примеры стандартных исключений и операторов или методов STL, которые их генерируют. Отличия между исключениями и ошибками времени выполнения.

Примеры стандартных исключений и операторов или методов STL, которые их генерируют.



Основные из них: `runtime_error`: общий тип исключений, которые возникают во время выполнения

- `range_error`: исключение, которое возникает, когда полученный результат превосходит допустимый диапазон
- `overflow_error`: исключение, которое возникает, если полученный диапазон превышает допустимый диапазон
- `underflow_error`: исключение, которое возникает, если полученный диапазон превышает допустимый диапазон
- `domain_error`: исключение, которое возникает, если для некоторого значения, передаваемого в функцию, не определено результата
- `invalid_argument`: исключение, которое возникает при попытке создать объект большего размера, чем допустим для данного типа
- `length_error`: исключение, которое возникает при попытке доступа к элементам вне допустимого диапазона
- `out_of_range`: исключение, которое возникает при попытке доступа к элементам вне допустимого диапазона

## Отличия между исключениями и ошибками времени выполнения.

## **Понятие контейнера. Последовательные контейнеры.**

### **Контейнер std::vector, схема его реализации. Схема работы и асимптотика методов push\_back, pop\_back, insert, erase.**

**Контейнер** - это класс STL (стандартная библиотека шаблонов), реализующий функциональность некоторой структуры данных, то есть хранилища нескольких элементов. Примеры разных контейнеров: vector, stack, queue, deque, string, set, map и т.д.

Различные контейнеры имеют различные способы доступа к элементом. Для обращения к элементам контейнеров существует понятие итератора. Итератор является обобщением идеи доступа к элементу по индексу и обобщением указателей языка С. Можно рассматривать итераторы, как "умные" указатели.

**Последовательные контейнеры** обеспечивают хранение конечного количества однотипных объектов в виде непрерывной последовательности. К базовым последовательным контейнерам относятся векторы (vector), списки (list) и двусторонние очереди (deque).

**Контейнер вектор** является аналогом обычного массива, за исключением того, что он автоматически выделяет и освобождает память по мере необходимости. Контейнер эффективно обрабатывает произвольную выборку элементов с помощью операции индексации [] или метода at. Однако вставка элемента в любую позицию, кроме конца вектора, неэффективна. Для этого потребуется сдвинуть все последующие элементы путем копирования их значений. По этой же причине неэффективным является удаление любого элемента, кроме последнего.

**Асимптотика методов push\_back, pop\_back, insert, erase:** (**Временная Сложность**) **push\_back()** - для добавления элементов в вектор, в который передается добавляемый элемент - $O(1)$  **pop\_back()** - удаляет последний элемент вектора - $O(1)$  **insert(pos, value)** - вставляет элемент value на позицию, на которую указывает итератор pos, аналогично функции **emplace insert(pos, n, value)** - вставляет n элементов value начиная с позиции, на которую указывает итератор pos **insert(pos, begin, end)** - вставляет начиная с позиции, на которую указывает итератор pos, элементы из другого контейнера из диапазона между итераторами begin и end **insert(pos, values)** - вставляет список значений начиная с позиции, на которую указывает итератор pos **erase(p)** - удаляет элемент, на который указывает итератор p. Возвращает итератор на элемент, следующий после удаленного, или на конец контейнера, если удален последний элемент - $O(n)$  **erase(begin, end)** - удаляет элементы из диапазона, на начало и конец которого указывают итераторы begin и end. Возвращает итератор на элемент, следующий после последнего удаленного, или на конец контейнера, если удален последний элемент

## **Понятие контейнера. Последовательные контейнеры.**

### **Контейнер std::vector, схема его реализации. Методы size, capacity, resize, reserve, shrink\_to\_fit.**

**Понятие контейнера.**

**Контейнерный класс** (или «**класс-контейнер**») в языке C++ — это класс, предназначенный для хранения и организации нескольких объектов определенного типа данных (пользовательских или фундаментальных).

### **Последовательные контейнеры.**

**Последовательные контейнеры** (или «**контейнеры последовательности**») — это контейнерные классы, элементы которых находятся в последовательности. Их определяющей характеристикой является то, что вы можете добавить свой элемент в любое место контейнера. Наиболее распространенным примером последовательного контейнера является **массив**

### **Контейнер std::vector, схема его реализации.**

**std::vector** (или просто «**вектор**») — это тот же динамический массив, но который может сам управлять выделенной себе памятью. Это означает, что вы можете создавать массивы, длина которых задается во время выполнения, без использования **операторов new и delete**

#### **Методы size, capacity, resize, reserve,**

**shrink\_to\_fit.** **size** - Возвращает количество элементов в векторе.

**capacity** - Возвращает количество элементов, которое может содержать вектор до того, как ему потребуется выделить больше места. **resize** - Изменяет размер вектора на заданную величину. **reserve** - Устанавливает минимально возможное количество элементов в векторе. **shrink\_to\_fit** - Уменьшает количество используемой памяти за счёт освобождения неиспользованной

### **Понятие контейнера. Последовательные контейнеры.**

### **Контейнер std::vector, схема его реализации. Методы для обращения по индексу, для обращения к первому и последнему элементам. Метод swap, его схема реализации.**

#### **Понятие контейнера.**

**Контейнерный класс** (или «**класс-контейнер**») в языке C++ — это класс, предназначенный для хранения и организации нескольких объектов определенного типа данных (пользовательских или фундаментальных).

### **Последовательные контейнеры.**

**Последовательные контейнеры** (или «**контейнеры последовательности**») — это контейнерные классы, элементы которых находятся в последовательности. Их определяющей характеристикой является то, что вы можете добавить свой элемент в любое место контейнера. Наиболее распространенным примером последовательного контейнера является **массив**

### **Контейнер std::vector, схема его реализации.**

**std::vector** (или просто «**вектор**») — это тот же динамический массив, но который может сам управлять выделенной себе памятью. Это означает, что вы можете создавать массивы, длина которых задается во время выполнения, без использования **операторов new и delete**

**Методы для обращения по индексу, для обращения к первому и последнему элементам.**

Доступ к элементам	vector::at	Доступ к элементу с проверкой выхода за границу
	vector::operator[]	Доступ к определённому элементу
	vector::front	Доступ к первому элементу
	vector::back	Доступ к последнему элементу

### Метод swap, его схема реализации.

Метод swap - Обменять содержимое двух векторов

## Контейнер `vector<bool>` и его отличия от обычного `vector`.

`std::vector<bool>` — это более компактная специализация `std::vector` для типа `bool`.

Способ, которым `std::vector<bool>` экономит пространство (а также то, оптимизируется ли он вообще), определяется реализацией. Одна потенциальная оптимизация включает в себя объединение векторных элементов таким образом, чтобы каждый элемент занимал один бит вместо `sizeof(bool)` байтов.

`std::vector<bool>` ведет себя аналогично `std::vector`, но для экономии места он:

1. Не обязательно сохраняет свои элементы как непрерывный массив.
2. Предоставляет класс `std::vector<bool>::reference` как метод доступа к отдельным битам. В частности, объекты этого класса возвращаются оператором [] по значению.
3. Не использует `std::allocator_traits::construct` для построения битовых значений.

## Контейнер `std::deque`, схема его реализации, асимптотика работы методов. Отличия `deque` от `vector` с точки зрения интерфейса. Контейнеры `std::list` и `std::forward_list`, схема их реализации, асимптотика работы методов. Отличия `list` от `deque` и `vector` с точки зрения интерфейса.

Контейнер двусторонняя очередь во многом аналогичен вектору, элементы хранятся в непрерывной области памяти, элементы можно добавлять и удалять как в начало, так и в конец, то есть дисциплинами обслуживания являются одновременно FIFO и LIFO. Но в отличие от вектора двусторонняя очередь эффективно поддерживает вставку и удаление первого элемента (так же, как и последнего).

`push_back(T&key)` - добавление в конец

(O(1)) `pop_back()` - удаление(O(1))

`push_front(T&key)` - добавление в начало

(O(1)) `pop_front()` - удаление из начала

(O(1)) `insert` - вставка в произвольное

место

(`insert(iter, val)`) - O(1)

`insert(iter, ne, val)` - O(n)

insert(iter, initA, initB) -

$O(n)$

erase - удаление из произвольного места [], at( $O(1)$ ) - доступ к произвольному элементу

front(): возвращает первый элемент -  $O(1)$  back(): возвращает последний элемент -  $O(1)$

assign(n, value) - заменяет содержимое контейнера n элементами, которые имеют

значение value emplace\_back(val)/emplace\_front(val) - добавляет значение val в конец

очереди и в начало очереди - $O(1)$

**Контейнер список** организует хранение объектов в виде двусвязного списка. Каждый элемент списка содержит три поля: значение элемента, указатель на предшествующий и указатель на последующий элементы списка. Вставка и удаление работают эффективно для любой позиции элемента в списке. Однако список не поддерживает произвольного доступа к своим элементам: например, для выборки n-го элемента нужно последовательно выбрать предыдущие n-1 элементов. методы такие же как и в двусторонней очереди (нету [] и at) +

L.emplace_front(val)	$O(1)$		Constructs and insert element at the beginning of the list.
L.emplace_back(val)	$O(1)$		Constructs and insert element at the end of the list.
L.push_front(val)	$O(1)$		Insert element at the beginning of the list.
L.push_back(val)	$O(1)$		Insert element at the end of the list.
L.pop_front()	$O(1)$		Delete element at the beginning of the list.
L.pop_back()	$O(1)$		Delete element at the end of the list.
L.insert(iterator, val)	$O(1)$		Insert element at the specified position.
L.erase(iterator)	$O(1)$		Delete element from specified position.
L.clear()	$O(n)$		Erase all the elements from list.

Контейнер **forward\_list** реализует односвязный список, элементы которого хранят указатель только на следующий элемент. Вставка и удаление элементов происходит быстро, достаточно изменить ссылку, но получить доступ к элементу списка по индексу нельзя, т. к. элементы могут быть расположены в разных местах памяти. Перебирать элементы можно только в прямом направлении. **resize(n)** - оставляет в списке n первых элементов assign() ; push\_front (); emplace(); insert(); clear(); swap()

## Методы, специфичные для list: sort, merge, splice, remove, unique, reverse, схема их реализации. Отличия list от forward\_list

sort() - сортирует список merge (list &ob) - объединяет упорядоченный список, содержащийся в объекте ob, с данным упорядоченным списком.

splice(iterator i, list &ob) - вставляет содержимое списка ob в данный список в позиции, указанной итератором i. После выполнения этой операции список ob остается пустым splice(iterator i, list &ob,

`iterator el`) - удаляет из списка `ob` элемент, адресуемый итератором `el`, и сохраняет его в данном списке в позиции, адресуемой итератором `i` `remove(val)` - удаляет из списка элементы со значением `val` `unique()` - удаляет из списка элементы-дубликаты `reverse()` - обратный список

List основан на двусвязном списке, а `forward_list` основан на односвязном списке (нет возможности получить доступ к отдельному методу нельзя повторить в обратном порядке).

## **Понятие адаптеров над контейнерами. Схема реализации std::stack, std::queue, std::priority\_queue, асимптотика работы методов, требования к используемому контейнеру.**

### **Понятие адаптеров над контейнерами.**

**Адаптеры** — это специальные предопределенные контейнерные классы, которые адаптированы для выполнения конкретных заданий. Самое интересное заключается в том, что вы сами можете выбрать, какой последовательный контейнер должен использовать адаптер.

### **Схема реализации std::stack, std::queue, std::priority\_queue, асимптотика работы методов, требования к используемому контейнеру.**

#### **stack**

Методы:

1. **empty()** – Returns whether the stack is empty – Time Complexity : O(1)
2. **size()** – Returns the size of the stack – Time Complexity : O(1)
3. **top()** – Returns a reference to the top most element of the stack – Time Complexity : O(1)
4. **push(g)** – Adds the element ‘g’ at the top of the stack – Time Complexity : O(1)
5. **pop()** – Deletes the top most element of the stack – Time Complexity : O(1)

#### **queue**

Методы:

1. **empty()** - Returns whether the queue is empty. It return true if the queue is empty otherwise returns false.  
- Time Complexity : O(1)
2. **size()** - Returns the size of the queue. - Time Complexity : O(1)
3. **emplace()** - Exchange the contents of two queues but the queues must be of the same data type, although sizes may differ. - Time Complexity : O(1)
4. **front()** - Insert a new element into the queue container, the new element is added to the end of the queue. - Time Complexity : O(1)
5. **back()** - Returns a reference to the first element of the queue. - Time Complexity : O(1)
6. **push(g)** - Adds the element ‘g’ at the end of the queue. - Time Complexity : O(1)
7. **pop()** - Deletes the first element of the queue. - Time Complexity : O(1) **priority\_queue**

Методы:

1. **empty()** - Returns true if the priority queue is empty and false if the priority queue has at least one element. Its time complexity is O(1).

2. **pop()** - Removes the largest element from the priority queue. Its time complexity is  $O(\log N)$  where  $N$  is the size of the priority queue.
3. **push()** - Inserts a new element in the priority queue. Its time complexity is  $O(\log N)$  where  $N$  is the size of the priority queue.
4. **size()** - Returns the number of element in the priority queue. Its time complexity is  $O(1)$ .
5. **top()** - Returns a reference to the largest element in the priority queue. Its time complexity is  $O(1)$ .

## Ассоциативные контейнеры. Класс std::map, его идея, его назначение и схема реализации. Асимптотика и принцип работы методов [], find, count, insert, erase. Класс std::pair и его связь с map. Функция make\_pair.

Ассоциативные контейнеры обеспечивают быстрый доступ к данным по ключу. Эти контейнеры построены на основе сбалансированных деревьев. Существует пять типов ассоциативных контейнеров: словари (map), словари с дубликатами (multimap), множества (set), множества с дубликатами (multiset) и битовые множества (bitset).

**map** (или «**ассоциативный массив**») — это set, в котором каждый элемент является парой «ключ-значение». «Ключ» используется для сортировки и индексации данных и должен быть уникальным, а «значение» — это фактические данные. Позволяет записать значение по некоторому ключу, просмотреть значение по ключу, а также удалить пару с заданным ключом. Ключи в этом контейнере являются уникальными, их нельзя модифицировать

Благодаря упорядоченной структуре контейнера **map** операции поиска или внесения нужного элемента могут быть выполнены за логарифмическое время  $O(\log(n))$ .

**count( )**: выполняет поиск элементов, сопоставленных по заданному ключу, и возвращает количество совпадений. Поскольку map хранит каждый элемент с уникальным ключом, то вернет 1, если совпадение найдено, в противном случае вернет 0.  $O(\log N)$ .

**find(k)** - возвращает итератор, указывающий на значение, соответствующее значению ключа **k**. Если такого значения в контейнере нет, то возвращается итератор **end**. **insert(pair(k,v))** - вставляет в контейнер пару **(k, v)**, возвращая адрес его позиции **erase(p)** - удаляет из контейнера элемент, на который указывает итератор **p**

**Класс pair** (пара) стандартной библиотеки C++ позволяет нам определить одним объектом пару значений, если между ними есть какая-либо семантическая связь. Эти значения могут быть одинакового или разного типа. Первое значение доступно через свойство **first**, а второе значение — через свойство **second**. **pair** - одна пара ключ  $\rightarrow$  значений, **map** - коллекция пар ключ  $\rightarrow$  значений

Шаблонной функцией **make\_pair()**

```
typedef struct std::pair<int, int> PAIR;
PAIR pr = std::make_pair(5, 10);
std::cout << pr.first << std::endl; // 5
std::cout << pr.second << std::endl; // 10
```

## **Ассоциативные контейнеры. Класс std::set, его отличия от std::map. Классы std::multimap и std::multiset, их отличия от std::map и std::set.**

Ассоциативные контейнеры обеспечивают быстрый доступ к данным по ключу. Эти контейнеры построены на основе сбалансированных деревьев.

**set** — это контейнер, в котором хранятся только уникальные элементы, и повторения запрещены. Элементы сортируются в соответствии с их значениями.

**multiset** — это set, но в котором допускаются повторяющиеся элементы.

**map** (или «**ассоциативный массив**») — это set, в котором каждый элемент является парой «ключ-значение». «Ключ» используется для сортировки и индексации данных и должен быть уникальным, а «значение» — это фактические данные.

**multimap** (или «**словарь**») — это map, который допускает дублирование ключей. Все ключи отсортированы в порядке возрастания, и вы можете посмотреть значение по ключу.

## **Класс std::unordered\_map, его идея, схема реализации, асимптотика работы методов, отличия от std::map.**

**unordered\_map** — это связанный контейнер, в котором хранятся элементы, образованные комбинацией ключ-значение и отображаемое значение. Значение ключа используется для уникальной идентификации элемента, а сопоставленное значение — это содержимое, связанное с ключом. И ключ, и значение могут быть любого типа, предопределеными или определенными пользователем.

map представляет собой упорядоченную последовательность уникальных ключей, тогда как в unordered\_map ключ может храниться в любом порядке.

## **Стандартные строки. Шаблонный класс basic\_string, его методы, отличия от vector. Класс string. Класс char\_traits, его специализации, его назначение.**

Класс **basic\_string** по сути является контейнером. Это значит, что итераторы и алгоритмы STL могут обеспечить работу со строками. Однако строки обладают дополнительными возможностями.

Тип **basic\_string** использует класс **char\_traits**, который определяет ряд атрибутов символов, составляющих строку.

Отличия от **vector**:

1. **basic\_string** не вызывает конструкторы и деструкторы своих элементов. **vector** вызывает.
2. Замена **basic\_string** делает недействительными итераторы (включая оптимизацию небольших строк), замена векторов - нет.
3. Символоподобные объекты в объекте **basic\_string** должны храниться непрерывно.
4. **basic\_string** имеет интерфейс для строковых операций. **vector** нет.
5. **basic\_string** может использовать стратегию копирования при записи. **vector** не может.

Класс **string** был введен как альтернативный вариант для работы со строками типа **char\***. Строки, которые завершаются символом '**\0**' еще называются С-строками. Поскольку, **string** есть классом, то можно объявлять объекты этого класса.

Основным недостатком типа `string` в сравнении с типом `char*`, есть замедленная скорость обработки данных.

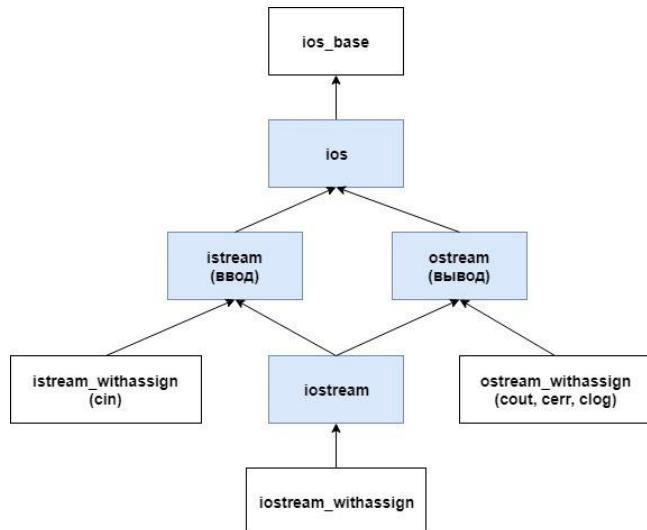
**Класс `char_traits`** - это шаблон класса `traits`, который абстрагирует основные символьные и строковые операции для данного типа символов. Определенный набор операций таков, что универсальные алгоритмы почти всегда могут быть реализованы с его точки зрения.

## Концепция потоков ввода-вывода. Иерархия классов стандартных потоков. Класс `ios_base`. Флаги потоков. Методы `precision` и `width`. Синхронизация с `stdio`.

**Поток** — это последовательность символов, к которым можно получить доступ.

**Поток ввода** (или «**входной поток**») используется для хранения данных, полученных от источника данных: клавиатуры, файла, сети и т.д.

**Поток вывода** (или «**выходной поток**») используется для хранения данных, предоставляемых конкретному потребителю данных: монитору, файлу, принтеру и т.д. При записи данных на устройство вывода, это устройство может быть не готовым принять данные немедленно.



**ios\_base**. Класс описывает хранилище и функции-члены, общие как для входных, так и для выходных потоков, которые не зависят от параметров шаблона.

<i>ios_base::scientific</i>	Устанавливается для вывода чисел с плавающей запятой в экспоненциальном формате.
<i>ios_base::showpos</i>	Указывает, что положительным и отрицательным числам предшествовать знаки "плюс" и "минус" соответственно.
<i>ios_base::showpoint</i>	Указывает, что числа с плавающей запятой должны выводиться с десятичной точкой. Обычно используют совместно с флагом <i>ios_base::fixed</i> .
<i>ios_base::uppercase</i>	Указывает, что прописная буква "E" должна использоваться при представлении значения с плавающей запятой в экспоненциальном формате. Также предполагает использование прописных букв в шестнадцатеричном целом.
<i>ios_base::oct</i>	Устанавливает восьмеричный формат представления данных (с основанием 8).
<i>ios_base::dec</i>	Устанавливает десятичный формат представления данных (с основанием 10).
<i>ios_base::hex</i>	Устанавливает шестнадцатеричный формат представления данных (с основанием 16).
<i>ios_base::showbase</i>	Указывает, что основание числа должно выводиться перед числом: для восьмеричных чисел выводится начальный символ "0", десятичные числа выводятся обычным способом, шестнадцатеричные числа выводятся с индикатором "0x" или "OX".
<i>ios_base::adjustfield</i>	Обычно задается в качестве второго параметра функции-члена <i>setf()</i> , если устанавливаются флаги <i>left</i> , <i>right</i> или <i>internal</i> (это гарантирует, что функция <i>setf()</i> установит только один из трех флагов).
<i>ios_base::unitbuf</i>	Позволяет задать очистку буфера после каждой операции вывода.

Допускается объединение разных флагов в одно значение (типа *long*) путем использования операции побитовое ИЛИ  
`(|)`; можно также использовать операцию "запятая **precision** - Позволяет управлять точностью печатаемых чисел с плавающей запятой **width** -

#### Установка ширины поля

Синхронизация с `stdin` можно включить или отключить используя функцию `sync_with_stdio(bool)`. Устанавливает, синхронизируются ли стандартные потоки C++ со стандартными потоками C после каждой операции ввода/вывода.

## Классы **basic\_ios** и **ios**. Состояние потоков, проверка и изменение состояния. Классы **basic\_istream**, **basic\_iostream**, **basic\_oiostream**.

Шаблон класса **basic\_ios<>**, производный от `ios_base<>`, определяет общие свойства всех потоковых классов, зависящие от типа и трактовок символов (определяют признак конца файла и способы копирования/перемещения ряда символов). В число этих свойств входит определение буфера, используемого потоком данных.

Стандартная библиотека потокового ввода-вывода для языка C++ ориентирована, прежде всего, на следующую схему работы с потоком ввода: «читать данные из потока, пока очередная операция чтения не закончится неудачей». Поэтому никакая ошибка, связанная с потоками, не приводит к аварийному завершению программы. Вместо этого изменяется состояние соответствующего потока, после чего все операции, связанные с этим потоком, блокируются.

Для обнаружения ошибки программа должна явным образом проверять состояние потока, а для продолжения работы с потоком (если это возможно) — сбрасывать состояние ошибки.

Средства, предусмотренные для проверки и изменения состояния потока; все эти средства реализованы в классе `ios`

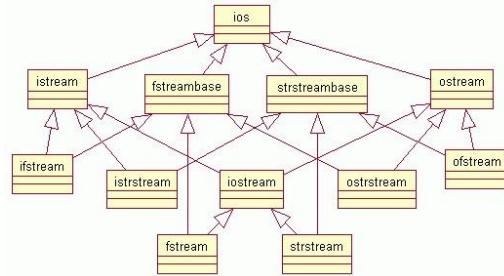
(точнее, в классе-шаблоне `basic_ios`) и поэтому

доступны для всех рассматриваемых далее потоков:

- 1) `good`: в данный момент ошибок, связанных с потоком, не обнаружено;

- 2) bad: при работе с потоком произошла фатальная ошибка, которую, скорее всего, не удастся исправить;
- 3) eof: прочесть очередной элемент данных не удалось, так как обнаружен конец потока;
- 4) fail: произошла какая-то ошибка; возможно, это фатальная ошибка (тогда значение true вернет и функция bad), возможно, это ошибка, связанная с концом потока (тогда значение true вернет и функция eof), возможно, это другая ошибка ввода-вывода или преобразования данных (в этой ситуации можно попытаться продолжить работу с потоком, сбросив состояние ошибки).

Класс **ios** содержит средства для форматированного ввода-вывода и проверки ошибок.



Стандартные потоки (istream, ostream, iostream) служат для работы с терминалом. Строковые потоки (istrstream, ostrstream, strstream) служат для ввода-вывода из строковых буферов, размещенных в памяти. Файловые потоки (ifstream, ofstream, fstream) служат для работы с файлами. ios (базовый потоковый класс), streambuf (буферизация потоков), istream (потоки ввода), ostream (потоки вывода), iostream (дву направленные потоки), istrstream (строковые потоки ввода), ostrstream (строковые потоки вывода), strstream (дву направленные строковые потоки), ifstream (файловые потоки ввода), ofstream (файловые потоки вывода), fstream (дву направленные файловые потоки).

- Потоковые классы делятся на три группы (шаблонов классов):
  - basic\_istream, basic\_ostream - общие потоковые классы, которые могут быть связаны с любым буферным объектом; basic\_iostream - потоковые классы для считывания и записи файлов; basic\_istringstream, basic\_ostringstream - потоковые классы для объектов-строк.

## Классы ostream, istream, iostream, их определения. Манипуляторы потоков, схема их работы. Примеры манипуляторов. Манипуляторы с параметрами, заголовочный файл <iomanip>. Файловые потоки и строковые потоки.

- ifstream, производный от istream, связывает ввод программы с файлом;
- ofstream, производный от ostream, связывает вывод программы с файлом;
- fstream, производный от iostream, связывает как ввод, так и вывод программы с файлом.

Манипулятор потока	Описание, пример использования
<code>setw</code>	Установка ширины поля: <code>cout&lt;Name</code> Выходная величина <code>Name</code> будет напечатана с шириной поля 5, т.е. ее значение будет содержать, по крайней мере, 5 символьных позиций. Если длина выходной величины менее 5 символов, она будет выровнена в поле по правому символу; в противном случае размер поля будет увеличен, чтобы вместить всю величину.
<code>width</code>	Установка ширины поля: <code>cout&lt;Name</code> Если величина <code>Name</code> имеет меньше символов, чем заданная ширина поля 5, то для заполнения лишних позиций будут использованы заполняющие символы; в противном случае размер поля будет увеличен, чтобы вместить всю величину. Для этих же целей можно использовать функцию-член <code>width()</code> : <pre>cout.width(10); cout&lt;</pre> <i>Метод <code>width()</code> касается только следующего отображаемого элемента, после чего ширина поля вернется к значению по умолчанию.</i>
<code>setprecision</code>	Позволяет управлять точностью печатаемых чисел с плавающей запятой (числом разрядов справа от десятичной точки): <code>cout&lt;ch</code> ; Действует для всех последующих операций вывода, пока не будет произведена другая установка точности. С тем же успехом можно использовать функцию-член <code>precision()</code> : <pre>cout.precision(4); cout&lt;&lt;ch;</pre>
<code>dec</code>	Устанавливает десятичный формат представления данных (с основанием 10): <code>cout&lt;ch</code> ;
<code>oct</code>	Устанавливает восьмеричный формат представления данных (с основанием 8): <code>cout&lt;ch</code> ;
<code>hex</code>	Устанавливает шестнадцатеричный формат представления данных (с основанием 16): <code>cout&lt;ch</code> ;
<code>setbase</code>	Изменяет основание потока, принимая какой-либо из целых параметров 8, 10 или 16, задающих основание системы счисления: <code>cout&lt;ch</code> ;

Есть три основных класса файлового ввода/вывода в языке C++:

**`ifstream`** (является дочерним классу **`istream`**);

**`ofstream`** (является дочерним классу **`ostream`**);

**`fstream`** (является дочерним классу **`iostream`**).

**Строковые потоки** — это универсальные средства перевода внутренних двоичных значений в строки (выходной строковый поток) и обратно (входной строковый поток).

можно в качестве объектов объявлять строковые потоки трех видов:

- входной `istringstream`
- выходной `ostringstream`
- двунаправленный

`stringstream`

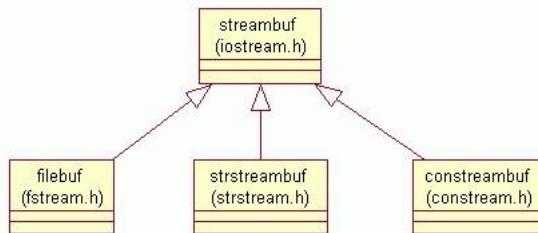
## Буфера потоков, классы **`basic_streambuf`** и **`streambuf`**. Итераторы потоков: **`istream_iterator`** и **`ostream_iterator`**, их схема реализации.

**Буфера потоков.** Потоки выполняют обмен данными между прикладной программой и файлом через буфера, которые являются объектами классов `filebuf` и `stringbuf` соответственно. Эти классы являются производными от абстрактного класса `streambuf`. Каждый буфер содержит в зависимости от режима доступа один или два символьных массива: один для ввода, а второй для вывода данных. Каждый

поток поддерживает внутренний указатель на буфер. Ввод-вывод данных из буфера в файл или в строку называется *синхронизацией буфера с внешним устройством ввода-вывода*.

**Классы basic\_streambuf и streambuf.** Вся система ввода-вывода C++ построена на двух связанных, но различных иерархиях шаблонных классов. Первая выведена из класса ввода-вывода нижнего уровня basic\_streambuf. Этот класс поддерживает базовые операции ввода и вывода нижнего уровня и обеспечивает основополагающую поддержку для всей системы ввода-вывода C++.

Класс **streambuf** обеспечивает организацию и взаимосвязь буферов ввода-вывода, размещаемых в памяти, с физическими устройствами ввода-вывода. Методы и данные класса **streambuf** программист явно обычно не использует. Этот класс нужен другим классам библиотеки ввода-вывода. Он доступен и программисту для создания новых классов на основе уже существующих.



#### Итераторы потоков: istream\_iterator и ostream\_iterator, их схема реализации.

Стандартные потоковые итераторы **istream\_iterator<T>** и **ostream\_iterator<T>** (шаблонные классы) определены в заголовочном файле **<iostream>**.

Имеются два варианта конструктора для итератора потокового чтения **istream\_iterator**: вариант с параметром-потоком **stream** создает итератор для чтения из данного потока, вариант без параметров создает итератор, обозначающий конец потока.

Ниже перечислены свойства потоковых итераторов чтения:

- 1) тип **T** определяет тип элементов данных, которыечитываются из потока;
- 2) чтение элемента из потока выполняется в начальный момент работы с итератором, а затем при каждой операции инкремента **++**;
- 3) имеются два варианта операции **++**: префиксный инкремент (**++p**) и постфиксный инкремент (**p++**);
- 4) при достижении конца потока итератор становится равным итератору конца потока;

Для итератора потоковой записи **ostream\_iterator<T>** также определены два конструктора: первый конструктор содержит единственный параметр **stream**, задающий поток вывода, а второй конструктор дополнительно к параметру **stream** содержит второй параметр **delim**, задающий разделитель, который добавляется в поток вывода после каждого выведенного элемента (если параметр **delim** не указан, то между выводимыми элементами никакой разделитель не добавляется).

Ниже перечислены свойства потоковых итераторов записи:

- 1) специальный конструктор для создания итератора конца потока вывода не предусмотрен;
- 2) операции **\*** и **++** не выполняют никаких действий и просто возвращают сам итератор;
- 3) операция присваивания **p = выражение** (где **p** – имя итератора записи) записывает значение выражения в поток вывода.