# Еще о классах

Спасибо CSCenter

# Основные операторы

## Арифметические

- Унарные: префиксные `+ - ++ --`, постфиксные `++ --`
- Бинарные: `+ - * / % += -= *= /= %=`

## Битовые

- Унарные: `~`.
- Бинарные: `& | ^ &= |= ^= >> <<`.

## Логические

- Унарные: `!`.
- Бинарные: `&& ||`.
- Сравнения: `== != > < >= <=`

# Другие операторы

1. Оператор присваивания: `=`
2. Специальные:
   - префиксные `* &`,
   - постфиксные `-> ->*`,
   - особые `, . ::`
3. Скобки: `[] ()`
4. Оператор приведения `(type)`
5. Тернарный оператор: `x ? y : z`
6. Работа с памятью: `new new[] delete delete[]`

Нельзя перегружать операторы `. ::` и тернарный оператор.

# Перегрузка операторов

```
Vector operator-(Vector const& v) {
    return Vector(-v.x, -v.y);
}

Vector operator+(Vector const& v,
                 Vector const& w) {
    return Vector(v.x + w.x, v.y + w.y);
}

Vector operator*(Vector const& v, double d) {
    return Vector(v.x * d, v.y * d);
}

Vector operator*(double d, Vector const& v) {
    return v * d;
}
```

# Перегрузка операторов внутри классов

**NB:** Обязательно для `(type) [] () -> ->* =`

```
struct Vector {
    Vector operator-() const { return Vector(-x, -y); }
    Vector operator-(Vector const& p) const {
        return Vector(x - p.x, y - p.y);
    }
    Vector & operator*=(double d) {
        x *= d;
        y *= d;
        return *this;
    }
    double operator[](size_t i) const {
        return (i == 0) ? x : y;
    }
    bool   operator()(double d)   const   { ... }
    void   operator()(double a, double b) { ... }
    double x, y;
};
```

# Перегрузка инкремента и декремента

```
struct BigNum {
    BigNum & operator++() { //prefix
        //increment
        ...
        return *this;
    }

    BigNum operator++(int) { //postfix
        BigNum tmp(*this);
        ++(*this);
        return tmp;
    }
    ...
};
```

# Переопределение операторов ввода-вывода

```
#include <iostream>

struct Vector { ... };

std::istream& operator>>(std::istream & is,
                         Vector & p) {
    is >> p.x >> p.y;
    return is;
}

std::ostream& operator<<(std::ostream &os,
                         Vector const& p) {
    os << p.x << ' ' <<  p.y;
    return os;
}
```

# Умный указатель

Реализует принцип: *“Получение ресурса есть инициализация”*
*Resource Acquisition Is Initialization (RAII)*

```
struct SmartPtr {
    Data & operator*()  const {return *data_;}
    Data * operator->() const {return  data_;}
    Data * get()        const {return  data_;}
    ...
private:
    Data * data_;
};

bool operator==(SmartPtr const& p1,
                SmartPtr const& p2) {
    return p1.get() == p2.get();
}
```

# Оператор приведения

```
struct String {
    operator bool() const {
        return size_ != 0;
    }

    operator char const *() const {
        if (*this)
            return data_;
        return "";
    }

private:
    char * data_;
    size_t size_;
};
```

# Операторы с особым порядком вычисления

```
int main() {
    int a = 0;
    int b = 5;
    (a != 0) && (b = b / a);
    (a == 0) || (b = b / a);

    foo() && bar();
    foo() || bar();
    foo(), bar();
}

// no lazy semantics
Tribool operator&&(Tribool const& b1,
                   Tribool const& b2) {
    ...
}
```

# Переопределение арифметических и битовых операторов

```
struct String {
    String( char const * cstr ) { ... }

    String & operator+=(String const& s) {
        ...
        return *this;
    }
    //String operator+(String const& s2) const {...}
};

String operator+(String s1, String const& s2) {
    return s1 += s2;
}
```

```
    String s1("world");
    String s2 = "Hello " + s1;
```

# “Правильное” переопределение операторов сравнения

```
bool operator==(String const& a, String const& b) {
    return ...
}
bool operator!=(String const& a, String const& b) {
    return !(a == b);
}
bool operator<(String const& a, String const& b) {
    return ...
}
bool operator>(String const& a, String const& b) {
    return b < a;
}
bool operator<=(String const& a, String const& b) {
    return !(b < a);
}
bool operator>=(String const& a, String const& b) {
    return !(a < b);
}
```

# О чём стоит помнить

- Стандартная семантика операторов.

```
void operator+(A const & a, A const& b) {}
```

- Приоритет операторов.

```
    Vector a, b, c;
    c = a + a ^ b * a; //?????
```

- Хотя бы один из параметров должен быть пользовательским.

```
    void operator*(double d, int i) {}
```

# Глобальные переменные

Объявление глобальной переменной:

```
extern int global;

void f () {
    ++global;
}
```

Определение глобальной переменной:

```
int global = 10;
```

Проблемы глобальных переменных:

- Масштабируемость.
- Побочные эффекты.
- Порядок инициализации.

# Статические глобальные переменные

Статическая глобальная переменная — это глобальная переменная, доступная только в пределах модуля.

Определение:

```
static int global = 10;

void f () {
    ++global;
}
```

Проблемы статических глобальных переменных:

- Масштабируемость.
- Побочные эффекты.

# Статические локальные переменные

Статическая локальная переменная — это глобальная переменная, доступная только в пределах функции.

Время жизни такой переменной — от первого вызова функции `next` до конца программы.

```
int next(int start = 0) {
    static int k = start;
    return k++;
}
```

Проблемы статических локальных переменных:

- Масштабируемость.
- Побочные эффекты.

# Статические функции

Статическая функция, доступная только в пределах модуля.

Файл 1.cpp:

```
static void test() {
    cout << "A\n";
}
```

Файл 2.cpp:

```
static void test() {
    cout << "B\n";
}
```

Статические глобальные переменные и статические функции проходят *внутреннюю линковку*.

# Статические поля класса

Статические поля класса — это глобальные переменные, определённые внутри класса.

Объявление:

```
struct User {
    ...
private:
    static size_t instances_;
};
```

Определение:

```
size_t User::instances_ = 0;
```

Для доступа к статическим полям не нужен объект.

# Статические методы

Статические методы — это функции, определённые внутри класса и имеющие доступ к закрытым полям и методам.

Объявление:

```
struct User {
    ...
    static size_t count() { return instances_; }
private:
    static size_t instances_;
};
```

Для вызова статических методов не нужен объект.

```
    cout << User::count();
```

# Ключевое слово `inline`

Советует компилятору встроить данную функцию.

```
inline double square(double x) { return x * x; }
```

- В месте вызова `inline`-функции должно быть известно её определение.
- `inline` функции можно определять в заголовочных файлах.
- Все методы, определённые внутри класса, являются `inline`.
- При линковке из всех версий `inline`-функции (т.е. её код из разных единиц трансляции) выбирается только одна.
- Все определения одной и той же `inline`-функции должны быть идентичными.
- `inline` — это совет компилятору, а не указ.

# Правило одного определения

Правило одного определения
(One Definition Rule, ODR)

- В пределах любой единицы трансляции сущности не могут иметь более одного определения.
- В пределах программы глобальные переменные и не-`inline` функции не могут иметь больше одного определения.
- Классы и `inline` функции могут определяться в более чем одной единице трансляции, но определения обязаны совпадать.

Вопрос: к каким проблемам могут привести разные определения одного класса в разных частях программы?

# Дружественные классы

```
struct String {
    ...
    friend struct StringBuffer;
private:
    char * data_;
    size_t len_;
};

struct StringBuffer {
    void append(String const& s) {
        append(s.data_);
    }
    void append(char const* s) {...}
    ...
};
```

# Дружественные функции

Дружественные функции можно определять прямо внутри описания класса (они становятся `inline`).

```
struct String {
    ...
    friend std::ostream&
        operator<<(std::ostream & os,
                   String  const& s)
    {
        return os << s.data_;
    }

private:
    char * data_;
    size_t len_;
};
```

# Дружественные методы

```
struct String;
struct StringBuffer {
    void append(String const& s);
    void append(char const* s) {...}
    ...
};

struct String {
    ...
    friend
      void StringBuffer::append(String const& s);
};

void StringBuffer::append(String const& s) {
    append(s.data_);
}
```

# Отношение дружбы

Отношение дружбы можно охарактеризовать следующими утверждениями:

- Отношение дружбы не симметрично.
- Отношение дружбы не транзитивно.
- Отношение наследования не задаёт отношение дружбы.
- Отношение дружбы сильнее, чем отношение наследования.

## Вывод

Стоит избегать ключевого слова `friend`, так как оно нарушает инкапсуляцию.

# Класс Singleton

```
struct Singleton {
    static Singleton & instance() {
        static Singleton s;
        return s;
    }

    Data & data() { return data_; }

private:
    Singleton() {}

    Singleton(Singleton const&);
    Singleton& operator=(Singleton const&);

    Data data_;
};
```

# Использование Singleton-а

```
int main()
{
    // первое обращение
    Singleton & s = Singleton::instance();
    Data d = s.data();

    // аналогично d = s.data();
    d = Singleton::instance().data();
    return 0;
}
```