Алгоритмизация и программирование

Лекция 8

enum (перечисления)

```
void set_color(/*какой тип?*/ color){
    /* Код */
}

void set_color(int color){
    /* Код */
}

void set_color(std::string color){
    /* Код */
}
```

enum

```
enum Color
{
    // Список перечислителей
    color_black,
    color_red,
    color_blue,
    color_green,
    color_white,
    color_cyan,
    color_yellow,
    color_magenta, //<- запятую можно оставить
}; // <- точкой с запятой

// Определяем несколько переменных перечислимого типа Color
Color paint = color_white;
Color house(color_blue);
Color apple { color_red };
```

enum

```
void set_color(Color color){
    /* Код */
}

set_color(color_red); // нормально
set_color(1);         // ошибка
```

enum

```
enum Color
{
  red,
  blue, // blue помещается в глобальное пространство имен
  green
};

enum Feeling
{
  happy,
  tired,
  blue // ошибка, blue уже использовался в enum Color в глобальном пространстве имен
};
```

enum

```
// определяем новое перечисление с именем Animal
enum Animal
{
    animal_cat = -3,
    animal_dog, // присвоено -2
    animal_pig, // присвоено -1
    animal_horse = 5,
    animal_giraffe = 5, // имеет то же значение, что и animal_horse
    animal_chicken // присвоено 6
};

int pet = animal_pig;
Animal mypet = animal_pig;

std::cout << mypet; // перед передачей в std::cout вычисляется как int
std::cin >> mypet;  // вызовет ошибку компилятора
```

enum

```
// Использовать в качестве базы для перечисления
// 8-битный целочисленный тип без знака.
enum Color : std::uint_least8_t
{
    color_black,
    color_red,
    // ...
};
```

enum class

```
enum class Color
{
  red,
  blue, // blue помещается в пространство имен Color
  green
};

enum class Feeling
{
  happy,
  tired,
  blue // нормально, blue помещается в пространство имен Feeling
};

Color color = Color::blue;
Color color = blue; // ошибка
```

union (объединения)

```
// Структура
struct building
{
    std::string owner;
    std::string city;
    int amountRooms;
    float price;
};

building apartment;
apartment.owner = "John";
apartment.city = "NY";
apartment.amountRooms = 10;
apartment.price = 100;
```

```
// Объединение
union building
{
    std::string owner;
    std::string city;
    int amountRooms;
    float price;
};

building apartment;
// Только что-то одно
```

Объединение занимает в памяти столько места, сколько занимает самое большое его поле.
После инициализации менять тип данных перечисления нельзя (хотя иногда можно)

namespace (пространство имён)

```
// Эта doSomething() складывает значения своих параметров
int doSomething(int x, int y)
{
    return x + y;
}


// Эта doSomething() вычитает значения своих параметров
int doSomething(int x, int y)
{
    return x - y;
}


std::cout << doSomething(4, 3) << '\n'; // какую doSomething мы получим?
```

namespace

```
namespace foo // определяем пространство имен с именем foo
{
    int doSomething(int x, int y) // Эта doSomething() принадлежит foo
    {
        return x + y;
    }
}

namespace goo // определяем пространство имен goo
{
    int doSomething(int x, int y) // Эта doSomething() принадлежит goo
    {
        return x - y;
    }
}

std::cout << doSomething(4, 3) << '\n'; // какую doSomething мы получим?
```

namespace

```
namespace foo // определяем пространство имен с именем foo
{
    int doSomething(int x, int y) // Эта doSomething() принадлежит foo
    {
        return x + y;
    }
}

namespace goo // определяем пространство имен goo
{
    int doSomething(int x, int y) // Эта doSomething() принадлежит goo
    {
        return x - y;
    }
}

// используем doSomething(), которая находится в пространстве имен foo
std::cout << foo::doSomething(4, 3) << '\n';
```

namespace

```
void print() // эта функция print находится в глобальном пространстве имен
{
    std::cout << " there\n";
}

namespace foo
{
    void print() // эта функция print находится в пространстве имен foo
    {
        std::cout << "Hello";
    }
}

int main()
{
    foo::print(); // вызываем print() из пространства имен foo
    ::print();    // вызываем print() из глобального пространства имен
                  // (в этом случае то же самое, что и просто вызов print())
}
```

namespace

```
void print() // эта функция print находится в глобальном пространстве имен
{
    std::cout << " there\n";
}

namespace foo
{
    void print() // эта функция print эта функция в пространстве имен foo
    {
        std::cout << "Hello";
    }

    void printHelloThere()
    {
        print();   // вызывает print() из пространства имен foo
        ::print(); // вызывает print() из глобального пространства имен
    }
}

int main()
{
    foo::printHelloThere();
}
```

namespace

```
namespace foo
{
    void print()
    {
        std::cout << "Hello";
    }
}

namespace foo
{
    void printHelloThere()
    {
        print();
    }
}
```

Не добавляйте пользовательские функции в пространство имен std.

namespace

```
namespace foo
{
    namespace goo // goo - это пространство имен внутри пространства имен foo
    {
        int add(int x, int y)
        {
            return x + y;
        }
    }
}


std::cout << foo::goo::add(1, 2) << '\n’;

namespace boo = foo::goo; // boo теперь ссылается на foo::goo
std::cout << boo::add(1, 2) << '\n'; // На самом деле это foo::goo::add()

using namespace foo;
std::cout << goo::add(1, 2) << '\n';

using foo::goo::add;
std::cout << add(1, 2) << '\n';
```

namespace (анонимное)

```
namespace // безымянное пространство имен
{
    void doSomething() // доступно только в этом файле
    {
        std::cout << "v1\n";
    }
}


doSomething(); // мы можем вызвать doSomething() без префикса пространства имен
// в другом cpp файле не доступна
```

namespace (встраиваемое)

```
void doSomething()
{
    std::cout << "v1\n";
}

int main()
{
    doSomething();
}
```

namespace (встраиваемое)

```
inline namespace v1 // объявить встраиваемое пространство имен с именем v1
{
    void doSomething()
    {
        std::cout << "v1\n";
    }
}

namespace v2       // объявить обычное пространство имен с именем v2
{
    void doSomething()
    {
        std::cout << "v2\n";
    }
}

int main()
{
    v1::doSomething(); // вызывает версию v1 функции doSomething()
    v2::doSomething(); // вызывает версию v2 функции doSomething()

    doSomething();     // вызывает встраиваемую версию doSomething() (это v1)

    return 0;
}
```

Дружественные функции

```
class Accumulator{
private:
    int m_value;
public:
    Accumulator() { m_value = 0; }
    void add(int value) { m_value += value; }

    // Сделаем функцию reset() другом этого класса
    friend void reset(Accumulator &accumulator);
};

// reset() теперь является другом класса Accumulator
void reset(Accumulator &accumulator)
{
    // и может получить доступ к закрытым данным объектов Accumulator
    accumulator.m_value = 0;
}


Accumulator acc;
acc.add(5); // добавляем 5 в накапливающий сумматор
reset(acc); // сбрасываем накапливающий сумматор в 0
```

Перегрузка операторов

- Во-первых, почти любой существующий оператор в C++ может быть перегружен. Исключениями являются: условный оператор (?:), sizeof, оператор разрешения области видимости (::), оператор выбора члена (.), оператор выбора указателя на член (.*), typeid и операторы приведения типов.

- Во-вторых, вы можете перегрузить только существующие операторы. Вы не можете создавать новые операторы или переименовывать существующие операторы. Например, вы не можете создать оператор ** для возведения в степень.

- В-третьих, по крайней мере, один из операндов в перегруженном операторе должен быть пользовательского типа. Это означает, что вы не можете перегрузить оператор плюс для работы с одним числом int и одним числом double. Однако вы можете перегрузить оператор плюс для работы с числом int и Mystring.

- В-четвертых, невозможно изменить количество операндов, поддерживаемых оператором.

- Наконец, все операторы сохраняют свой приоритет и ассоциативность по умолчанию (независимо от того, для чего они используются), и это не может быть изменено

Перегрузка операторов

- Обычная внешняя функция
- Дружественная функция
- Член класса

Перегрузка операторов

```
class Cents{
private:
    int m_cents;

public:
    Cents(int cents) { m_cents = cents; }
    int getCents() const { return m_cents; }
};
```

Перегрузка операторов

```
class Cents
{
private:
    int m_cents;

public:
    Cents(int cents) { m_cents = cents; }

    // складываем Cents + Cents с помощью дружественной функции
    friend Cents operator+(const Cents &c1, const Cents &c2);

    int getCents() const { return m_cents; }
};

// обратите внимание: эта функция не является функцией-членом!
Cents operator+(const Cents &c1, const Cents &c2)
{
    return Cents(c1.m_cents + c2.m_cents);
}

int main()
{
    Cents cents1(6);
    Cents cents2(8);
    Cents centsSum = cents1 + cents2;
    std::cout << "I have " << centsSum.getCents() << " cents.\n";
}
```

Перегрузка операторов

```
class Cents
{
private:
    int m_cents;

public:
    Cents(int cents) { m_cents = cents; }

    friend Cents operator+(const Cents &c1, int value);
    friend Cents operator+(int value, const Cents &c1);
    int getCents() const { return m_cents; }
};

Cents operator+(const Cents &c1, int value)
{
    return { c1.m_cents + value };
}

Cents operator+(int value, const Cents &c1)
{
    return { c1.m_cents + value };
}

int main()
{
    Cents c1 = Cents(4) + 6;
    Cents c2 = 6 + Cents(4);

    std::cout << "I have " << c1.getCents() << " cents.\n";
    std::cout << "I have " << c2.getCents() << " cents.\n";
}
```

Перегрузка операторов

Операторы присваивания (=), индекса ([]), вызова функции (()) и выбора члена (->) должны быть перегружены как функции-члены, потому что так требует язык.

Перегрузка операторов

```
class Cents{
private:
    int m_cents;

public:
    Cents(int cents) { m_cents = cents; }

    // Перегрузка Cents + int
    Cents operator+(int value);

    int getCents() const { return m_cents; }
};

// обратите внимание: эта функция является функцией-членом!
// параметр cents в дружественной версии теперь неявный параметр *this
Cents Cents::operator+(int value)
{
    return Cents(m_cents + value);
}

int main()
{
    Cents cents1(6);
    Cents cents2 = cents1 + 2;
    std::cout << "I have " << cents2.getCents() << " cents.\n";
}
```

Перегрузка операторов

```
class Cents
{
private:
    int m_cents;
public:
    Cents(int cents=0): m_cents(cents){
    }

    // перегруженное приведение в int
    operator int() const { return m_cents; }

    int getCents() const { return m_cents; }
    void setCents(int cents) { m_cents = cents; }
};
```

Перегрузка операторов

```
class Matrix
{
private:
    double m_data[4][4]{};

public:
    double& operator()(int row, int col);
    double operator()(int row, int col) const; // для константных объектов
};

double& Matrix::operator()(int row, int col)
{
    assert(col >= 0 && col < 4);
    assert(row >= 0 && row < 4);

    return m_data[row][col];
}

double Matrix::operator()(int row, int col) const
{
    assert(col >= 0 && col < 4);
    assert(row >= 0 && row < 4);

    return m_data[row][col];
}

int main()
{
    Matrix matrix;
    matrix(1, 2) = 4.5;
    std::cout << matrix(1, 2) << '\n';
}
```