Немного про строки

```
#include <iostream>
//про строки
int main () {
    //сначала создадим указатель
    char* wrong;
    wrong = "wtf"; //так можно, но не рекомендуется
    //потому что "wtf" - это константный строковый литерал с константным адресом,
    //при любой попытке изменения будет такая ошибка:
    //wrong[0] = 't'; //EXC_BAD_ACCESS
    //следовательно, лучше делать так:
    const char* right; //сразу запрещаем любое изменение значения под указателем
    //на уровне компиляции
    right = "All right";
    //для указателей типа char побитовый сдвиг << определён так, что
    //вставляется в поток уже самое значение, а не адрес:
    std::cout << right << '\n';
    //такие константные указатели можно изменять так:
    right = "No";
    std::cout << right << '\n';
    //если метод (функция) может принимать константные строковые литералы, лучше принимать
    //const char*, если в неё передать просто char*, то char* прекрасно прикастуется
    //к const char*:
    right = wrong;
    std::cout << right << '\n';
}
```

Про const char* в методах

```
#include <iostream>
//про строки
struct Test {
    char * name;
    Test (const char* n){ //имеется в виду этот момент
        name = new char[strlen(n)+1];
        strcpy(name, n);
    }
   ~Test() {delete [] name; }
};
int main () {
    //Как работаем с указателем:
    char* wrong = new char[4];
    strcpy(wrong,"wtf");
    //если метод (функция) может принимать константные строковые литералы, лучше принимать
    //const char*, если в неё передать просто char*, то char* прекрасно прикастуется
    //к const char*:
    Test object("Hello"); //const char*
    Test object1(wrong); // char *
    std::cout <<object.name << '\n';
    std::cout <<object1.name << '\n';
    delete [] wrong;
}
```

Статический массив char

```
#include <iostream>
//про строки
void f(const char* t) {
    std::cout << t;
}
int main () {
    //а теперь статический массив
    char stat[20];
    //он создаётся в стековой памяти с фиксированным и неизменяемым адресом.
    //stat - это адрес статического массива.
    //любые попытки что-то присвоить stat будут расценены как нелегальное
    //изменение адреса:
    //stat = "wrong"; //Array type 'char [20]' is not assignable
    char *p;
    //stat = p; //Array type 'char [20]' is not assignable
    //в обратную сторону, конечно, можно:
    p = stat;
    //в статический массив что-то положить можно только через strcpy
    //(ну или посимвольным копированием):
    strcpy(stat, "wtf");
    std::cout << stat << '\n'; //wtf
    std::cout << p << '\n';    //wtf
    //из-за того, что p указывает туда же,
    //можно сделать и так:
    strcpy(p, "Hello");
    std::cout << stat << '\n'; //Hello
    std::cout << p << '\n';    //Hello
    //Понятное дело, его тоже можно передавать в функцию:
    f(stat);
}
```

Немного про преобразование типов

```
#include <iostream>
//если char* без проблем кастуется к const char*,
//то const char* не кастуется к char*:
char* f(const char* t) {
    return t;
}
//Cannot initialize return object of type 'char *' with an lvalue of type 'const char *'
//То есть попытка инициализировать char* (обычный указатель),
//используя объект типа const char*, который является lvalue (то есть существующей переменной).
int main () {
    std::cout << f("test");
}
```

Как делать НЕ НАДО
(про функцию преобразования потом)

```
#include <iostream>

class S {
private:
    const int x{5};
    double z{0};
public:
    void f(int y) {
        std::cout << x + y;
    }
};

int main () {
    S str;
    (int&)str += 6; //это не просто преобразование, тут низкоуровневая
работа с памятью. Весь объект становится lvalue, которое интерпретируется
как int и может изменяться
    std::cout << (int&)str << '\n';
    str.f(7);
}
```

Друзья класса

```cpp
#include <iostream>

class S {
    private:
        int x = 5;
    public:
        void f(int y) {
            std::cout << x + y;
        }
 };

void g (const S& s, int y) {
    std::cout << s.x + y +1 << '\n';
}

int main () {
    S str;
    str.f(5);
}
```

2 'x' is a private member of 'S'

friend

```
class S {
    private:
        int x = 5;
    public:
        void f(int y) {
            std::cout << x + y;
        }
    friend void g(const S&, int);
 };

void g(const S& s, int y) {
    std::cout << s.x + y +1 << '\n';
}

int main () {
    S str;
    str.f(5);
}
```

```
class S {
    private:
        int x = 5;
    friend void g(const S&, int);
    friend class G;
 };

class G {
private:
    int c = 7;
public:
    S test;
    void g (const S&, int y) {
        std::cout << test.x + y + 2 << '\n';
    }
};

void g(const S& s, int y) {
    std::cout << s.x + y +1 << '\n';
}

int main () {
    S str;
    g(str, 5);
    G gtr;
    gtr.g(str, 5);
}
```

Проектирование и программирование

Проектирование (*designing*) — процесс определения архитектуры, компонентов, интерфейсов и других характеристик системы или её части (ISO 24765).

Процесс **программирования** (*programming*) состоит в последовательной или итеративной реализации компонент программной системы средствами конкретного языка.

ООП и ООП

Объектно-ориентированное **программирование** – это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Объектно-ориентированное **проектирование** – это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приемы представления логической и физической, а также статической и динамической моделей проектируемой системы.

Именно объектно-ориентированная декомпозиция отличает объектно-ориентированное проектирование от структурного; в первом случае логическая структура системы отражается абстракциями в виде классов и объектов, во втором – алгоритмами.

Абстракция

Абстракция данных – это метод проектирования, который основан на разделении интерфейса и реализации.

абстрагирование направлено на наблюдаемое **поведение**(контракт) объекта;

Интерфейс и реализация

- **Интерфейс** отражает внешнее поведение объекта, специфицируя поведение всех объектов данного класса.

- **Внутренняя реализация** описывает представление этой абстракции и механизмы достижения желаемого поведения объекта

Объект

Объект — это экземпляр класса, или его копия, которая находится в памяти компьютера.

Можно сказать, что объекты — это сущности, у которых есть свойства и поведение.

Инициализация полей

При наличии нескольких полей, при инициализации можно ссылаться на значения ранее инициализированных полей:

```
class A {
private:
        count_t count1 = 0;
        count_t count2 = count_1 + 1;
        count_t count3 = count_2 + 1;
public: ...
};
```

Инициализация в конструкторе будет в приоритете

Конструктор

Конструктор – это метод класса, который вызывается при создании объекта, сразу же после выделения объекту памяти. Основное назначение этого метода – **провести инициализацию полей класса** и **выделить дополнительную память** под внутренние переменные или массивы.

Конструкторы бывают:

- **По умолчанию** – это конструктор без параметров в случае описания и неявный конструктор, назначаемый самой программой в случае отсутствия явного описания.
- **Обычный** – конструктор с параметрами.
- **Копирующий** – создающий копию имеющегося объекта.
- **Делегирующий**
- **Перемещающий** – выполняющий действия по перемещению данных их одного объекта в другой (C++11)

Конструктор по умолчанию

```
class Point {
    int x = 0;
    int y = 0;
};

int main (){
    Point p;
}
```

```
class Point {
    int x = 0, y = 0;
public:
    Point() {
        x = 0;
        y = 0;
    }
};
int main () {
    Point p;
}
```

```
class Point {
    int x, y;
public:
    Point(int a = 0, int b = 0) {
        x = a;
        y = b;
    }
};
int main (){
    Point p;
}
```

Вызывается:

```
int main (){
    Point p1;
    Point p2 = Point();
    Point* p3 = new Point;
    // В массивах:
    Point arr[10];
    delete p3;
}
```

Конструктор с параметрами

```
class Point {
    int x, y;
public:
    Point(int x_coord, int y_coord) {
        x = x_coord;
        y = y_coord;
    }
};


int main() {
    Point p1(10, 20);
}
```

Примеры инициализации объекта. Конструктор с параметрами

```
class Point {
private:
    double x = 0.0;
    double y = 0.0;
public:
    Point(double x1, double y1 = 4.4) {
        x = x1;
        y = y1;
    }
};

int main () {
    Point p(9.0);
    Point p1 = 8.0;
    Point p2{7.0};
    Point p3 = {6.0};
    Point p4 = Point(6.0);
}
```

```
class Str {
    char* s= nullptr;
public:
    Str(const char * s1) {
        s = new char[strlen(s1) + 1];
        memcpy (s, s1, strlen(s1));
    }
    char* getstr () {
        return s;
    }
    ~Str() {delete [] s;}
};
int main () {
    Str str("First object");
    std::cout << str.getstr();
}
```

Конструктор по умолчанию не генерируется, если определён любой другой

```
class Point {
private:
    double x = 0.0;
    double y = 0.0;
public:
    Point(double x, double y = 1.4): x(x), y(y) { }
};

int main () {
    Point p(9.0);
    Point p2{7.0};
    Point p3 = {6.0};
    Point p4;   No matching constructor for initialization of 'Point'
}
```

```
class Point {
private:
    double x = 0.0;
    double y = 0.0;
public:
    Point(){}
    Point(double x, double y = 1.4): x(x), y(y){}
};

int main () {
    Point p(9.0);
    Point p2{7.0};
    Point p3 = {6.0};
    Point p4;
}
```

Лучше использовать списки инициализации

```
class Point {
private:
    double x = 0.0;
    double y = 0.0;
public:
    //member initializer list
    Point(double x);
    Point(double x, double y): x(x), y(y) { }
};
Point::Point (double x): x(x), y(13) {
    std::cout << "I'm working\n";
}

int main () {
    Point p(9.0);
    Point p1 = 8.0;
    Point p2{7.0};
    Point p3 = {6.0};
}
```

Если ссылка или константа

```
class P {
    int& a;
public:
    P(int& b) {
        a = b;
    }
};

int main () {
    int aa;
    P pp (aa);
}
```

Constructor for 'P' must explicitly initialize the reference member 'a'

Ссылка, константа инициализируются только списком инициализации

```
class Test {
    int& ta;
public:
    Test(int& ta) : ta(ta) {}
    void setta (int& tb) {
        ta = tb;
    }
    int getta () {
        return ta;
    }
};
int main () {
    int a = 6;
    Test test(a);
    std::cout << test.getta() <<std::endl;
    int b = 7;
    test.setta(b);
    std::cout << test.getta() <<std::endl;
}
```

Ссылки переустанавливать нельзя!

```
int x = 6;

class P {
    int& xx = x;
public:
    P() {}
    P(int y) {
        xx = y;
    }
    int getstr () {
        return xx;
    }
};
int main () {
    P pp = 8;
    std::cout << x << std::endl;
    int r = 9;
    P test(r);
    std::cout << x << std::endl;
}
```

Полное и неполное объявление класса

```
class A {
public:
    int c;
};


class P {
    A a;
public:
    P(A b): a(b) {}
};


int main () {
    A aa;
    P pp (aa);
}
```

```
class A;
class P {
    const A& a;
public:
    P(const A& b): a(b) {}
};


class A {
public:
    char a = 'a';
};
int main () {
    A aa;
    P pp (aa);
}
```

Общее замечание про конструкторы

Если в классе присутствуют только тривиальные типы, которые не требуют особой обработки при инициализации, и не требуется создавать объект с какими-то входными данными, то можно конструктор писать только по умолчанию, без реализации, либо можно его вообще не определять.

Но тем не менее, для удобочитаемости и восприятия кода, даже тривиальный конструктор писать всё же желательно ☺

P.S.: Если в классе определён хотя бы один конструктор, то конструктор по умолчанию автоматически генерироваться не будет.

Если в классе есть поле нетривиального типа, конструктор определять обязательно

```
class Name {
    char* n = new char{};
public:
    Name() {}
    Name(const char* n): n(new char[strlen(n) + 1]) {
        memcpy (this->n, n, strlen(n));
    }
```

//, также как и деструктор

```
    ~Name() {
        delete [] n;
    }
};
```

Конструктор копирования

Если в классе присутствуют переменные нетривиальных типов, создание которых требует дополнительной обработки, то для таких классов обязательно требуется наличие конструктора (и деструктора), а также обязательно следует определять конструктор копирования. В противном случае копирование будет происходить по значению, что в итоге может спровоцировать множество ошибок.

Конструктор копирования, если не задан явно, создаётся компилятором автоматически

```
48  class Name {
49      char* n = nullptr;
50  public:
51      Name(): n(new char){}
52      Name(const char* n): n(new char[strlen(n) + 1]) {
53          memcpy (this->n, n, strlen(n));
54      }
55      ~Name() {
56          std::cout << "Dtor for: " << n <<std::endl;
57          delete [] n;                                  Thread 1: signal SIGABRT
58      }
59  };
60
61  int main () {
62      Name test("First");
63      Name test1("Second");
64      Name test2 = test1;
65  }
66
```

Lections 〉 Thread 1 〉 5 Name::~Name() Line: 40 Col: 46

this = (Name *) 0x7ffeefbff460

```
Dtor for: Second
Dtor for: Second
Lections(37047,0x1000efe00) malloc: ***
    error for object 0x100607ae0: pointer
    being freed was not allocated
Lections(37047,0x1000efe00) malloc: *** set
    a breakpoint in malloc_error_break to
    debug
(lldb)
```

Конструктор копирования для нетривиальных типов задавать обязательно

```
class Name {
    char* n = nullptr;
public:
    Name(): n(new char{}) {}
    Name(const char* n): n(new char[strlen(n) + 1]) {
        memcpy (this->n, n, strlen(n)+1);
    }
    Name(const Name& other): n(new char[strlen(other.n) + 1]) {
        memcpy (n, other.n, strlen(other.n)+1);
    }
    char* getname () {
        return n;
    }
    ~Name() {
        delete [] n;
    }
};
int main () {
    Name test("First object");
    Name test1 = test;          //ЭТО ВЫЗОВ КОНСТРУКТОРА КОПИРОВАНИЯ!
    Name test2(test1);
    std::cout << test2.getname() << std::endl;
}
```

Когда вызывается конструктор копий

Случай 1. В момент объявления нового объекта и его инициализации данными другого объекта.

```
Name test2 = test1;
Name test5(test2);
Name test6 = {test2};
Name test7{test2};
```

Случай 2. Когда нужно передать объект в функцию как параметр-значение. В этом случае создается полная копия объекта.

```
void func (Name test_name) {
    std::cout << "Hi";
}
```

Случай 3. Когда нужно вернуть объект из функции по значению. В этом случае также создается полная копия объекта (но если есть, вызывается конструктор перемещения*).

```
Name func (Name p) {
    std::cout << "Hi";
    return p;
}
```

Делегирующий конструктор

```
class Name {
    char* n = nullptr;
    Name(size_t size): n(new char[size + 1]) {
        n[size] = '\0';
    }
public:                                          //An initializer for a delegating constructor must appear alone
    Name() = default;
    Name(const char* n): Name(strlen(n)) {
        memcpy (this->n, n, strlen(n));
    }
    Name(const Name& other): Name(other.n) { }
    ~Name() {
        delete [] n;
    }
};
int main () {
    Name test("First");
    Name test1("Second");
    Name test2 = test1;
}
```

```
class Name {
    char* n = nullptr;
    Name(size_t size): n(new char[size + 1]) {
        n[size] = '\0';
    }
public:
    Name() = default;
    Name(const char* n): Name(strlen(n)) {
        std::cout << "Work\n" ;
        memcpy (this->n, n, strlen(n));
    }
    Name(const Name& other): Name(strlen(other.n)) {
        std::cout << "Work copy \n" ;
        memcpy (this->n, other.n, strlen(other.n));
    }
    Name& operator= (const Name& other) {
        if (this == &other)
            return *this;
        delete [] n;
        size_t length = strlen(other.n);
        n = new char[length + 1];
        memcpy (n, other.n, length);
        return *this;
    }
    ~Name() {
        delete [] n;
    }
};
```

Оператор присваивания, если отсутствует, тоже создаётся компилятором автоматически

Идиома copy&swap

```
class Employee {
    char* name = nullptr;
    char* surname = nullptr;
    char date[11] {"01.01.1901"};
public:
    Employee () { }
    Employee (const char* n, const char *s, const char d[]):
          name(new char[strlen(n)+1])
        , surname(new char[strlen(s)+1]) {
            memcpy(name, n, strlen(n));
            memcpy(surname, s, strlen(s));
            memcpy(date, d, strlen(d));
    }
    Employee (const Employee& other): Employee(other.name, other.surname, other.date){}
    Employee& operator=(Employee other) { //const Employee& other) {
        //if(this ==&other) return *this;
        //Employee copy = other;
        swap(other); //swap(copy);
        return *this;
    }
    void swap(Employee& other) {
        std::swap(name, other.name);
        std::swap(surname, other.surname);
        std::swap(date, other.date);
    }
    ~Employee () {
        delete [] name;
        delete [] surname;
    }
};
```