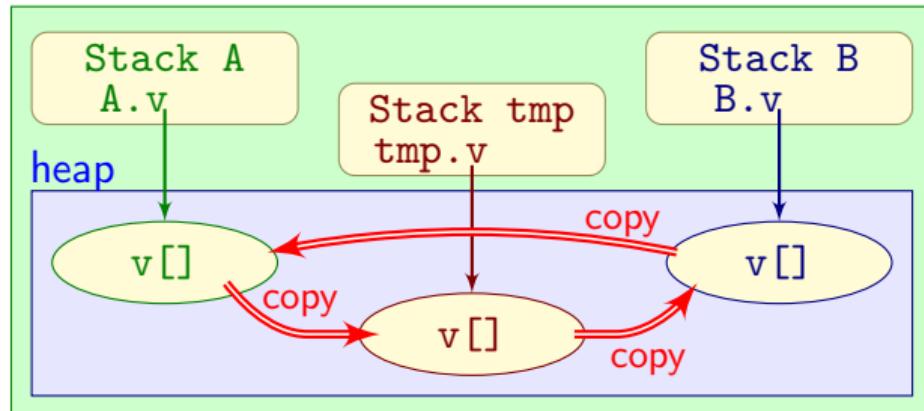


# Концепция перемещений (move) в C++11

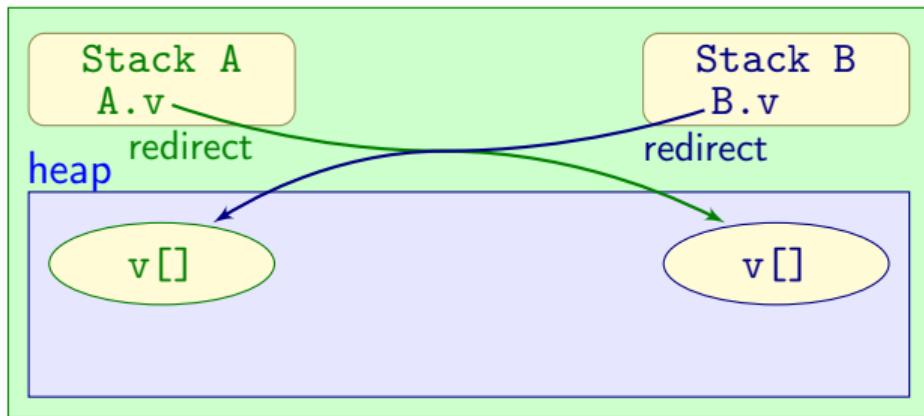
Пример ① : функция обмена для двух стеков

```
void swap_copy(Stack& a, Stack& b) {  
    Stack tmp(a); // а нужен ли tmp.v[]?  
    a = b;  
    b = tmp;  
}
```



## «Идеальное» решение

- ☛ Никакого копирования буферов нет и `tmp` не нужен!



- ☛ Возможное решение в C++98: внешняя функция, `friend of Stack`, для перенаправления указателей без копирования данных

## Пример ② : функция возвращает «большой объект»

```
Stack Reverse(Stack s) { // attention: here 's' is a copy
    Stack rs;
    while ( !s.isEmpty() ) {
        rs.push(s.pop());
    }
    return rs;
}
...
Stack rS( Reverse(S) ); // copy or move ctor...
Stack rS2 = Reverse(S); // copy or move assignment ...
```

### Дополнительные, ненужные расходы:

- 👉 Объект возвращается из функции, копируется во временный, а затем уничтожается

# Перемещающий конструктор и присваивание

- ☞ В стандарт C++11 введены перемещающий конструктор и перемещающий оператор присваивания для оптимизации

Две новые «обязательные функции»:

- move constructor: `Class(Class&&);`
  - move assignment: `Class& operator=(Class&&);`
- ☞ Если они отсутствуют, компилятор создаст свои версии при условии, что так же нет пользовательских копирующих «функций» и деструктора

Обратите внимание

- ☞ В качестве аргумента используется «R-value reference» (`&&`) – ссылка на временный объект который мы можем изменить

## Перемещающий конструктор (C++11)

```
Stack::Stack(Stack&& a) : top(a.top),v(a.v) { // move
    a.top = 0;      // clear "old" object
    a.v = nullptr;
    std::cout << "Move ctor called " << *this << std::endl; // for debug
}
```

## Перемещающее присваивание (C++11)

```
Stack& Stack::operator=(Stack&& a) {
    if ( this == &a ) return *this; // protection against S = move(S);
    delete [] v; // exclude memory leak
    top = a.top; // move
    v = a.v;
    a.top = 0;    // clear "old" object
    a.v = nullptr;
    std::cout << "Move operator= " << *this << std::endl; // for debug
    return *this;
}
```

## Копирующий обмен

```
void swap_copy(Stack& a,Stack& b){  
    Stack tmp(a);  
    a = b;  
    b = tmp;  
}
```

## Перемещающий обмен

```
void swap_move(Stack& a,Stack& b){  
    Stack tmp(move(a));  
    a = move(b);  
    b = move(tmp);  
}
```

- ☞ `std::move()` компилятор выполняет преобразование типа к R-value:  
объект «становится временным», без `move()` будет копирование

```
Copy ctor called 1 2 3  
Copy operator= called 8 9  
Copy operator= called 1 2 3
```

### ➊ test swap

```
Stack X {1,2,3};  
Stack Y {8,9};  
swap_copy(X,Y);  
swap_move(X,Y);
```

```
Move ctor called 8 9  
Move operator= 1 2 3  
Move operator= 8 9
```

## ② test Reverse

```
Stack S {1,2,3};  
Stack rS( Reverse(S) );  
S = Reverse(rS);
```

### компиляция по умолчанию

```
Copy ctor called 1 2 3  
Copy ctor called 3 2 1  
Move operator= 1 2 3
```

### -fno-elide-constructors

```
Copy ctor called 1 2 3  
Move ctor called 3 2 1  
Move ctor called 3 2 1  
Copy ctor called 3 2 1  
Move ctor called 1 2 3  
Move operator= 1 2 3
```

- в ранних версиях C++ исключение ненужных операций копирования не гарантируется
- в C++11 используются «легкие» move-функции
- в C++17 «исключение ненужного копирования» — часть стандарта

# Конструктор как преобразователь типа

- Конструктор с одним аргументом задает преобразование:  
(type of argument) → (type of class)

```
Stack::Stack(int max) : top(0) {  
    v = new float[std::max(20,max)];  
    std::cout << " capacity=" << max << std::endl; // for test  
}  
Stack S(100); // вызов конструктора, OK!
```

... возможно нежелательное поведение

```
Stack S1 = 100; // Stack S1=Stack(100): int->Stack: capacity=100  
Stack S2 = 23.5; // double->int->Stack: capacity=23  
Stack S3(13.5); // double->int->Stack: capacity=20
```

# explicit **конструктор**

## explicit

- Спецификатор **explicit** в декларации конструктора запрещает неявные (**implicit**) преобразования:

public:

```
    explicit Stack(int max);
```

```
Stack S1(100);    // OK explicit
Stack S1 = 100; // ERROR: not explicit
Stack S2 = 13.5; // ERROR: conversion from 'double'
Stack S3(13.5);  // still OK, but we can 'delete' Stack(double) ctr!
```

- в C++98 **explicit** используется только в конструкторах
- в C++11 **explicit** можно использовать и для операторов преобразования типа

# Оператор преобразования типа (type casting)

преобразования CLASS → type

- функция член класса вида: operator type()

☞ возвращаемое значение не указывается

Пример: преобразование Stack → int

```
public:  
    explicit operator int() const {return top;}
```

```
Stack S11 {5,6,7};           // list initializer C++11  
int a = S11;           // ERROR: implicit conversion  
int b = int(S11);          // b = 3: explicit conversion  
int c = static_cast<int>(S11); // c = 3: cast conversion
```

☞ explicit operator type() – запрет неявного преобразования в C++11

# Конструктор с инициализацией списком (C++11)

- `auto il = {1,2,3};` создает объект с типом `initializer_list<int>`
- Похож на вектор, но только на чтение

`initializer_list` удобно использовать как аргумент конструктора:

```
#include <initializer_list> // for initializer_list<> type
Stack::Stack(std::initializer_list<float> il) { // ctor
    v = new float[std::max(int(il.size()),20)];
    top = 0;
    for (auto a : il) { v[top++]=a; }
}
Stack Sli {1,2,3,4,5};
cout << " Sli= " << Sli << endl; // Sli= 1  2  3  4  5
```

## Приоритеты конструкторов при инициализации фигурными скобками

- ① Конструктор по умолчанию
- ② Конструктор со списком инициализации
- ③ «Обычные конструкторы»

```
Stack S1 {};           // Stack()  
Stack S2 {1,2,3};     // Stack(initializer_list)  
Stack S3 {50U};       // Stack(initializer_list) !  
Stack S4(50U);        // Stack(int)  
Stack S5 {"test"};    // Stack(char*), если есть
```

### Обратите внимание

Stack S1(); // ERROR: это не вызов конструктора!  
Stack S1{}; // OK!

# Делегирование конструкторов в C++11

- Можно вызвать один конструктор из другого, но рекурсия запрещена

## Пример:

```
struct A {  
    A(int a, double b, char* t) {...}      // common part  
    explicit A(int a):A(a,0,nullptr){}        // A(int)  
    explicit A(double b):A(0,b,nullptr){}      // A(double)  
    A(char* t,int n):A(0,0,t){...}            // others...  
};
```

- ✓ Такую же функциональность можно получить и в C++98 с помощью функции «инициализации» и вызова ее в конструкторах

# Статические члены класса

- ☞ Переменные, объявленные внутри класса как `static`, являются глобальными переменными общими для всех объектов класса

```
struct Test {  
    static int counter;           // static member: counter for objects  
    Test(){counter++;}  
    Test(const Test&){counter++;}  
    ~Test(){counter--;}  
};
```

- ☞ Статические члены должны быть явно объявлены и определены в глобальной области видимости вне функций
- ☞ В момент объявления статических членов объектов класса еще нет и доступ осуществляется в виде: `class-name::identifier`

```
int Test::counter = 0; // определение и задание значения: без static
int main() {
    Test a;                  // default ctor
    Test* b = new Test(); // default ctor
    cout << " Test::counter = " << Test::counter << endl; // 2
    Test& c = a;            // reference - no new object
    cout << " Test::counter = " << Test::counter << endl; // 2
{
    Test e = c;              // copy ctor
    cout << " e.counter = " << e.counter << endl; // 3
}
                                // end of life for 'e'
cout << " Test::counter = " << Test::counter << endl; // 2
delete b;                   // end of life for 'b'
cout << " Test::counter = " << Test::counter << endl; // 1
}
```

☞ Если объект **object** создан, то доступ к статическим членам возможен и в обычном виде: **object.identifier**

## static const member and inline variables in C++17

inside of struct Test

```
static const int N = 100;      // C++98 only 'int' can be defined
static const double Pi;        // C++98
static const std::string Str; // C++98
static constexpr double Pi_11 = 3.142; // C++11: constexpr
static inline double Pi_17 = 3.1416; // C++17: inline variable
static inline std::string Str_1="Str1"; // C++17: inline variable
```

global definition in C++98

```
// const int Test::N = 100;
const double Test::Pi = 3.14;
const string Test::Str("Str");
```

print test

```
cout << " Test::N= " << Test::N << endl;           // 100
cout << " Test::Pi= " << Test::Pi << endl;         // 3.14
cout << " Test::Str= " << Test::Str << endl;        // Str
cout << " Test::Pi_11= " << Test::Pi_11 << endl; // 3.142
cout << " Test::Pi_17= " << Test::Pi_17 << endl; // 3.1416
cout << " Test::Str_1= " << Test::Str_1 << endl; // Str1
```

# Статические функции члены класса

Функция, объявленная в классе как static:

- статическая функция может быть вызвана даже если нет объектов класса: `class-name::function-name()`
- в такой функции отсутствует указатель `this`, поэтому в ней можно использовать только статические члены и функции класса

```
struct Test {  
    Test(){counter++;}  
    ~Test(){counter--;}  
    static int get_counter() {return counter;} // static function  
private:  
    static int counter; // теперь counter в private  
};
```

```
int Test::counter = 0; // определение и задание значения
int main() {
    cout << Test::get_counter() << endl; // 0
    Test a;
    Test* b = new Test();
    Test& c = a;
    cout << Test::get_counter() << endl; // 2
}
```

### Обратите внимание:

- ☞ Инициализация статической переменной всегда одинаково и не зависит от секции public/private
- ☞ тип указателя на статическую функцию класса отличается от типа указателя на обычную функцию класса
- ☞ статическая функция не может быть `virtual`, `const`, `volatile` или переопределяться другой функцией класса

# Личные конструкторы и деструктор. Singleton

## Зачем это может быть нужно?

- когда **все конструкторы в private** создание объектов запрещено всюду кроме как в статических или дружественных функциях
- поместив деструктор в **private** мы запрещаем удаление ранее созданных объектов

## Пример применения – класс «одиночка»

- ✓ **Singleton** – класс позволяющий создать ровно один объект
- ✓ обычно этот объект существует до конца выполнения программы

## Пример реализации

```
class Single {  
public:  
    static Single* Instance(); // basic function for creating  
                            // a single object of this class  
    void set_sg(int i) {sg = i;} // Some useful functions  
    int get_sg() const {return sg;}  
private:  
    Single() = default; // the only constructor for internal use  
    Single(const Single&) = delete;  
    Single& operator= (const Single&) = delete;  
    ~Single() = delete;  
  
    static Single* p_instance; // global static pointer for  
                            // a single instance of a class  
    int sg=0;                // another variable(s)  
};
```

## Статическая функция создающая ровно один объект

```
Single* Single::p_instance = nullptr;

Single* Single::Instance() {
    if( !p_instance ) p_instance = new Single();
    return p_instance;
}
```

### Комментарии

- ✓ Первый вызов `Single::Instance()` приводит к созданию объекта и сохраняет указатель на него в `Single::p_instance`
- ✓ Следующие вызовы `Single::Instance()` приводят к возврату этого указателя на уже существующий объект

## Пример использования:

```
int main() {
    // Single* a = new Single; // ERROR: Single::Single() is private
    Single* a = Single::Instance(); // единственный способ получить объект
    a->set_sg(15); // сохраняем данные
    cout << a->get_sg() << endl; // 15
    testfun();
    // delete a; // ERROR: use of deleted Single::~Single()
    cout << "test of move-ctor: " << std::boolalpha
        << is_move_constructible<Single>::value << '\n'; // false
}

void testfun() {
    Single* t = Single::Instance(); // для доступа к данным
    cout << "testfun: " << t->get_sg() << endl; // 15
}
```

👉 Имеется ровно один объект который живет до конца программы