Концепция перемещений (move) (C++11)

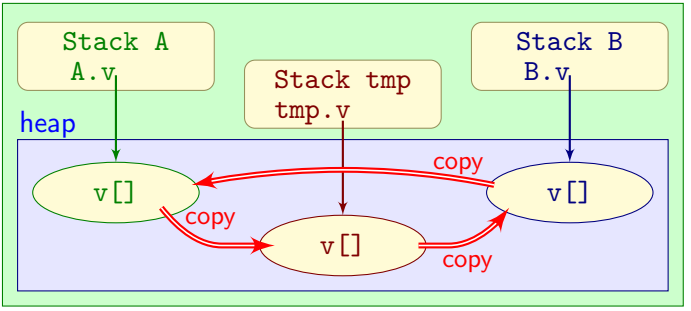
Мотивация или для чего это нужно?

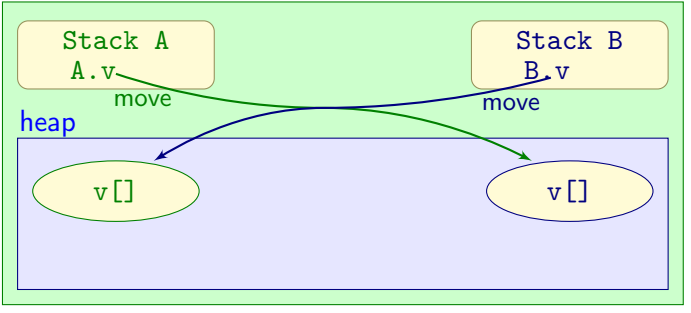
Пример ❶ : функция обмена для двух стеков

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

☞ Используется дополнительный буфер (tmp) и выполняется ненужное копирование (см. следующий слайд)

☞ Возможное решение – специальная функция (`friend of Stack`) для перенаправления указателей без копирования данных


Stack A
A.v
Stack tmp
tmp.v
Stack B
B.v
heap
v[]
copy
copy
v[]
copy
v[]


Stack A
A.v
Stack B
B.v
move
move
heap
v[]
v[]

Пример ❷ : функция возвращает «большой объект»

```
Stack Reverse(Stack s) {
  Stack rs;
  while ( !s.isEmpty() ) {
    rs.push(s.pop());
  }
  return rs;
}
...
Stack rS( Reverse(S) ); // copy ctor
rS2 = Reverse(S);       // assignment operation
```

Дополнительные, ненужные расходы:

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

Перемещающие конструктор и присваивание

☞ В стандарт C++11 введены перемещающий конструктор и перемещающий оператор присваивания оптимизирующие управление объектами

Две новые обязательные «функции»:

- move constructor: `Class(const Class&&);`
- move assignment: `Class& operator=(const Class&&);`

☞ Если в классе эти функции отсутствуют, компилятор создаст для них свои версии

Обратите внимание

☞ В качестве аргумента используется «R-value reference» (`&&`)

R-value references (C++11)

- R-value ссылка ссылается всегда на что-то, что может стоять только справа (R-value), но позволяет модифицировать его значение

```
int&& rref1 = sqrt(25);        // sqrt(25) is the R-value
rref1 += 1;                    // OK!
int&& rref_i = i;              // i is L-value
```

- «Обычные» ссылки теперь называют L-value references

```
int& ref_i = i;                // L-value reference
const int& ref_c = sqrt(25);   // L-value const reference on R-val
```

Пример: L- и R- ссылки в функциях

```
void fun(int& x)  { cout << " fun(int&) x= " << x << endl; }
void fun(int&& x) { cout << " fun(int&&) x= " << x << endl; }

int i {1}, j {2};
fun(i);   // fun(int&) x= 1
fun(i+j); // fun(int&&) x= 3
```

Перемещающий конструктор (C++11)

```
Stack::Stack(Stack&& a) : top(a.top),v(a.v) {
  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;
  a.v = nullptr;
  std::cout << "Move operator= " << *this << std::endl; // for debug
  return *this;
}
```

Тестируем как это оаботает на примере 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
```

g++ -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
```

- Компилятор "обычно" оптимизирует код так что «ненужные» операции копирования или перемещения исключаются

☞ Стандарт C++ не гарантирует такую оптимизацию

- C++11 использует оптимальные «операторы move»

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()` выполняет преобразование типа к R-value, явно указывая компилятору, что объект временный: без `move()` будет копирование

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

Test:

```
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
```

Конструктор как преобразователь типа

- Конструктор который может быть вызван с одним аргументом задает преобразование: `(type of argument)` $\rightarrow$ `(type of class)`

Пример

```
Stack::Stack(int max) : top(0) {
  v = new float[std::max(20,max)];
  std::cout << " capacity=" << max << std::endl; // for test
}
Stack S1 = 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 Se(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 (C++11)

Конструктор с инициализацией списком (C++11)

- `auto il = {1,2,3};` создает объект с типом `initializer_list<int>`
- Похож на вектор, но в нем хранятся константные объекты

☞ Надо указывать заголовочный файл `<initializer_list>`

`initializer_list` удобно использовать как аргумент конструктора:

```
#include <initializer_list>
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
```

Какие конструкторы вызываются при инициализации с { }?

```
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
```

Приоритеты вызова конструкторов:

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

Обратите внимание

~~`Stack S1();`~~ // ☞ ERROR: это не вызов конструктора!

Делегирование конструкторов (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 создания функции «инициализации» и вызова ее из конструкторов

Инициализация внутри класса (C++11)

In-class member initializers

☞ Для любой переменной класса можно указать значение, которое будет использовано при инициализации, если конструктор не инициализирует эту переменную в явном виде

Пример

```
class Stack {
  Stack() {v = new float[capacity];}
  explicit Stack(int cap) : capacity(cap) {...}
...
private:
  double*       v = nullptr;
  int           top = 0;
  int           capacity = 20;
};
```

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 определенны 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`

Статические функции-члены

Функция, объявленная в классе как `static`:

- статическая функция может быть вызвана даже если нет объектов класса
- 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:

- ☞ Определение статической переменной всегда одинаково и не зависит от секции public/private
- ☞ статическая функция не может быть `virtual`, `const`, `volatile` или переопределяться функцией членом класса
- ☞ адрес статической функции можно сохранить в обычном указателе на функцию и отличается от указателя на функцию класса

Личные (запрещенные) конструкторы и деструктор

Зачем это может быть нужно?

- когда все конструкторы в `private`, создание объектов запрещено всюду кроме статических или дружественных функций
- поместив деструктор в `private` мы запрещаем удаление ранее созданных объектов

Пример применения – класс «одиночка» (singleton)

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

Пример реализации «одиночки»

```
class Single {
  public:
    static Single* Instance(); // basic function for creating
                               // a single object of this class
    Single() = delete;
    Single(const Single&) = delete;
    Single& operator= (const Single&) = delete;
    ~Single() = delete;

    void set_sg(int i)  {sg = i;}
    int  get_sg() const {return sg;}
  private:
    static Single*  p_instance; // global static pointer for a single
                                //  instance of a class
    int sg;       // 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`
- ✓ Последующие вызовы `Single::Instance()` приводят к возврату этого указателя на уже существующий объект

Пример использования:

```
void testfun() {
  Single* t = Single::Instance(); // указатель на объект
  cout << "testfun: " << t->get_sg() << endl; // доступ к данным
}

int main() {
//  Single* a = new Single;       // Single::Single() is private
  Single* a = Single::Instance(); // единственный способ создать объект
  a->set_sg(15);                  // сохраняем данные
  cout << a->get_sg() << endl;    // 15
//  delete a;                     // use of deleted Single::~Single()
  testfun();                      // testfun: 15
}
```

- ☞ Все конструкторы и оператор присваивания в `private`: нельзя ни создать еще один новый объект ни удалить имеющийся
- ☞ Созданный объект «живет» до конца программы