при объявлении

ИНИЦИАЛИЗАЦИЯ ПЕРЕМЕННЫХ КЛАССА

До стандарта C++11

```
class A{
        int m_a=0; //???
        …
};
```

Исключение???

Инициализация переменных класса C++11:

```
class A{
        int m_a=0; //OK
public:
        //конструкторов нет
};

A a;     //a.m_a =0;
```

Инициализация переменных класса C++11:

```
class A{
        int m_a=0;
public:
        A(int a): m_a(a) {}
};

A a;     //ошибка – нет default конструктора
A a1(1); //OK – инициализация игнорируется
```

Инициализация членов класса - альтернатива – значениям параметров конструктора по умолчанию

```
class A{
        int m_a = 0;
public:
        A() {}
        A(int a) :m_a(a){}
};
A a1; // m_a=0
A a2(33); // m_a=33
```

Отличия инициализации и присваивания (компилятор генерирует **разный** код)

```
class A{
        int m_a=0;
public:
        A(int a): m_a(a) {}
};
```

A a(1); //инициализация нулем игнорируется

```
class A{
        int m_a=0;
public:
        A(int a) { m_a=a; }
};
```

A a(1); //1. инициализация нулем
2. присваивается a

Стандарт C++11-C++14

```
class A {
        int m_a = 0;
        int m_b{ 55 };
        char m_name[10] = "abc";
        std::string m_str2 = std::string("abc");
        std::string m_str = "abc";
        std::vector<int> m_v1 = std::vector<int>(5, 33);
        std::vector<int> m_v2{1,2,3};
        const char* p1 = "abc";
        char* p2 = new char(0);
        char* p3 = new char[5]{ 'a', 'b', 'c' };
};
```

Специфика использования

```
class A{
        int m_x = 0;
        int m_y = 0;
public:
        A(int x) :m_x(x){}      //m_y = 0
};
```

Наследование

OVERRIDE И FINAL

Проблемы. Неудобство:

```
class A {
…
 public:
 virtual void f(int);
};
class B : public A {
 public:
 [virtual] void f(int); //опционально
};
```

Проблемы. Ошибка программиста:

```
class A {
…
 public:
 virtual void f();
};
class B : public A {
 public:
 virtual void f(int); //???
};

A* p = new B;
p->f(5); //???
```

Реальная проблема:

```
class A {
…
 public:
 virtual void f(int =0);
};
class B : public A {
 public:
 virtual void f(); //???
};

A* p = new B;
p->f(); //???
```

Реальная проблема:

```
class A {
…
 public:
 virtual void f();
};
class B : public A {
 public:
 virtual void f() const; //???
};

A* p = new B;
p->f();
```

Реальная проблема:

```
class A {
…
 public:
void f(); //НЕ виртуальная!
};
class B : public A {
 public:
 virtual void f(); //???
};

A* p = new B;
p->f();
```

А еще можно просто «опечататься» в имени функции…

```
class A {
public:
 virtual void func();
};
class B : public A {
 public:
 virtual void fync(); //???
};

A* p = new B;
p->func(); //??? И можно дооолго искать ошибку!
```

Решение:

- **override** – указание компилятору, что метод является переопределением виртуального метода базового класса
- **final -** производный класс не может переопределять виртуальный метод базового

Следствие использования override:

```
class A {
 public:
 virtual void f();
};
class B : public A {
 public:
 virtual void f(int) override; // error C3668: 'B::f' : method with
        override specifier 'override' did not override any base class methods
};
```

Следствие использования override:

```
class A {
…
 public:
 virtual void f();
};
class B : public A {
 public:
 virtual void f() const override; //???
};
```

Следствие использования override:

```
class A {
…
 public:
void f(); //НЕ виртуальная!
};
class B : public A {
 public:
 virtual void f() override; //???
};
```

final

- виртуальный метод
- класс

Полезно, когда:

- нельзя позволять создавать производные классы
- нельзя производным классам перегружать виртуальный метод класса-предка

Следствие использования final:

```
class A {
 public:
            virtual void f();
};

class B : public A {
public:
           virtual void f() [override] final;
};

class C:public B{
virtual void f() [override] ; //B::f': function declared as 'final' cannot be
                                                    overridden by 'C::f'
};
```

final для запрета наследования

```
class A final {};
class B : public A {}; // ошибка!
```

final для запрета наследования
Другой способ:

```
class A {
…
public:
        virtual ~A() final;
        virtual void f()=0 final;
};
class B : public A {}; // ошибка!
```

DEFAULT И DELETE

Компилятор может сгенерировать автоматически:

- default конструктор (если в классе не объявлен любой пользовательский конструктор)
- конструктор копирования
- move конструктор копирования
- деструктор
- оператор присваивания
- move оператор присваивания

C++11 – Посредством =default

можно предписать компилятору автоматически сгенерировать:

- default конструктор
- конструктор копирования
- move конструктор копирования
- operator=
- move operator=
- деструктор

Какие важные действия компилятор при этом может сгенерировать автоматически???

Использование **default**:

```
class A{
int m_a;
public:
        A(int a=0);
 };

class B{
                A m_A;
public:
        B(int a): m_A(a){}
        B() = default;
 };
 int main()
 {
                B b; //???
 }
```

default – при наследовании (на примере классического конструктора копирования)

```
class A{…};
class B:public A{…};
```

A - тривиальный	**B - тривиальный**
A-тривиальный	**B - нетривиальный**
A-нетривиальный	**B - тривиальный**
A-нетривиальный	**B - нетривиальный**

default при внедрении:

```
template<typename Key, typename Value> class Pair {
        Key m_k;
        Value m_v;
public:
      Pair(const Key& k, const Value& v) :m_k(k), m_v(v) {}
//какие методы компилятор может сгенерировать а)автоматически и б) корректно?
};
```

Использование =default

```
template<typename Key, typename Value> class Pair {
        Key m_k;
        Value m_v;
public:
        Pair(const Key& k, const Value& v) :m_k(k), m_v(v) {}
        Pair() = default;
        Pair(const Pair&) = default;
        Pair& operator=(const Pair&) = default;
        Pair(Pair&&) = default;
        Pair& operator=(Pair&&) = default;
        ~Pair() = default;
};
```

delete используется для:

- запрещения вызова метода (при этом определение метода не требуется)
- запрет вызова метода с другим типом параметра – запрещает компилятору осуществлять неявное приведение типа параметра

спецификатор delete

а) **???**

```
class A{
private:
        A(const A& ){}
        A& operator=(const A& ){}
};
```

б)

```
class A{
public:
        A(const A& ) = delete;
        A& operator=(const A& ) = delete;
};
```

delete для предотвращения неявного преобразования типа параметра

```
class A
{
public:
    A(long long); // OK
    A(long) = delete; // невозможно вызвать с char, short, int, long
};
```

Или так:

```
class A
{
public:
    A(long long);
    template<typename T>  A (T) = delete; // запрет
                                  на все остальные типы
};
```

В общем случае любой метод может быть delete

```
class A {
public:
        void f(double) = delete;
        void f(int) {…};
};

        A a;
        a.f(2.2); //???
        a.f(2); //???
```

Запрет на инстанцирование шаблонной функции

```
template<typename T> void pf(T * ptr) {…}

template<>
void pf<const char >(const char * ptr) = delete;

int main()
{
        int n = 1;
        pf(&n); //Ok
        pf("abc"); //ошибка
}
```

Запрет на динамическое создание объектов

```
class A{
…
 public:
void *operator new(std::size_t) = delete;
void *operator new[](std::size_t) = delete;
};

A* p = new A; // ошибка!
```

Понятие

RVALUE REFERENCE

lvalue и rvalue:

Для lvalue - всегда можно получить **адрес,** который программист может использовать на всем протяжении жизни объекта:

```
int v = <выражение>; //???
int* p1 = &(++v); //???
int* p2 = &(v++); //???
ar[i] = <выражение>; //???
int& f1();   f1() = 33; //???
int& v1 = 1; //???
int* p3= &(x+y);    //???
A* p4 = &A(); //???
int f2();   f2() = 44; //???
```

Что делает в каждом случае компилятор? Специфика

```
void f1(int);                f1(1); //???
void f2(int&);               f2(1); //???
void f3(const int& );        f3(1); //???

int f4(){return 1;}          int n=f4();
int& f5(){return 1;} //???   int& n=f5();
const int& f6() {return 1;}  const int& n3=f6();
```

согласно пункту 12.2/5 стандарта C++, время жизни временного объекта продлевается и становится таким же, как и время жизни ссылки, которая указывает на этот объект. Выражение же, в свою очередь, приобрело свойство lvalue

Итог: до C++11

- const T& - для передачи адреса временного объекта
- но! различия между временным и постоянным объектами сделать было **НЕВОЗМОЖНО**!

```
void f(const A&);
int main(){
        A a1;  f(a1);
        f(A());
}
```

C++11

- **lvalue**
- **rvalue** (до C++11) **/prvalue** (pure rvalue – C++11)
- **xvalue** (eXpiring) – временный объект, который будет гарантированно «уничтожен» в «ближайшее время» компилятором. Для обозначения таких объектов используются *rvalue ссылки*

lvalue - можно получить адрес, пока существует сам объект!

Для любого **именованного** объекта всегда можно получить его адрес!

```
int global=1;
void f1(int* );

void f(int n) {
        f1(&global);
        f1(&n); //???
        int m=22;
        f1(&m); //???
}
```

prvalue – нельзя явно получить адрес!

- литерал (кроме строкового литерала) – **1**, **true**, **nullptr**
- значение, возвращаемое функцией по значению, например

  A f();

  int main(){ A* p = &(f()); //ошибка }
- адрес автоматического неименованного объекта

  void f(A*);

  int main(){ f(&A()); //по стандарту ошибка }

xvalue

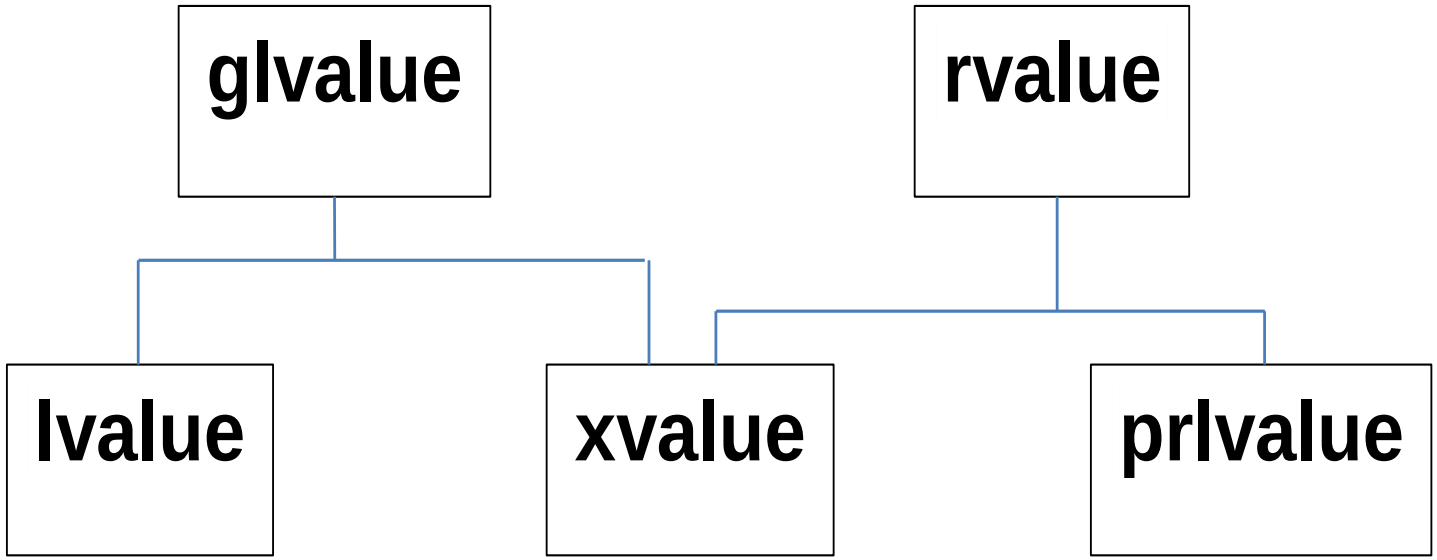
- временный объект, для которого в общем случае нельзя получить адрес, а в **некоторых** случаях можно

```
void f(const A& a) {const A* p = &a; //???}
int main(){  f(A()); }
```

- локальные объекты, время жизни которых в ближайшем будущем закончится!

```
A f() { A tmp; … return tmp;}
```

Для упорядочения понятий вводятся:



Как различить?

```
int main()
{
        A a;
        f(a); //ссылка на относительно постоянный объект
        f(A()); //ссылка на временный объект
}
```

&& - rvalue reference

- T& (const T&) - **lvalue reference**– переменная содержит адрес относительно постоянного объекта
- T**&&** (const T&&) - **rvalue reference** - переменная содержит адрес относительно временного объекта

Демонстрация отличия lvalue и rvalue

```
A&  r1 = A();  // ошибка!
A&& r2 = A();  // Ok
```

Цель введения &&

- **Повышение эффективности** за счет использования **семантики перемещения** (позволяет различать временные и постоянные объекты):
  - Классическое (глубокое) копирование/присваивание
  - Копирование/присваивание посредством перемещения (move)
  - В общем случае передача в **любую функцию**!
- **Решение проблемы forwarding (forwarding references)**: передача обобщенных параметров шаблона таким универсальным образом, чтобы:
  - их квалификаторы (cv-qualifiers) оставались неизменными
  - и категории (lvalue, rvalue) тоже

=> конструкция T&& используется:

- Для обозначения **временных** объектов => позволяет избежать ненужного копирования
- Для обозначения **универсального параметра шаблона** (с возможностью дальнейшего восстановления типа в зависимости от реально передаваемого значения) – обеспечивает возможность идеальной передачи актуального типа

Перемещающие операции.

Цель – избавиться от ненужных копий!

MOVE-СЕМАНТИКА (DESTRUCTIVE READ)

Заготовка для демонстрации семантики перемещения

```
class MyString{
        char* m_pStr;
public:
        … //???

};
```

Требуют вмешательства программиста:

```
class MyString{
        char* m_pStr;
public:
        MyString(const char*="default");

        ~MyString(){ delete[] m_pStr;}

        MyString(const MyString& ); //классический конструктор копирования

        MyString& operator=(const MyString& ); //классический оператор
                                                присваивания
        …
};
```

```
MyString::MyString(const char* p){
        m_pStr = new char[strlen(p)+1];
        strcpy(m_pStr,p);
}

MyString::MyString(const MyString& other) {
        m_pStr = new char[strlen(other.m_pStr)+1];
        strcpy(m_pStr,other.m_pStr);
}

MyString& MyString::operator=(const MyString& other){
        if(this != &other)
        {
                delete[] m_pStr;
                m_pStr = new char[strlen(other.m_pStr)+1];
                strcpy(m_pStr,other.m_pStr);
        }
        return *this;
}
```

Затраты при использовании классического конструктора копирования для нетривиальных классов

```
MyString f()
{
        MyString s;
        //формирование строки
        return s; //???
}//???
```

move конструктор копирования

```
class MyString{
        char* m_pStr;
public:
        …
        MyString(const MyString& ); //классический
                                    конструктор копирования
        MyString(MyString&& ); //перемещающий
                                  конструктор копирования
        …
};
```

Реализация move конструктора копирования

```
MyString::MyString(MyString&& other)
{
        m_pStr = other.m_pStr;
        other.m_pStr=nullptr;
}
```

деструктор other???

Эффективность

```
MyString f()
{
        MyString s;
        //формирование строки
        return s; //???
}//???
```

Замечание:

перемещающими могут быть:

- любые другие операции (не только конструктор копирования и оператор=)
- не только методы класса, но и глобальные функции

std::move()

```
MyString s1(“abc”);
MyString s2 (“qwerty”);
s2= std::move(s1);
//s2 = ???
//s1 = ???
```

Предупреждение!

std::move() может отменить оптимизации!

```
MyString ProbaMove(){
        MyString tmp(“aaa”);
        return tmp;
}

int main(){
        MyString s1("bbb");
        s1 = ProbaMove();  //???

        MyString s2 = ProbaMove(); //???
        MyString s3 = std::move(ProbaMove()); //???
}
```

Что такое std::move() ?

- Не перемещает!
- А приводит тип (cast)

Шаблон move
#include <utility>

```
template<typename T> typename remove_reference<T>::type&&
        constexpr std::move(T&& a) noexcept
 {
 typedef typename
        remove_reference<T>::type&& RvalRef;
 return static_cast<RvalRef>(a);
}
```

Эквивалентно записи попроще:

```
X&& std::move(X& a) noexcept {
                    return static_cast<X&&>(a); }
```

Смысл std::**move** **- безусловно «приводит» к rvalue**

Оператор присваивания

MyString s1(“abc”), s2(“qwerty”);

s1=s2; **//???**

s1= MyString(“Hello”); **//???**

s1 = “Bye”; **//???**

move оператор присваивания

```
class MyString{
        char* m_pStr;
public:
        …
        MyString& operator=(const MyString& );
                        //классический   operator=
         MyString& operator=(MyString&& );
                        //перемещающий operator=

        …
};
```

Реализация move operator=

```
MyString& MyString::operator=(MyString&& other)
{
        delete[] m_pStr;
        m_pStr = other.m_pStr;
        other.m_pStr=nullptr;
        return *this;
}
```

Деструктор other ???

Реализация move operator= поинтереснее

```
void MyString::swap(MyString& other)
{
        char* p = m_pStr;
        m_pStr = other. m_pStr;
        other. m_pStr = p;
}
MyString& MyString::operator=(MyString&& other)
{
        swap(other);
        return *this;
}
```

Деструктор other **???**

Разница ?

MyString s1(“abc”), s2(“qwerty”);

s1 = s2; **//???**

s1= MyString(“Hello”); **//???**

s1 = std::move(s2); **//???**

Важно!!!

- Если явно определен классический конструктор копирования, то компилятор перестает автоматически генерировать move
- Если явно определен move конструктор копирования, автоматический классический конструктор копирования становится delete => вызывать его компилятор не разрешит => ошибка
- можно попросить компилятор сгенерировать автоматический move конструктор копирования (=default)

Аналогично operator=

Правила выбора компилятором перегруженного варианта функции:

- если перемещающая операция (T&&) не реализована, то компилятор вызывает (const T&)

```
void f(const A&);
int main()
{
        f( A() ); //???
}
```

Модификация контейнеров STL

Добавлены:

- move конструктор копирования, например vector
- move operator=

- методы типа push(), insert()

Псевдокод:

```
template<typename T> class stack{
…
public:
        void push(const T&){…}
        void push(T&&){…}
};
```

Бьерн Струструп:

“To move,

or

not to move?”

std::move_if_noexcept

Возвращает то же, что move(arg), но приводит тип возвращаемого значения:

- или к T&&
- или к const T&

в зависимости от свойств T!

=> приведет к перемещению объекта только, если перемещающая операция гарантированно не генерирует исключений.

Как компилятор использует noexcept

Важно!

- Спецификация **noexcept –** это способ для программиста предоставить компилятору информацию о том, что функция не будет генерировать исключения
- Получив информацию о том, что функция безопасна, компилятор может использовать **дополнительные оптимизации**
- Например, некоторые контейнеры (vector) поддерживают перемещение элементов только в том случае, когда **move конструктор noexcept**, а в противном случае вызывают глубокое копирование

Специфика noexcept

- Спецификатор времени компиляции noexcept значительно уменьшает размер итогового файла и ускоряет работу программы.
- => полезно его использовать «по делу»!!!
- если функция со спецификатором noexcept все-таки сгенерирует исключение, то будет вызвана std::terminate() => аварийное завершение.

Демонстрация

```
struct Bad{
    Bad() {}
    Bad(Bad&&); // из сигнатуры не следует, что функция безопасна
=> компилятор исходит из того, что может генерировать исключения
    Bad(const Bad&); // аналогично
};

int main()
{
    Bad b;
    Bad b2 = std::move_if_noexcept(b); // Bad(const Bad&)
}
```

Демонстрация

```
struct Good{
   Good() {}
   Good(Good&&) noexcept; // не генерирует исключений!
   Good(const Good&) noexcept; // тоже не генерирует исключений!
};

int main()
{
   Good g;
   Good g2 = std::move_if_noexcept(g); // Good(Good&&)
}
```

Правило.
Компилятор воспринимает:

- **Любую именованную rvalue-ссылку как lvalue** (так как для именованной ссылки всегда можно получить адрес)
- **Любую неименованную rvalue-ссылку как rvalue**

=>

```
void f(A&& a) {/*»a» превращается при использовании внутри функции в lvalue*/}
int main()
{
        A a;
        f(a); //ошибка! если нет перегруженной void f(const A&);
        f(A()); //неименованный объект => rvalue
}
```

move – семантика при внедрении (неэффективный вариант)

```
class X{
        …
public:
X& operator=(const X& other)  ;
X& operator=(X&& other) ;
};

class WrapX{
        X m_X;
public:
  WrapX(const X& rx):m_X(rx){}
  WrapX& operator=(const WrapX&)  = default;
  WrapX& operator=(WrapX&& other) {// другие действия
        m_X = other.m_X; // !!!вызывается X& operator=(const X& other)  ;
        return *this;
        }
};
```

move – семантика при внедрении (эффективный вариант)

```
class X{
public:
X& operator=(const X&)  ;
X& operator=(X&&) ;
};

class WrapX{
        X m_X;
public:
        WrapX(const X& rx):m_X(rx){}
        WrapX& operator=(const WrapX&) =default;
        WrapX& operator=(WrapX&& other){
          …
          m_X = std::move(other.m_X); // !!!вызывается X& operator=(X&&)
          return *this;
        }
};
```

move – семантика при наследовании (неэффективный вариант)

```
class X{ …
public:
X(const X& other);
X(X&& other);
X& operator=(const X& other) ;
X& operator=(X&& other) ;
};

class Y:public X{
public:
Y(Y&& other):X(other){…}

Y& operator=(Y&& other){
            X::operator=(other);
            …
            return *this;
            }
};
```

move – семантика при наследовании (эффективный вариант)

```
class X{ …
public:
…
X(const X& other);
X(X&& other);
X& operator=(const X& other) ;
X& operator=(X&& other) ;
};

class Y:public X{
public:
Y(Y&& other):X(std::move(other)){…}
Y& operator=(Y&& other) { X::operator=(std::move(other)); return *this;}
};
```

move и lambda-функции – C++14

```
class A {
public:
        A() {}
        A(const A&) {}
        A(A&&) {}
        A& operator=(const A&) {}
        A& operator=(A&&) {}
};

A a1;
[a1]() { … }(); //???
[&a1]() { … }(); //???
[a = std::move(a1)]() { … }(); //???
```

[C++14] move и “=default”

```
class A {…
public:
        A() = default;
        A(const A& other) {…} //hand made
        A(A&&) {…} //hand made
};

class B {
        A m_A;
public:
        B() = default;
        B(const B&) = default;
        B(B&&) = default;
};

B b1;
B b2 = b1;
B b3 = std::move(b1);
```

Можно запретить копирование, а оставить только перемещение:

```
class A {
public:
        A(const A&) =delete;
        A(A&&) {…}
};
```

Пример

```
class A{
        …
public:
        …
A(const A&) = delete;
A(A&&);
A& operator=(const A&) = delete;
A& operator=(A&&);
};

A a1;
A a2(a1);//???
A a3 = std::move(a1); //???
a1 = a3; //???
a1 = std::move(a3);
```

ССЫЛОЧНЫЕ КВАЛИФИКАТОРЫ ФУНКЦИЙ

Без квалификаторов

```
class REF {
public:
        void f() {}
};

int main(){
        REF r1;
        r1.f();
        REF().f();
}
```

Ссылочные квалификаторы функций

```
class REF {
public:
        void f() & {} //для lvalue
        void f() && {} //для rvalue
};

int main(){
        REF r1;
        r1.f();
        REF().f();
}
```

<iterator>

ПЕРЕМЕЩАЮЩИЕ ИТЕРАТОРЫ

std::move_iterator
#include <iterator>

- итератор-адаптер (обертка для базового input-итератора, который минимум позволяет x=*it)
  Возможная реализация:
  template <class Iterator> class move_iterator {
  Iterator current;
  …
  };
- обладает той же функциональностью, что и базовый, но
- вместо копирования и присваивания осуществляет перемещение

Основная функциональность:

- конструкторы:

move_iterator<тип_базового_итератора>(базовый_итератор)

- operator*
- operator++
- operator==
- …

Пример:

```
vector<string> v = {"www" ,"abc", "qwerty" };
typedef vector<string>::iterator IT;
set<string> s(move_iterator<IT>(v.begin()),
                        move_iterator<IT>(v.end()));
//v ???
```

Но!

```
set<string> s= { "abc", "qwerty", "www" };
typedef set<string>::iterator IT;
vector<string> v(move_iterator<IT>(s.begin()),
                 move_iterator<IT>(s.end()));
//???
```

шаблон std::make_move_iterator #include <iterator>

создает перемещающий итератор для заданного базового (аналогично конструктору):

```
vector<string> v = { "abc", "qwerty", "www" };
set<string>
        s(make_move_iterator(v.begin()),
          make_move_iterator(v.end()));
```

СВЕРТКА ССЫЛОК

Правило «свертки» ссылок

A& & -> A&

A&& & -> A&

A& && -> A&

A&& && -> A&&

Когда компилятор применяет свертку ссылок:

- при инстанцировании шаблонов
- вывод типа для auto-переменных
- задание псевдонимов – typedef

Демонстрация правила «свертки» ссылок на примере auto

int n = 1;

auto x = n; //int

auto& y = n; //int&

//auto & & = n; //ошибка – reference to reference are not allowed

auto && z = n; //(n - lvalue) => int& && -> **int&**

auto&& w = 1; // (1- rvalue) => int&& && -> **int&&**

«Снятие» ссылочности

Если параметр шаблона – ссылочный тип, то предоставляет псевдоним указываемого типа, иначе тип

template< class **T** > struct **remove_reference** {typedef T type;};

template< class T > struct **remove_reference<T&>** {typedef T type;};

template< class T > struct **remove_reference<T&&>** {typedef T type;};

Как работает **remove_reference**

```
std::remove_reference<A>::type x; //A
std::remove_reference<A&>::type y; //A
std::remove_reference<A&&>::type z; //A

A a(1);
std::remove_reference<A>::type& rx=a; //A&
std::remove_reference<A&>::type& ry = a; //A&
std::remove_reference<A&&>::type& rz = a; //A&

std::remove_reference<A>::type&& rrx = A(2); //A&&
std::remove_reference<A&>::type&& rry = A(3); //A&&
std::remove_reference<A&&>::type&& rrz = A(4); //A&&
```

Также <type_traits>

- std::is_reference
- std::is_lvalue_reference
- std::is_rvalue_reference

Перегружены:

- operator bool
- operator ()

Пример:

bool b=std::is_rvalue_reference<A>::value;

b=std::is_rvalue_reference<A&>::value ;

b=std::is_rvalue_reference<A&&>::value ;

FORWARD

Perfect forwarding

В первую очередь предназначен для уменьшения дублирующего кода

Экспериментальный класс:

```
class A {
   int m_a;
public:
   explicit A(int a=0):m_a(a){}
   A(const A& other) ;
   A(A&& other) ;
   A& operator=(const A& other) ;
   A& operator=(A&& other) ;
   ~A() ;
};
```

Независимо от типа действия требуется сделать одни и те же. Проблемы?

```
class A{…};
void f(A& a) { std::cout<<a; }

int main(){
    A a1;
    f(a1); //???

    const A a2;
    f(a2); //???

    f(A()); //???
}
```

Решение до C++11
Универсальное, но неэффективное

```
class A{…};
void f(const A& a) { std::cout<<a; }

int main(){
    A a1;
    f(a1); //???

    const A a2;
    f(a2); //???

    f(A()); //???
}
```

Решение до C++11
Эффективное, но текст дублируется

```
class A{…};
void f(const A& a) { std::cout<<a; }
void f(A&& a) { std::cout<<std::move(a); }

int main(){
    A a1;
    f(a1); //???

    const A a2;
    f(a2); //???

    f(A()); //???
}
```

А если требуется для объектов разного типа выполнить разные действия?

```
class A{…};
void f(const A& a) { std::cout<<a; }
void f(A& a) {++a;}
void f(A&& a) { std::cout<<(a+1); }

int main(){
    A a1;
    f(a1); //???

    const A a2;
    f(a2); //???

    f(A()); //???
}
```

Усугубляем ситуацию:

```
class A{…};
void f(const A& a) { std::cout<<a; }
void f(A& a) {++a;}
void f(A&& a) { std::cout<<std::move(a)); }

void wrap(??? a){ … f(a); …}

int main(){
    A a1;
    wrap(a1); //???

    const A a2;
    wrap(a2); //???

    wrap(A()); //???
}
```

Пояснение проблемы

void wrap(<**тип**> t) // тип параметра – любой из требуемых – A&, const A&, A&&

{

//действия, не зависящие от типа “t”

f(t); **//здесь должна быть вызвана соответствующая функция f()!!!**

// действия, не зависящие от типа “t”

} => сколько раз нужно перегрузить функцию **wrap()**?

Проблемы множатся **нелинейно**…

А если функция f() принимает не один параметр?

- два ,
- …

Нужно предусмотреть **все** сочетания разных типов => при росте числа параметров (N) количество перегруженных функций будет расти **нелинейно** ???

=> нужно обеспечить:

универсальную **перенаправляющую** функцию независимо от типа параметров

Идеальная перенаправляющая ф-ция (perfect fowarding function) **wrap(a1,a2,...,aN)**, которая вызывает **f(a1,a2,...,aN)**

должна удовлетворять следующим критериям:

- Для всех наборов a1,a2,...,aN, для которых запись f(a1,a2,...,aN) корректна(well-formed), запись wrap(a1,a2,...,aN) должна быть так же корректна.
- Для всех наборов a1,a2,...,aN, для которых запись f(a1,a2,...,aN) некорректна(ill-formed), запись wrap(a1,a2,...,aN) должна быть так же некорректна.
- Количество работы, которую придётся проделать для реализации такой идеально-перенаправляющей ф-ции f должно не более чем **линейно** зависеть от N.

T&& применительно к шаблонам

- принимает «все, что угодно» с сохранением категории (lvalue/rvalue) и cv-квалификаторов
- Обрабатывается согласно правилам:
  - вывода аргумента шаблона
  - свертывания ссылок

Смысл forward<> - **условный** cast:

для

template<typename T> void wrap(T&& t){…}

- Если функция вызывается для lvalue, то результирующий тип будет **A&**
- Если функция вызывается для rvalue, то результирующий тип будет **A&&**

Возможная имплементация

```
template<typename T> T&&
        forward ( typename
        remove_reference<T> : : type& param)
{
        return static_cast<T&&> ( param) ;
}
```

Актуальный тип формируемого параметра:

template<typename T> void wrap(T&& param){…}

```
int main()
{
        A a1(1);
        const A a2(2);

        wrap(a1);               // wrap<A &>(A&);
        wrap(a2);               // wrap<A const &>(const A&);
        wrap (A(66));           // wrap<A>(A&&);
}
```

Но!

```
void f(const A& a) { std::cout<<a; }
void f(A& a) {++a;}
void f(A&& a) { std::cout<<std::move(a)); }
template<typename T> void wrap(T&& t) {…          f(t);     … }

int main(){
    A a1(1);
    const A a2(2);

    wrap(a1); //f(A&)  - OK
    wrap(a2); //f(A const &) - OK
    wrap(A(66));//но!  f(A&) - !!! Почему?
}
```

Решение: вариант № 1 – перегрузка

```
void wrap(A&& a)  {    f(std::move(a));   }
void wrap(A& a)  {    f(a);   }
void wrap(const A& a) {   f(a);   }

int main(){
    A a1(1);
    const A a2(2);
     wrap(a1);  //f(A &)
     wrap(a2);  //f(A const &)
     wrap(A(66)); //f(A&& )
}
```

Решение: вариант № 2 – forward

```
template<typename T> void wrap(T&& t){
        … f(std::forward<T>(t)); …
}
int main(){
   …
    wrap(a1); //f(A&) - OK
    wrap(a2); //f(const A&) - OK
    wrap(A(66));//f(A&&) - OK!!!
}
```

???

```
class WrapA {
   A m_A;
public:
   …
   void setNewA(const A& newA) { m_A = newA; }
};
int main(){
        WrapA w;
        w.setNewA(a1);
        w.setNewA(A(2)); //???
}
```

Перегрузка:

```
class WrapA {
   A m_A;
public:
   …
   void setNewA(const A& newA) { m_A = newA; }
   void setNewA(A&& newA) { m_A = std::move(newA); }
};
int main(){
        WrapA w;
        w.setNewA(a1);
        w.setNewA(A(2)); //???
}
```

Но!

```
class WrapA {
   A m_A;
public:
   template<typename T> void setNewA(T&& newA)
                                { m_A = newA; }
};
int main(){
          A a1(1);
          WrapA w(1);
          w.setNewA(a1); //какой A::operator= ???
          w.setNewA(A(2)); //какой A::operator= ???
}
```

«Перенаправление»

```
class WrapA {
  A m_A;
public:
  template<typename T> void setNewA(T&& newA)
                          { m_A = std::forward<T>(newA); }
};
int main(){
        A a1(1);
        WrapA w(1);
        w.setNewA(a1); // A::operator= (const A&)
        w.setNewA(A(2)); // A::operator= (A&&)
}
```

Полезный пример использования

forward

Перегрузка конструкторов (без использования forward)

```
class MyString{
        char* m_pStr;
public:
    …
        MyString(const MyString& );
        MyString(MyString&& );
};

class A{
        MyString m_str;
public:
        A(MyString && s) : m_str( std::move(s) ){}
        A(const MyString & s) : m_str(s) {}
        A(const char* p) : m_str(p){}
        …
};
```

Перегрузка конструкторов (без использования forward)

```
{
    MyString s(“ABC”);
    A a1(s); //???
    A a2(MyString(“QWERTY”)); //???
    A a3(“Hello”); //???
}
```

Пытаемся объединить три перегруженных конструктора посредством шаблона **(пока без forward)**

```
class MyString{
        char* m_pStr;
public:
        …
        MyString(const MyString& );
        MyString(MyString&& );
};

class A{
        MyString m_str;
public:
        template<typename T> A(T&& t) : m_str(t){}
};
```

Пытаемся объединить два перегруженных конструктора посредством шаблона (пока без forward)

```
{
  MyString s("ABC");
  A a1(s); //MyString(const MyString& )

  A a2(MyString("QWERTY")); //MyString(const MyString& );
                                                    !!!
  A a3("Hello"); //MyString(const char* );
}
```

Полезный пример использования **forward**

```
class MyString{
        char* m_pStr;
public:
    …
        MyString(const MyString& );
        MyString(MyString&& );
        MyString(const char*);
};

class A{
        MyString m_str;
public:
        template<typename T> A(T&& t) : m_str(std::forward<T>(t) ){}
};
```

Полезный пример использования **forward**

```
{
  MyString s(“ABC”);
  A a1(s); //MyString(const MyString& )

  A a2(MyString(“QWERTY”)); //MyString(MyString&& ) !!!

  A a3(“www”); ???
}
```

Еще один полезный пример использования forward

```
class MyString{…
        char* m_pStr;
public:
    …
        MyString& operator=(const MyString& );
        MyString& operator=(MyString&& );
        MyString& operator=(const char*);
};

class A{
        MyString m_str;
public:
        …
        template<typename T> void setNewString(T&& newStr)
        { m_str = std::forward<T>(newStr) ;}
};
```

DELEGATING CONSTRUCTORS
INHERITING CONSTRUCTORS

До стандарта C++11. Дублирование кода

```
class A{
…
public:
        A(){<код_1>}

        A(int x){
        <код_1> //повтор!
        <код_2>
        }
};
```

???

```
class A{
…
public:
        A(){<код_1>}

        A(int x){
        A(); //???
        <код_2>
        }
};
```

До стандарта C++11. Решение

```
class A{
…
public:
        void Init(){<код_1> }

        A(){ Init(); }

        A(int x){
         Init();
        <код_2>
        }
};
```

Стандарт C++11. Делегирующие конструкторы

```
class A{
…
public:
        A(){<код_1>}

        A(int x) : A(){
                <код_2>
        }
};
```

«Наследуемые» конструкторы

```
class A{
        int m_a;
public:
        A(int a): m_a(a){}
};

class B : public A{
public:
        B(int a) ; // параметр предназначен базовому
                   классу, а в производном никаких
                   инициализирующих действий НЕ требуется!
};
```

Аналогия: перегрузка невиртуальных методов класса

```
class A{
…
public:
void f(int);
};
class B: public A{
…
public:
void f(int, int);
};
int main()
{
        B b;
        b.f(1,2); //OK
        b.f(1); //???
}
```

Решение (одно из решений)

```
class A{
…
public:
void f(int);
};
class B: public A{
…
public:
void f(int, int);
using A::f;
};
int main()
{
        B b;
        b.f(1,2); //OK
        b.f(1); //ОК
}
```

До стандарта C++11

- такое использование using **не** распространялось **на конструкторы**

Без Inheriting constructors

```
class A{
        int m_a;
public:
        A(int a): m_a(a){}
};

class B : public A{
public:
        B(int a) : A(a){}
};
```

Inheriting constructors

```
class A{
        int m_a;
public:
        A(int a): m_a(a){}
};
class B : public A{
public:
        using A::A;
};
int main()
{
        B b(22);
}
```

Какие конструкторы можно наследовать:

```
class A {
public:
        A(int) {};
        A() {};
        A(const A& a) {}
        A(A&& a) {}
};
class B : public A {
using A::A;
};

B b1(1);
B b2;
B b3 = b1;
B b4 = std::move(b1);
```

Замечание:

- При добавлении в производный класс конструктора с такой же сигнатурой пользовательский конструктор будет приоритетнее

EXPLICIT ДЛЯ ЗАПРЕТА НЕЯВНОГО ПРЕОБРАЗОВАНИЯ

До C++11

использование explicit?

Пример использования explicit для запрета неявного преобразования

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

int main()
{
        A a = 1; //???
}
```

C++11

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

A a2(2); // OK
A a1 = 1; // OK (implicit)
A a3{ 4, 5 }; // OK:  A::A(int, int)
A a4 = { 4, 5 }; // OK: A::A(int, int)
int na1 = a1; // OK: A::operator int()
int na2 = static_cast<int>(a1); // OK: A::operator int()
```

C++11

```
class B {…
public:
        explicit B(int);
        explicit B(int, int);
        explicit operator int() const;
};

//  B b1 = 1; // ошибка
B b2(2); // OK
B b3{ 4, 5 }; // OK: B::B(int, int)
//  B b4 = {4,5}; //ошибка
//  int nb1 = b2; // ошибка
int nb2 = static_cast<int>(b2); // OK:
```

explicit и универсальные списки инициализации

```
class A{
        int m_x,m_y;
public: A(int x, int y);
};
class B{
        int m_x, m_y;
public: explicit B(int x, int y);
};

A a1{ 1, 2 };
B b1{ 3, 4 };
A a2={ 1, 2 };
//B b2={ 3, 4 };
```