ООП C++. Объекты. Жизненный цикл. Перегрузка операторов

Подготовительное отделение C/C++ (открытый курс)

образование

Отличия C и C++: парадигма программирования

Парадигма программирования — это совокупность идей и понятий, определяющих стиль написания компьютерных программ (подход к программированию).

C: Процедурно-ориентированная парадигма

- Проектирование в терминах взаимодействующих функций
- Код воздействует на данные

C++: Объектно-ориентированная парадигма

- Проектирование в терминах взаимодействующих классов
- Данные управляют доступом к коду

```
+++ #include <time.h>

+++ int lifespan_max = 0;

    typedef struct {
        int* data;
        size_t size;
+++     time_t create_time;
    } vector_t;

    vector_t* create() {
        vector_t* vector = calloc(1, sizeof(*vector));
+++     vector->create_time = time(NULL);
        return vector;
    }

    void destroy(vector_t* vector) {
+++    int lifespan = time(NULL) - vector->create_time;
+++    if (lifespan > lifespan_max)
+++        lifespan_max = lifespan;
        free(vector->data);
        free(vector);
    }

    {
        ...
        vector_t* vector = create();
        ...
        destroy(vector)
        ...
    }
```

```
...

vector_t v = {
    .data = calloc(10, sizof(int)),
    .size = 10,
};
...
free(v.data);

...

vector_t a = { malloc(128*sizeof(a.data[0])), 128 };
...
free(a.data);

...
```

```c
typedef struct {
    int* data;

    size_t size;
} stack_t;


stack_t* stack_create();
void stack_destroy(stack_t* stack);
void stack_push(stack_t* stack, int elem);
...
bool stack_is_empty(stack_t* stack);
int stack_pop(stack_t* stack);
int stack_top(stack_t* stack);
```

```c
typedef struct {
    int* data;
    int* max;
    size_t size;
} stack_with_max_t;


stack_with_max_t* stack_with_max_create();
void stack_with_max_destroy(stack_with_max_t* stack);
void stack_with_max_push(stack_with_max_t* stack, int elem);
...
bool stack_with_max_is_empty(stack_with_max_t* stack);
int stack_with_max_pop(stack_with_max_t* stack);
int stack_with_max_top(stack_with_max_t* stack);

int stack_with_max_get_max(stack_with_max_t* stack);
```

Постулаты ООП

1) **Абстракция** – возможность оперировать сущностями произвольной сложности как единым целым, не вдаваясь в детали внутреннего построения и функционирования.
2) **Инкапсуляция** – механизм, связывающий вместе код и данные, которыми он манипулирует, и одновременно защищающий их от произвольного доступа со стороны другого кода, внешнего по отношению к рассматриваемому. Доступ к коду и данным жестко контролируется интерфейсом.
3) **Наследование** – механизм, с помощью которого один объект (производного класса) приобретает свойства другого объекта (родительского, базового класса). При использовании наследования новый объект не обязательно описывать, начиная с нуля, что существенно упрощает работу программиста. Наследование позволяет какому-либо объекту наследовать от своего родителя общие атрибуты, а для себя определять только те характеристики, которые делают его уникальным внутри класса.
4) **Полиморфизм** – механизм, позволяющий использовать один и тот же интерфейс для общего класса действий.

Отличия C и C++: работа с динамической памятью

```c
int* e = malloc(sizeof(*e)); *e = 10;

int* zero_arr = calloc(10 * sizeof(*zero_arr));

int* arr = malloc(5 * sizeof(*arr));
for (int i = 0; i < 5; ++i)
    arr[i] = i + 1;

free(e);
free(zero_arr);
free(arr);
```

```c
int* e = new int(10); // new int{ 10 } also possible

int* zero_arr = new int[10];

int* arr = new int[]{1, 2, 3, 4, 5};

delete e;
delete[] zero_arr;
delete[] arr;
```

Отличия C и C++: операторы инициализации

```
switch (int i = f(); i) {
    ...
}

for (int i = 0; i < 10; ++i) {
    ...
}

if (int i = f(); i > 10) {
    ...
} else {
    ...
}
```

Отличия C и C++: значения параметров по умолчанию

```
void f(int a, int b = -1, int c = -2);

...

f(1);          // a == 1, b == -1, c == -2
f(1, 2);       // a == 1, b ==  2, c == -2
f(1, 2, 3);    // a == 1, b ==  2, c ==  3
```

Отличия C и C++: стандартный ввод-вывод

```c
#include <stdio.h>


int i;
long l;
char s[10 + 1];


scanf("%d%ld%10s", &i, &l, s);

printf("%d %ld %s\n", i, l, s);
```

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


int i;
long l;
std::string s;


std::cin >> i >> l >> s;

std::cout << i << " " << l << " " << s << std::endl;
```

Отличия C и C++: ссылочный тип данных

```
int a = 3;
int* b = &a; // b – указатель на a
int* c; // c – невалидный указатель

a = 5;
std::cout << *b << std::endl; // 5



int i = 3;
b = &i; // b теперь указывает на i

...

void f(int* i) {
    // i может быть NULL
    // нужно разыменование указателя
    // для обращения к значению
    int previous = *i;
    *i = 0;
}
...
int a = 5;
f(&a); // требуется явно передать в функцию адрес
```

```
int a = 3;
int& b = a; // b – синоним a
int& c; // ОШИБКА КОМПИЛЯЦИИ!

a = 5;
std::cout << b << std::endl; // 5

assert(&a == &b); // всегда истинно

int i = 3;
b = i; // ОШИБКА! Ссылку нельзя изменить
       // после создания
...

void f(int& i) {
    // i не может быть NULL

    // для обращения к значению не нужно разыменование
    int previous = i;
    i = 0;
}
...
int a = 5;
f(a);  // передача аргумента выполняется
       // аналогично передаче по значению
```

Отличия C и C++: пространства имён. Операция разрешения области видимости имён (::)

```
int x;

namespace A {
    int x;
    void f() {
        std::cout
            << "local: " << x
            << "global: " << ::x
            << std::endl;
    }
}

namespace B {
    void f() {
        f(); // <- рекурсивный вызов
        A::f(); // <- вызов другой функции
    }
}
...

std::cout << A::x << std::endl;
```

```
namespace outer {
    namespace inner {
        void f();
    }
}
// same
namespace outer::inner {
    void f();
}
...

outer::inner::f();
```

```
using namespace std;


cout << "Hello, World!" << endl;
```

Отличия C и C++: необработанные строковые литералы

```
std::cout <<
    "{\n\
        \"a\": 3,\n\
        \"b\": \"str\"\n\
    }"
    << std::endl;

std::cout <<
    R"({
        "a": 3,
        "b": "str"
    })"
    << std::endl;

std::cout <<
    R"~({
        "a": 3,
        "b": ")"
    })~"
    << std::endl;
```

Классы и объекты в C++

Класс — центральное понятие в ООП.

Класс описывает тип, на основе которого создаются **объекты**.

Класс характеризуется:

1) множеством значений, которые могут принимать объекты класса;
2) множеством функций, задающих операции над объектами.

Классы и объекты в C++. Синтаксис описания класса

```
class имя_класса { описание_членов_класса }
```

Члены класса:

1) Информационные члены (поля)
   - Хранятся в объектах класса
2) Функции-члены (методы)
   - Одинаковые для всех объектов класса

Классы и объекты в C++. Управление доступом к членам класса

```
class имя_класса {
public:
    определение_открытых_членов_класса
private:
    определение_закрытых_членов_класса
protected:
    определение_защищённых_членов_класса
};
```

```
class имя_класса {
    определение_закрытых_членов_класса
    ...
};
```

```
struct имя_структуры {
    определение_открытых_членов_класса
    ...
};
```

Классы и объекты в C++. Модификаторы и селекторы

```
class vector {
    ...
public:
    void set_size(size_t new_size) {
        if (new_size != size) {
            int* new_buffer = new int[new_size];
            for (int i = 0; i < std::min(size, new_size); ++i)
                new_buffer[i] = buffer[i];
            delete[] buffer;
            buffer = new_buffer;
            size = new_size;
        }
    }

    size_t get_size() {
        return size;
    }

private:
    int* buffer;
    size_t size;
};
```

Классы и объекты в C++. Создание объектов и доступ к их членам

```
class X {
public:
    int n;
    int f() {...}
};

...

X x1;
X& x2 = x1;
X* p = &x1;


int i = x1.f();
int j = x2.f();
int k = p->f();


x1.n = 3;
assert(x2.n == 3);
assert(p->n == 3);
```

```
class Y {...} y1, y2;
```

Классы и объекты в C++. Класс как область видимости

```
void f() {...}


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


    void g() {
        f(); // A::f()
        A::f(); // то же самое


        ::f(); // вызов глобальной функции
    }
};
```

Классы и объекты в C++. Объявление и определение методов класса. Спецификатор *inline*

```hpp
// .hpp

class X {
    int n;

public:
    void f();

    void g() {
        f();
    }
};
```

```cpp
// .cpp

void X::f() {
    std::cout << n << std::endl;
}
```

```hpp
// .hpp

class X {
    int n;

public:
    inline void f();

    void g() {
        f();
    }
};
```

```cpp
void X::f() {
    std::cout << n << std::endl;
}
```

Классы и объекты в C++. Указатель *this*

```
class A {
    int n;

public:
    // A& f(A* this, int i)
    A& f(int i) {
        this->n = i; // то же самое, что n = i;
        std::cout << n << std::endl;
        return *this;
    }

    // A& g(A* this, int i)
    A& g(int i) {
        return f(i - 1); // A::f(this, i - 1);
    }
};
...


A a;
A* ap = &a;

a.g(1);   // A::g(&a, 1);
ap->g(2); // A::g(ap, 2);
```

Конструкторы и деструкторы

```
class vector {
    size_t size;
    int* buff;

public:
    vector() {
        size = 0; buff = NULL;
    }

    vector(int init_size, int init_value = 0) {
        size = init_size;
        buff = new int[size];
        for (int i = 0; i < size; ++i)
            buff[i] = init_value;
    }

    vector(const vector& other) {
        size = other.size;
        buff = new int[size];
        for (int i = 0; i < size; ++i)
            buff[i] = other.buff[i];
    }

    ~vector() { delete[] buff; }
};
```

```
{
    vector a;          // vector()
    vector b(5);       // vector(int init_size = 5,
                       //        int init_value = 0)
    vector c(10, 1);   // vector(int init_size = 10,
                       //        int init_value = 1)
    vector d(c);       // vector(const vector& other = c)
    vector e = d;      // vector(const vector& other = d)


    vector* p = new vector(3, 7); // vector(
                                  //     int init_size = 3,
                                  //     int init_value = 7)
    delete p;          // ~vector()


} // other destructors in reverse order: e, d, c, b, a
```

Конструкторы и деструкторы. Конструктор копирования

```
class X {
public:
    X(X&);          // может изменять копируемый объект
    X(const X&);    // не может изменять копируемый объект
};
```

Конструкторы и деструкторы. Спецификатор *explicit*

```
class Y {...};


class X {
public:
    X(const Y&);
};


Y a;
X b = a; // X(const Y&)


void f(X);


f(a);    // X(const Y&)
```

```
class Y {...};


class X {
public:
    explicit X(const Y&);
};


Y a;
X b = X(a); // X(const Y&)


void f(X);


f(X(a));    // X(const Y&)
```

Конструкторы и деструкторы. Автоматическая генерация

```
class X {
    // public: X();
    // public: X(const X&);
    // public: ~X();
};
```

```
class X {
public:
    X(X&);


    // public: X();
    // public: X(const X&);
    // public: ~X();
};
```

```
class X {
public:
    X(int);


    // public: X();
    // public: X(const X&);
    // public: ~X();
};
```

```
class X {
public:
    ~X();


    // public: X();
    // public: X(const X&);
    // public: ~X();
};
```

Конструкторы и деструкторы. Инициализация объекта

```
class A {
public:
    A();
    A(int, int);
};


class X {
public:
    int a;
    int b;
    int c{};
    int d = 3;
    A x;
    A y{5, 9};


    X(): b(0), c{3}, x(7, 10)
    {
        ...
    }
};
```

Статические члены класса

```
static void f() {
    static int counter = 0;
    std::cout << ++counter << std::endl;
}
```

```
...

f(); // 1
f(); // 2
f(); // 3
```

Статические члены класса

```
// .hpp

class X {
    static int count;


public:
    X() { ++count; }

    static int getCount() { return count; }
};



// .cpp

int X::count = 0;

...

X a, b;

std::cout << a.getCount() << " " << b.getCount() << " " << X::getCount() << std::endl; // 2 2 2
```

Константные члены класса

```
class X {
    ...
    int size;

public:
    int getSize() const { return size; } // int getSize(const X* this)
};


void f(const X& x) {     // typeof(&x) == const X*
    int n = x.getSize(); // X::getSize(&x)
    ...
}


...

X x;

f(x);
```

Друзья класса

```
class Vertex;


class SetOfVertexPointers {
public:
    void insert(Vertex*);
    void erase(Vertex*);
};


class DirectedGraph {
public:
    Vertex* AddVertex();
    void AddEdge(Vertex* from, Vertex* to);

private:
    SetOfVertexPointers vertices;
};
```

```
class Vertex {
public:
    ...

private:
    friend DirectedGraph;
    friend class DirectedGraph;  // forward declaration
    friend void DirectedGraph::AddEdge(Vertex* from, Vertex* to);

    SetOfVertexPointers outgoing_edges;
};


Vertex* DirectedGraph::AddVertex() {
    Vertex* vertex = new Vertex;
    vertices.insert(vertex);
    return vertex;
}


void DirectedGraph::AddEdge(Vertex* from, Vertex* to) {
    from->outgoing_edges.insert(to);
}
```

Статический полиморфизм. Перегрузка операций

```
class Fraction {
    signed dividend;
    unsigned divisor;

public:
    Fraction(signed a = 0, signed b = 1) {
        assert(b != 0);
        if (b < 0) {
            dividend = -a;
            divisor = -b;
        } else {
            dividend = a;
            divisor = b;
        }
    }
    Fraction operator+(const Fraction& other) const {
        return Fraction(
            dividend * other.divisor + other.dividend * divisor,
            divisor * other.divisor
        );
    }
};

Fraction a(3, 5);
Fraction b(1, 3);
Fraction c = a + b; // a.operator+(b)
```

Статический полиморфизм. Перегрузка операций

```
class Fraction {
    signed dividend;
    unsigned divisor;

public:
    Fraction(signed a = 0, signed b = 1);
    Fraction operator+(const Fraction& other) const {
        return Fraction(
            dividend * other.divisor + other.dividend * divisor,
            divisor * other.divisor
        );
    }
    Fraction operator-() const { return Fraction(-dividend, divisor); }
    Fraction operator-(const Fraction& other) const { return *this + -other; }
    friend Fraction operator*(const Fraction&, const Fraction&);
};

Fraction operator*(const Fraction& a, const Fraction& b) {
    return Fraction(a.dividend * b.dividend, a.divisor * b.divisor);
}

Fraction a(3, 5);
Fraction b(1, 3);
Fraction c = a + b; // a.operator+(b)
Fraction d = -c;    // c.operator-()
Fraction e = a - b; // a.operator-(b)
Fraction x = a * b; // operator*(a, b)
```

Статический полиморфизм. Перегрузка операций

```
class Fraction {
    signed dividend;
    unsigned divisor;

public:
    ...
    friend std::ostream& operator<<(std::ostream&, const Fraction&);
    friend std::istream& operator>>(std::istream&, Fraction&);
};
std::ostream& operator<<(std::ostream& out, const Fraction& fraction) {
    out << fraction.dividend;
    if (fraction.dividend != 0 && fraction.divisor != 1)
        out << '/' << fraction.divisor;
    return out;
}

std::istream& operator>>(std::istream& in, Fraction& fraction) {
    char c;
    if (((in >> (fraction.dividend)) >> c) >> (fraction.divisor)) && (c != '/'))
        in.setstate(std::ios_base::failbit);
    return in;
}

Fraction a;
std::cin >> a;
std::cout << a << std::endl;
```

Статический полиморфизм. Перегрузка операций

```
class Fraction {
    signed dividend;
    unsigned divisor;

public:
    Fraction(signed a = 0, signed b = 1);
    ...
    operator double() const { return double(dividend) / divisor; }
};

std::ostream& operator<<(std::ostream& out, const Fraction& fraction) {
    out << fraction.dividend;
    if (fraction.dividend != 0 && fraction.divisor != 1)
        out << '/' << fraction.divisor;
    return out;
}



...
Fraction a(1, 4);
double b = a;

std::cout << a << " == " << b << std::endl; // 1/4 == 0.25
```

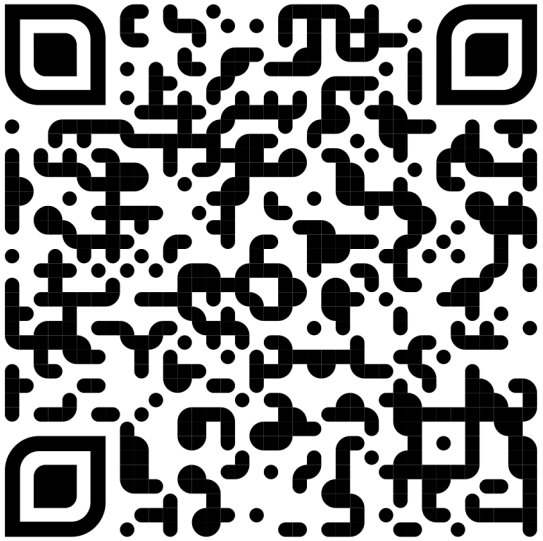
Статический полиморфизм. Перегрузка операций

```
class Vector {
    size_t size;
    int* data;

public:
    Vector(size_t init_size = 0):
        size(init_size), data(new int[size]){}
    ~Vector() { delete[] data; }
    size_t getSize() const { return size; }

    int& operator[](size_t pos) {
        return data[pos];
    }
    const int& operator[](size_t pos) const {
        return data[pos];
    }
    Vector& operator=(const Vector& other) {
        if (this != &other) {
            delete[] data;
            data = new int[size = other.size];
            for (int i = 0; i < size; ++i)
                data[i] = other.data[i];
        }
        return *this;
    }
};
```

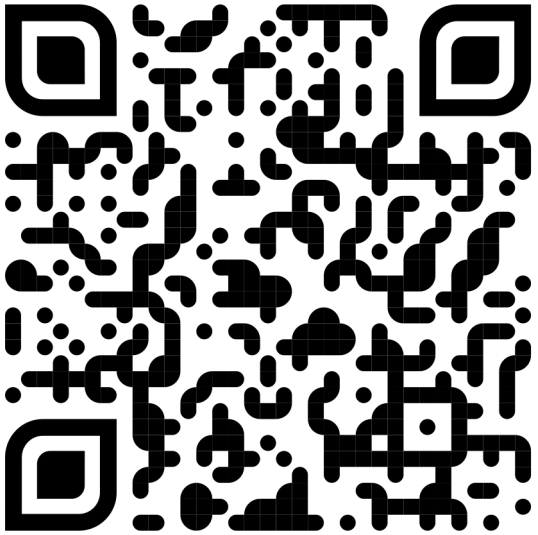
```
std::ostream& operator<<(std::ostream& out, const Vector& v) {
    for (int i = 0; i < v.getSize(); ++i) {
        out << v[i];
        if (i + 1 != v.getSize())
            out << ", ";
    }
    return out;
}
...

Vector a(3);
Vector b;

a[0] = 1;
a[1] = 2;
a[2] = 3;

std::cout << a << std::endl;  // 1, 2, 3

b = a;

std::cout << b << std::endl;  // 1, 2, 3

a[1] = 20;

std::cout << b << std::endl;  // 1, 20,33
```

Статический полиморфизм. Перегрузка операций

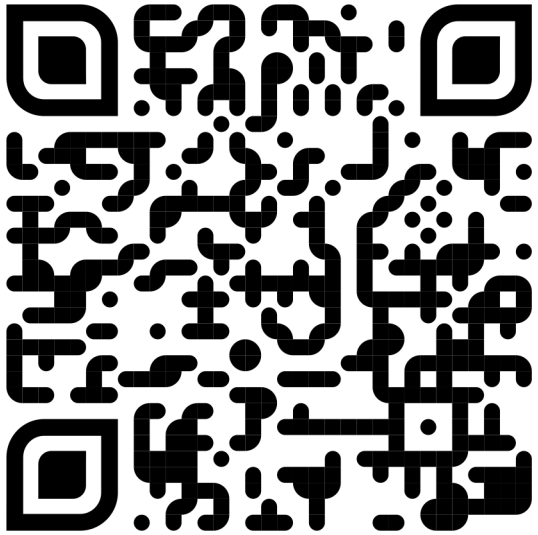
Полезные ссылки:



Операторы в C++



Правила перегрузки операций



Приоритет и ассоциативность операторов в C++

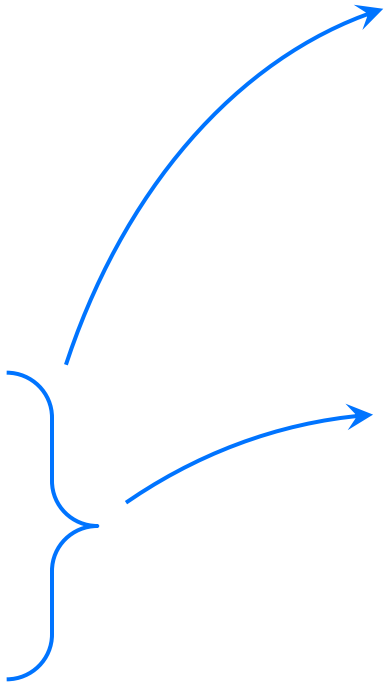
Алгоритм поиска оптимально отождествляемой функции

1. Отбираются функции с необходимым количеством формальных параметров;
2. Для каждого фактического параметра вызова функции строится множество функций, оптимально отождествляемых по этому параметру (best matching);
3. Находится пересечение этих множеств;
4. Если полученное множество состоит из одной функции, то вызов разрешим. Если множество пусто или содержит более одной функции, то генерируется сообщение об ошибке.

```
void f(int, int, int = 0)             { std::cout << __PRETTY_FUNCTION__ << std::endl; }
void f(double, double, double = 0.0) { std::cout << __PRETTY_FUNCTION__ << std::endl; }
void f(int, double, double = 0.0)     { std::cout << __PRETTY_FUNCTION__ << std::endl; }
void f(int, double, ...)              { std::cout << __PRETTY_FUNCTION__ << std::endl; }

...
f(0, 0); // void f(int, int, int)
f(0, 0.0); // ambiguous: 3, 4
f(0.0, 0); // no candidate
f(0.0, 0.0); // void f(double, double, double)
```

Алгоритм поиска оптимально отождествляемой функции для одного параметра

1. Точное отождествление:
   1. Точное совпадение;
   2. Совпадение с точностью до `typedef`;
   3. Тривиальные преобразования:
      - `T[] <-> T*;`
      - `T   <-> T&;`
      - `T    -> const T;`
2. Отождествление с помощью расширений:
   1. Целочисленные расширения без потери точности;
   2. Расширения с плавающей точкой: `float -> double;`
3. Отождествление с помощью стандартных преобразований;
   1. Остальные стандартные целочисленные и вещественные преобразования;
   2. Преобразование указателей
4. Отождествление с помощью преобразований пользователя;
   1. Конструктор преобразования;
   2. Функция преобразования (операция преобразования);
5. Отождествление по «…».



Неявные преобразования

Приоритеты неявных преобразований

Спасибо за внимание!

E-mail: i.anferov@corp.mail.ru
Telegram: igor_anferov
GitHub: igor-anferov

Игорь Анфёров

образование