Стандарт C++11-C++14-C++17 для прикладного программирования

Полубенцева Марина Игоревна

ВВЕДЕНИЕ

История:

- C++98 – 1998г
- C++03 – «технические поправки» к C++98
- TR1 – «Technical report on C++ Library Extensions» (namespace std::tr1)
- **C++11 (C++0x)** «Information Technology – Programming Languages – C++»
- C++14
- C++17

Стандарт C++11 это:

- результат предложений
  - компаний
  - и отдельных лиц,
- одобренный организацией по стандартизации

Стандарт C++11

- Это первое значительное изменение **стандарта** с 1998-го года!
- Радикальные изменения были добавлены:
  - как в **ядро** языка
  - так и в **стандартную библиотеку**
- Многие полезные нововведения были позаимствованы из популярной библиотеки **Boost**
- Поддерживается большинством современных компиляторов, но в **разном объеме**

Назначение изменений:

- повышение производительности,
- повышение безопасности,
- предоставление прикладному программисту удобных средств для уменьшения «нагрузки на мозг»
- добавление принципиально новой функциональности

Основные нововведения (C++11 /C++14):

- универсальная инициализация
- rvalue reference и move semantics
- продвижение типа (forwarding)
- шаблоны с переменным числом параметров (variadic templates)
- lambda-функции
- auto
- регулярные выражения
- поддержка многопоточности
- …

C++11 (C++14, C++17)

ISO/IEC 14882:2011

STRONGLY TYPED ENUMS

Проблемы – перечисления C++ **не** являются типобезопасными:

```
enum COLORS{RED, GREEN, BLUE};
COLORS color = RED;
int val = RED; //???
…
if(color==GREEN)…        //???
if(color==55)…      //???

//или
enum {OK, CANCEL, RETRY};
if(color ==OK)…    //???
```

Проблемы:

- способ представления в памяти (целочисленный тип) зависит от реализации и поэтому **не является переносимым**
- при создании переменных перечислимого типа резервируется **sizeof(int)** байтов => обычно «**с избытком**»

Проблемы области видимости

```
{
    enum VALUES{x, y, z};
    enum MEMBERS{x, y, z}; // 'redefinition'
}
```

Проблема!

```
enum Color     { RED, BLUE  };

enum Fruit    { BANANA,  APPLE };
Color a = RED;
Fruit b = BANANA;

if (a == b) //???
```

C++11 - enum class

```
enum class COLORS{RED=1, GREEN, BLUE=4};
COLORS color = COLORS::RED;
int val = COLORS::RED;// cannot convert from 'main::COLORS' to 'int'
if(color==COLORS::GREEN)... //класс формирует область видимости
if(color==55)...      //операнды должны быть одного типа

enum class CHOICE{OK, CANCEL, RETRY};
if(color == CHOICE::OK)... //операнды должны быть одного типа
if(static_cast<CHOICE>(color) == CHOICE::OK);
```

Тип перечисления

- задается неявно – эквивалент **signed int**
- можно задать явно:

```
enum class Values : unsigned char {x,y,z};
char c1 = Values::y; //???
Values c2 = Values::y;
```

- но!

```
char c3 = static_cast<char>(Values::y); //OK
//enum class ValuesD : double { x, y, z }; //ошибка
```

Предварительное неполное объявление перечислений

```
//.h
enum class Values : unsigned char;

class B {
        Values m_val;
};

enum class Values : unsigned char { x1, y1, z1 };
```

Спецификатор

AUTO

До C++11

auto ???

C++11 – использует ключевое слово **auto** для:

- автоматического определения компилятором типа **auto** переменной исходя из типа инициализирующего выражения
- обозначения типа **возвращаемого значения** при использовании trailing return type (не осуществляет выведение типа – это просто синтаксис)
- C++14 – для автоматического определения типа возвращаемого любой функцией значения
- в будущем (когда будет введено понятие консепта) при задании параметров шаблона (в качестве placeholder для типа)
- C++14 – для обозначения типа параметра лямбда-функции
- C++14 – при использовании в качестве выражения в decltype

Важно!

Используется :

- не только для простой и короткой формы сложных/длинных типов
- но и для выведения типов, зависящих от **конкретной** реализации

Важно!
Замечание:

вывод типа auto-переменной происходит так же, как вывод типа параметра шаблона (кроме параметра шаблона универсального типа **T&&**)

=>

- “ссылочность” игнорируется!
- “бантики” игнорируются!

Напоминание: правила выведения типа параметра шаблона

```
template<typename T> void f(T t){…}

int main(){
        int n = 1;
        f(n);           //T==??? t==???
        f(&n);          //???
        int& r = n;
        f(r);           //???
}
```

Продолжение напоминания правила выведения типа параметра шаблона

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

int main(){
        int n = 1;
        const int nc = n;
        const int& rc = n;
        f(n);           //T==??? t==???
        f(nc);          //???
        f(rc);          //???
}
```

Продолжение напоминания правила выведения типа параметра шаблона

```
template<typename T> void f(const T& t){…}

int main(){
        int n = 1;
        const int nc = n;
        const int& rc = n;
        f(n);           //T==??? t==???
        f(nc);          //???
        f(rc);          //???
}
```

Или типа возвращаемого шаблонной функцией значения:

```
template<typename T> T f(){
        T t;
        T& ret = t;
        return ret; //???
}

int main(){
        int res = f<int>();
}
```

Замечание:

«The old meaning of auto *was* ***removed*** *from C++11 to avoid confusion»*

Старое значение ключевого слова auto было удалено из C++11, чтобы избежать путаницы

auto int n = 1; //ошибка

Специфика

- **auto**-переменная не может хранить значения разных типов! => после инициализации сменить тип переменной невозможно! (C++ как был, так и остается статически типизированным языком)

Пример использования auto

Как делать НЕ стоит!

```
auto n = 1;
```

А так удобно:

```
vector<int> v;
…
vector<int>::iterator it1 = v.begin(); //тип задан явно программистом
auto it2 = v.begin(); //тип «выведен» компилятором неявно
```

Специфика auto:

- тип определяется только при **инициализации**:

```
auto x=0;      // ???
auto y = 4, z = 3.14; //???
auto a; //???
```

- изменить тип (при присваивании) **невозможно**:

```
x="abc"; //???
x=5.5; //???
```

- со спецификатором auto можно использовать **const**, **static**, *, **&**, и **&&**

```
std::vector<int> v;
const auto& rv = v;
```

Полезность auto:

1.

int* p;

//забыли присвоить адрес => появляется возможность использования не проинициализированного значения

2.

auto p; //???

auto p = new int[n];

«Подводные камни»

```
std::map<std::string, int> m;
//формирование значений
for (auto it = m.begin(); it != m.end(); ++it){
   const std::pair<std::string, int>& p = *it;
   std::cout << p.first << " " << p.second << std::endl;
}
//эффективность???
```

auto???

```
std::map<std::string, int> m;
//формирование значений
for (auto it = m.begin(); it != m.end(); ++it){
   auto p = *it; //???
   const auto& p1 = *it; //???
   std::cout << p.first << " " << p.second << std::endl;
}//тип p???
//эффективность?
```

???

```
int count = 1;
int& countRef = count;
auto myAuto = countRef; //???

countRef = 2; // count ==???

myAuto = 3; // count ==???
```

Примеры

1.

```
auto x = "qwerty"; //какого типа x???
```

2.

```
short a=1, b=2;
auto y = a/b; //какого типа y???
```

Примеры (продолжение)

3.

```
int f();
int main()
{
        auto res = f();  //какого типа res???
}
```

4.

```
auto p1 = new auto('a'); // какого типа p1???
auto* p2 = new int[10]; // какого типа p2???
auto p3 = new int[10];
```

Примеры (продолжение)

5.

```
auto x=1, *p=&x;  //какого типа x и p???
```

6.

```
//и даже так:
auto a = {1,2,3,4,5}; // std::initializer_list<int>
```

Примеры (продолжение)

7.

```
auto x=1;                    //тип x ???
const auto y = 1;            //тип y ???
const auto& z = 1;           //тип z???
```

Реализовать функцию суммирования элементов любого контейнера

???

Удобно:

```
template<typename C> ... sum(const C& c) {…}

int main()
{
        vector<int> v(10,1);
        …
        auto result = sum(v); //???
}
```

C++14 - Автоматическое определение типа возвращаемого значения :

Невозможно:

```
.h

auto f();
```

У компилятора есть возможность вывести тип

```
.h
inline auto f()
{
        return 1;
}
```

auto – вывод типа возвращаемого функцией значения

- C++ 11 разрешает вывод возвращаемых типов лямбда-выражений из одной инструкции return,
- а C++ 14 расширяет эту возможность на все лямбда-выражения и все функции (включая состоящие из множества инструкций return)
  при условии, что программист обеспечивает однозначность компилятору

Замечания:

- Ключевое слово auto – это лишь **средство автоматического выведения типа** компилятором, а **не тип**!

=> нельзя использовать:

  - в операторах приведения типа,
  - в операторе sizeof
  - в операторе typeid
  - для выведения типа элемента массива
    auto ar[] = { 1,2,3 }; //ошибка!

- исключительно полезно при использовании шаблонов!

C++17

```
std::pair<auto, auto> p2 = std::make_pair(0, 'a');
```

Замечание
ВАЖНОЕ!

Перечисленные правила **не** распространяются на выведение типа **универсального параметра** шаблонной функции!!!

Диапазонный for

RANGE BASED FOR LOOP (FOR-ЦИКЛ ПО КОЛЛЕКЦИЯМ)

М.Полубенцева

Range based for loop
(эквивалент for_each())

позволяет:

- если для последовательности существуют **begin()** и **end()**
- выполнить тело цикла для всех элементов последовательности от **begin()** до **end()**
- для каждого элемента последовательности

Синтаксис:

```
for(тип_элемента формальное_имя
                    :
                    имя_последовательности)
{
    тело_цикла
}
```

Range based for loop

неявно:

1. формирует итераторы на начало и на конец для указанной последовательности
2. переменная цикла – это итератор
3. выполняет тело цикла,
   - передавая на каждой итерации по формальному имени разыменованный итератор (то есть очередное значение элемента последовательности)
   - перемещает итератор на следующий элемент

Эквивалент

Важно! диапазон вычисляется до выполнения цикла

```
for(auto it = begin(cont), itEnd =  end(cont);
                                        it != itEnd; ++it)
{
        T  формальное_имя = *it;
        //или T&, const T&
        тело_цикла
}
```

=> Следствие!!! В теле цикла **НЕЛЬЗЯ** делать итератор недействительным (например, вставлять/удалять элементы)

Замечания:

- область видимости формального имени распространяется только на тело цикла!
- в теле цикла можно использовать:
  - break
  - continue
  - return
  - goto (не рекомендуется)
  - throw

Range based for loop

(не модифицирующее использование)

```
vector<int> v;
…
for (auto it=v. begin(); it!=v.end(); ++it)
{std::cout<<*it<< ' ';}

for (int x: v) {  std::cout << x<< ' ';}

for (const auto& x: v) {  std::cout << x<<' '; }
```

Пример

```
std::vector<int> v;
//формирование значений
for (const auto& i : v) {
        if (i == 0) continue;
        std::cout << i << ' ';
}
```

Пример range-based for loop

(модифицирующее использование)

```
int my_array[5] = {1, 2, 3, 4, 5};
// все элементы умножить на 2:

for (int &x : my_array) {
    x *= 2;
}

for (auto &x : my_array) {
    x *= 2;
}
```

Без использования auto:

```
std::map<std::string, int> m;
//сформировали значения
for (const std::pair<std::string, int>& p : m){
        std::cout << p.first << " " <<
                        p.second << std::endl;
}
//эффективность???
```

Полезность auto:

```
std::map<std::string, int> m;
//сформировали значения
for (const auto& p : m){
        std::cout << p.first << " " <<
                        p.second << std::endl;
}
//тип p ==
const std::pair<const std::string, int>&
```

Посредством range-based for loop???

std::vector<int> v(5,1);

//модифицировать вектор таким образом, чтобы значения стали:

1 2 1 2 1 2 1 2 1

Пример – печать map

```
int main() {
   unsigned int days[] = { 31,28,31,…};
   const char* names[] = { "January", "February", … };
   std::map<const char*, int> months;
   for (int i = 0; i < 12; i++)
   {
         months[names[i]] = days[i];
   }
//Порядок???
//Вывести соответствие ???
}
```

Замечание

range-based for может быть использован со списком инициализации:

```
for(int n: {0, 1, 2, 3, 4, 5})
{ std::cout << n << ' '; }
```

Чтобы range-based for работал с пользовательской структурой данных:

- должны быть реализованы begin() и end()
- должен быть определен итератор, для которого перегружены операторы *, !=, ++

list initialization

УНИВЕРСАЛЬНАЯ ИНИЦИАЛИЗАЦИЯ

До стандарта C++11 – только для POD (plane old data)

инициализация массивов, структур языка С и даже объектов классов (при соблюдении ограничений)

int ar[] = **{1,2,3,4,5}**; //OK - все инициализаторы одного и того же типа)

```
struct A{
        int n;
        char ar[5];
};
```

A a = **{4, “abc”}**; //OK - инициализаторы разного типа, поля структуры должны быть public: , конструктор запрещен

До C++11

```
class A{
        int n;
        char str[20];
public:
        A(int, const char*);
};

A a1 = {1,”qwerty”}; // ???
```

Стандарт C++11 предоставляет

универсальную форму инициализации любых типов данных (это **обобщение** инициализации массивов, структур языка C, классов C++ … не только для POD, но и для любых пользовательских типов)

Очень простые примеры для базовых типов

```
int i=1;
int j(i); //???
int k = int(); //???
int l(); //???
int ar1[] = {1, i, j};

//C++11:
int x{i}; //эквивалентно int x={i};
int y{};
int ar2[] {1, i, j};
```

Специфика: запрещены неявные «**сужающие**» приведения типа:

```
int x=1.2; //???
int y(1.2); //???
int z{1.2}; //ошибка
//но!
char c = 100;
int w{ c }; //корректно
```

Замечание:

При инициализации **константой** в C++11 по возможности используется не только анализ типов, но и анализ реальных значений инициализаторов:

- **char c1{5}; //** ОК: 5 – это int, но он умещается в char
- **char c2{555}; //** ошибка - сужение

Две формы использования списка инициализации

direct-list-initialization

```
int n{1};
char ar[]{ 'a','b','c' };
std::string s1{ "abc" }; //вызывается конструктор (const char*)
```

copy-list-initialization

```
int n = { 1 };
char ar[]={ 'a','b','c' };
std::string s = { "abc" };
//вызывается конструктор (const char*)
```

Динамические массивы и списки инициализации

- **До C++11:**

```
int* p1 = new int[3]; //инициализация ???
int* p2 = new int(3); // ???
```

- **C++11**

```
int* p3 = new int[3] { 1, 2 }; //1,2,0

int n=5;
int* p4 = new int[n] { 1, 2 }; //до VS15RC 1,2,<неиниц>, VS15RC 1,2,0,0,0

int a = 1, b = 2;
int* p5 = new int[3] { a, b};
```

Замечание:

```
int * p1 = new int[3]; //???
int * p2 = new int[3](); //вызов default-конструктора для базовых типов
int * p2 = new int[3]{};
int n = 3;
int * p3 = new int[n]();
```

Если инициализаторов больше:

```
int n=3;
int* p6 = new int[3]{ 1,2,3,4,5 }; //ошибка компилятора – слишком много инициализаторов
int* p7 = new int[n]{ 1,2,3,4,5 }; //vs15 RC – ошибка времени выполнения, vs15 ultimate – инициализация за пределами выделенной памяти!!!
```

Для пользовательских типов

string * p8 = new string[**5**]{"aaa", "bbb"}; //"aaa", "bbb”, “”, “”, “”

//но!!!

int n=5;

string * p9 = new string[**n**]{ string("aaa"),

string("bbb") }; //"aaa", "bbb”, не иниц, не иниц, не иниц

=> delete[] p9; //???

Если есть конструктор, компилятор использует {} для вызова конструктора:

```
class A{
public:    int m_x; double m_y;
};
class B{
           int m_x;  //спецификатор???
           double m_y; //спецификатор???
public:    B(int x, double y) ; //для инициализации предоставляется конструктор
};

A a1 = {1, 2.2}; //инициализация в «старом стиле»
A a2 {3, 4.4}; //аналогично
A a3 = { 5 }; //???

B b1 = {1, 2.2}; //вызов конструктора
B  b2 {3, 4.4}; // аналогично
//B b3{ 5 }; //ошибка
```

Простой пример:

```
struct A{
        int m_a;
        std::string m_str;
};

A a1{ 1, "abc" };
A a2 = { 2, "qwerty" };
```

Пример поинтереснее:

```
class A{
int m_a;
std::string m_str;
public:
A(int a, const char* s)
:m_a(a), m_str(s){}
};
```

```
class B{
A m_A;

public:
B(int a, const char* s) : m_A{ a, s }{}
};
```

```
B b1{ 1, "abc" };
B b2 = { 2, "qwerty" };
```

И даже так:

```
class A{
        int m_a;
        std::string m_str;
public: A(int a, const char* s) :m_a(a), m_str(s){}
};

A fA() { return{ 1, "abc" }; } //тип не указывается явно!

int main(){
        A a = fA();
}
```

Для инициализации встроенных массивов

```
class A {
        int a[4] ;
public:
   A() : a{1,2,3,4} {}
    A(int a0, int a1, int a2, int a3) : a{ a0, a1, a2, a3 } {}
};
```

Аналогично с пользовательскими типами

```
class A {
        std::string a[2];
public:
        A() : a{ "a","b" } {}
        A(const char* a0, const char* a1) : a{ a0, a1 } {}
};
A a{ "c", "d" };
A a1;
A a2(); //???
A a3{}; //
```

Шаблонный класс **std::initializer_list**
#include <initializer_list>

- предоставляет «обертку» для массива const T[] – элементов (proxy)
- может быть реализован посредством пары указателей или указатель + длина массива
- тип T – должен быть copy constructable!
- можно явно создавать объекты **initializer_list** для дальнейшего использования
- обычно создаются компилятором автоматически
- ! время жизни массива может быть меньше времени жизни **initializer_list!**
- время жизни объектов **initializer_list** определяется по общим правилам

Специфика **std::initializer_list**

- Компилятор создает неименованный локальный (стековый) массив
- Компилятор создает объект std::initializer_list (именованный или неименованный), в котором формирует указатель на начало массива + признак конца
  Следствие: объекты должны быть copy-constructible!
- Итераторы std::initializer_list предназначены только для чтения!

Замечание:

- В случае :

f{1,2,3,4}; //лучше принимать по значению, так как компилятор соптимизирует и создаст std::initializer_list в качестве параметра

- А если

std::initializer_list<int> l = {1,2,3,4};

f(l); //то лучше принимать по ссылке

std::initializer_list предоставляет:

- итераторы на начало и конец последовательности можно получить с помощью:
  - методов begin() и end()
  - глобальных функций begin() и end()
- size()

Явное создание объектов initializer_list (для дальнейшего использования)

1.

```
std::initializer_list<int> iList1 = { 1, 2, 3, 4, 5 };
auto iList2 = { 1, 2, 3, 4, 5 }; // std::initializer_list<int>
```

2.

```
auto iList = { 1, 2, 3, 4, 5 };
for (int x : iList) {…}
```

3.

```
std::initializer_list<int> iList1 = { 1, 2, 3, 4, 5 };
std::initializer_list<int> iList2 = { 7, 8, 9};
iList2 = iList1; //что копируется???
```

???

```
std::initializer_list<int> f(){
std::initializer_list<int> iList = { 1, 2, 3, 4, 5 };
return iList;
}

int main(){
std::initializer_list<int> iList2 = f();
…
}
```

Когда компилятор может создавать объекты initializer_list **неявно** (исходя из контекста)

1.

for (int x : **{1, 2, 3}**) {…}

Когда компилятор может создавать объекты initializer_list **неявно** - продолжение

2. если в классе есть конструктор, принимающий initializer_list

```
class Array{
        int m_ar[4];
public:
        Array(std::initializer_list<int> l){
          size_t n = min(sizeof(m_ar)/sizeof(int), l.size());
          auto it = l.begin();
          for (size_t i = 0; i < n; i++)   { m_ar[i] = *it;  ++it; }
         }
};

Array ar1 { 1, 2, 3, 4, 5 };
Array ar2 { 1, 2, 5 };
```

Когда компилятор может создавать объекты initializer_list **неявно -** продолжение

- }

3.

```
void f(std::initializer_list<int> iList){…}


int main()
{
        f({ 1, 2, 3 });
}
```

Когда компилятор может создавать объекты initializer_list **неявно**

```
auto x1 = 1;  //тип x1 ???
auto x2 ( 1 ) ; //тип x2 ???
auto x3 = { 1 } ; //тип x3 ???
auto x4 { 1, 2 } ; //тип x4 ???
//но!
auto x5 { 1, 2.2 } ; //ошибка!
```

Когда компилятор **не** может создавать объекты initializer_list **неявно**

1. Ошибка:

```
template<typename T>  void f (T param) ;

f ( { 1,2,3} ) ;
```

2. OK!

```
template<typename T>
void f ( std: : initializer_list<T> initList ) ;

f ( { 1,2,3} ) ;
```

???

```
class Array{
        int m_ar[10];
public:
        Array(std::initializer_list<int> l);
};

int main()
{
        Array ar { 1, 2, 3, 4, 5 };
        ar = {5,6,7,8}; //???
}
```

Замечание насчет эффективности:

1. эффективно

```
vector<string> vec1 {"aaa", "bb", "cccc"};
```

2. дополнительные копирования!

```
string s1(“aaa”), s2(“bbb”), s3(“ccc”);
vector<string> vec2 {s1, s2, s3};
```

C++11 std::initializer_list и контейнеры STL

- для всех контейнеров добавлен **конструктор**, принимающий initializer_list:

```
std::vector<int> v = { 1, 5, 6, 0, 9 };
std::vector<int> v { 1, 5, 6, 0, 9 };
```

- а также **методы**, которые принимают initializer_list в качестве параметра

```
std::vector<int> v;
v.insert(v.end(), {0, 1, 2, 3, 4});
```

Важно!

Если в классе есть конструктор, принимающий в качестве аргумента список инициализации

initializer_list<SomeType>,

он будет иметь **более высокий приоритет** по сравнению с другими возможностями создания объектов

Специфика **initializer_list**

(нет конструктора, который принимает initializer_list)

```
class A{
…
public:
        A(int, int);
};
int main(){
        A a1(1,2);  //конструктор
        A a2{3,4};  //конструктор
        A a3{ 5 };   //???
        A a4{ 6,7,8 }; //???
}
```

Специфика **initializer_list**

```
class A{
…
public:
        A(int, int); //1)
        A( std::initializer_list<int> ); //2)
};
int main(){
        A a1(1,2);  //???
        A a2{3,4};  //???
        A a3{ 5 };   //???
        A a4{ 6,7,8 }; //???
}
```

Пример. Разница???

```
int ar[2][5] =
{
        {1,2,3},
        {4,5,6,7,8}
};

std::vector<std::vector<int>> vv =
{
        {1,2,3},
        {4,5,6,7,8}
};
```

Разница ???

```
std::vector<int> v1(8); //???

std::vector<int> v2{8}; //???
```

Вложенные списки инициализации

```
std::map<std::string, int> m{
        { "bbb",2 }, { "aa", 1 }
};
//Порядок?
```

Специфика:

1. ошибка

```
auto createinitList() {
return { 1 , 2 , 3 };
}
```

2. ОК

```
auto createinitList() {
return std::initializer_list<int>({ 1 , 2 , 3 });
}
```

TRAILING RETURN TYPE

trailing return type

```
class A {
        int m_a;
public:
        A(int a=0){m_a = a;}
        auto Get()->int //trailing return type
        {return m_a;}
};
```

trailing return type используется:

- в лямбда-функциях (нет других вариантов задать тип возвращаемого значения)
- для удобства (чтобы уменьшить количество текста)
- в шаблонных функциях для вывода типа возвращаемого значения

Без trailing return type:

```
                                                    //Person.h
class Person{
public:
   enum PersonType { ADULT, CHILD, SENIOR };
   …
   PersonType getPersonType ();
private:
   PersonType person_type;
};
```

```
                                                    //Person.cpp
#include “Person.h”
Person::PersonType Person::getPersonType () { return person_type; }
//компилятор еще не знает, что такое PersonType
```

Использование trailing return type:

```
                                                    //Person.h
class Person{
public:
   enum PersonType { ADULT, CHILD, SENIOR };
   …
   PersonType getPersonType ();
private:
   PersonType person_type;
};
                                                    //Person.cpp
#include “Person.h”
auto Person::getPersonType () -> PersonType{ return person_type; }
//компилятор уже знает, что такое PersonType
```

DECLTYPE

спецификатор decltype

- позволяет на этапе компиляции выводить тип по типу выражения
- в качестве выражения можно использовать:
  - другую переменную
  - любое **корректное с точки зрения компилятора** выражение,
  - возвращаемое функцией/функтором значение)
- => исключительно полезно при использовании шаблонов

Примеры

```
int i;
decltype(i) n; // ???
decltype(i + 1) m; // ???
decltype(i = 4) x=i; //???

int f();
decltype(f()) ccc;  // ???
```

Пример decltype

1.

```
int x=1;
decltype(x) y; //???
```

2.

```
int x=1;
double y = 5.5;
decltype(x+y) z; //???
```

3. но!

```
auto p0 = "abc";
//decltype("qwerty") p1 = "abc";// ош: справа - const char*, слева const char (&)[7]
decltype("qwerty") p2 = "abcabc";// OK
```

decltype и шаблоны C++11:

```
template<typename A, typename B>
        auto
        sum(const A& a, const B& b)
                                ->decltype(a+b)
{
        return a+b;
}
```

decltype и шаблоны **C++14**:

Замечание: в C++14 trailing return type можно опустить:

```
template<typename A, typename B>
        auto
        sum(const A& a, const B& b)
{
        return a+b;
}
```

Разница auto и decltype

```
const std::vector<int> v(5);
auto a = v[0]; // тип a?
decltype(v[0]) b = 1; // тип b?
```

Определение псевдонимов (typedef) посредством decltype

```
const vector<int> vi;
typedef decltype (vi.begin()) CIT;
CIT another_const_iterator;
```

Специфика decltype

decltype **не** вычисляет выражение для вывода типа =>

```
auto a = 10;
decltype(a++) b; //тип b???
//a==???
decltype(++a) c=a; //тип c???
```

Разница auto и decltype

A a;

const A& cr = a ;

auto a1 = cr; //вывод типа auto : тип a1 – **A**

decltype(auto) **cr2** = cr; // вывод типа decltype :

// тип cr2 - **const A &**

Проблемы?

```
template<typename Cont, typename index>
                                    auto f1(Cont& c, const index& i){
        //...
        return c[i];
}

int main(){
        std::vector<int> v = {1,2,3};
        int x = f1(v, 1); //OK
        f1(v, 1)=33; //ошибка - int f1(vector<int>&, const int&)
}
```

C++14 - decltype(auto)

Решение – функция должна возвращать в точности тот же тип, что и выражение c[i]:

```
template<typename Cont, typename index> decltype(auto)
                                        f2(Cont& c, const index& i){
        //...
        return c[i];
}

int main(){
        std::vector<int> v = {1,2,3};
        int x = f2(v, 1);
        f2(v, 1)=33; //OK – int& f1(vector<int>&, const int&)
}
```

decltype(auto)

```
std::map<std::string, int> m;
//формирование значений
for (auto it = m.begin(); it != m.end(); ++it){
   auto p = *it; //тип p ???
   auto& p1 = *it; // тип p1???
   decltype(auto) p2 = *it; // тип p2???
   std::cout << p.first << " " << p.second << std::endl;
}
```

decltype(auto)

```
int innerF1() { return 1; }
int& innerF2() { static int res = 33; return res; }
```

- **C++11:**

```
int WrapF1C11() { return innerF1(); }
int& WrapF2C11() { return innerF2(); }
```

- **C++14**

```
decltype(auto) WrapF1C14() { return innerF1(); }
decltype(auto) WrapF2C14() { return innerF2(); }
```

Ограничения decltype

Несмотря на то, что decltype не вычисляет выражение, выражение, используемое в decltype должно быть «**действительным**»!

```
class A{ …
private:
A();
};

cout << typeid(decltype(A())).name(); // ошибка: A() - private
```

Аналогично:

```
class A {
        int f() const { return 1; }
 };

int main(){
        //decltype(A().f()) n1 = 1; //inaccessible
}
```

Осторожно!

для lvalue- выражений, более сложных, чем просто имена, decltype гарантирует, что возвращаемый тип всегда будет lvalue-ссылкой:

```
decltype (auto) f1(){
int x = 0;
return x; // просто имя переменной =>decltype (x) == int
}
decltype(auto) f2() {
int x = 0;
return (x); // а это уже выражение => decltype ((x)) == int& =>???
}
```

#include <utility>

DECLVAL

шаблон std::declval

преобразует любой тип T к ссылке =>
позволяет использовать методы класса в decltype без использования конструкторов

Пример declval

```
struct A { int f() const { return 1; } };

struct B{//нет default конструктора
    B(const B&) { }
    int f() const { return 1; }
};

int main(){
    decltype(A().f()) n1 = 1;                  // тип n1 ???
//  decltype(B().f()) n2 = n1;       // error: no default constructor
    decltype(std::declval<B>().f()) n2 = n1; // тип n2???
}
```

Дополнения стандартной библиотеки

ПОЛУЧЕНИЕ ИТЕРАТОРОВ НА НАЧАЛО И НА КОНЕЦ ПОСЛЕДОВАТЕЛЬНОСТИ

М.Полубенцева

Как получить итераторы на начало и на конец?

1.

```
std::vector<int> v (10,1);
???
```

2.

```
template<typename C> auto sum(const C& c)
{
    //???
}
```

C++11 – для контейнеров добавлены методы получения константных итераторов

для повышения надежности в C++11 добавлены методы:

- const_iterator **cbegin**() const noexcept;
- const_iterator **cend**() const noexcept;
- const_reverse_iterator **crbegin**() const noexcept;
- const_reverse_iterator **crend**() const noexcept;

Отличия cbegin() и cend() от begin() и end()

```
std::vector<int> v = { 1,2,3 };
auto it1 = v.begin(); //std::_Vector_iterator
auto it2 = v.cbegin(); //std::_Vector_const_iterator

*it1 = 5; //???
*it2 = 10; //???
```

Но!

```
template<typename C> void Print(const C& c)
{
        auto itConstBegin = c.cbegin();

//Но! Пока продолжает работать
        auto itBegin = c.begin();
        …
}
```

C++11 - глобальные функции std::begin() и std::end() #include <xutility>

Перегруженные функции:

- для контейнеров STL (вызывают соответствующие методы)
- и обычных массивов (посредством sizeof)

для контейнеров STL :

```
std::vector<int> v (10,1);
auto b = std::begin(v);
auto e = std::end(v);
```

C++14

для получения константных итераторов:

std::cbegin()

и std::cend()

C++11 - глобальные функции begin() и end() для массива

```
#include <xutility>

int ar[] = {1,2,3};
int* b = std::begin(ar);
int * e = std::end(ar);

//Замечание: для получения константных итераторов cbegin() и cend() – C++14
```

C++11 - глобальные функции begin() и end() для **массива**.

Важно!

У компилятора должна быть возможность определить размер массива (sizeof) =>

В качестве параметра ссылка на массив

???

```
void f1(int ar[]){
        int* b = std::begin(ar); //???
        int * e = std::end(ar); //???
}
void f2(int (&ar)[3]){
        int* b = std::begin(ar);
        int * e = std::end(ar);
}
int main(){
        int ar[] = {1,2,3};
        f1(ar);
        f2(ar);
}
```

C++17 – шаблон глобальной std::size()

Перегружена:

- для контейнеров STL (вызывает метод size())
- для обычных массивов (принимает в качестве параметра ссылку на массив => sizeof)

???

модифицировать функцию суммирования элементов любого контейнера на массивы:

```
std::vector<double> vd = {1.1, 3, 5.5};
auto res1 = sum(vd);

int ar[] = { 1, 2, 3 };
auto res2 = sum(ar);
```

Замечание:

- глобальные begin() и end() можно применить к **std::initializer_list**

Пример

```
std::vector<int> v = {1,2,3,4,5};
auto itb = v.cbegin(), ite = v.cend(); // тип итераторов??
while (itb != ite)
{
        std::cout << *itb << ' ' ;
        ++itb; //???
        *itb = 33; // ???
}
```

Распечатать в обратном порядке (C++14 – C++17)

```
std::vector<int> v = {1,2,3,4,5};
```

???

C++17 std::distance()

```
std::string str("abcde");
int pos = std::distance(str.begin(),
        std::find(str.begin(), str.end(), 'd'));

char ar[] = "abcde";
int posar = std::distance(begin(ar),
std::find(std::begin(ar), std::end(ar), 'd'));
```

std::next(), std::prev()
#include <iterator>

```
std::vector<int> v{ 1,2,3,4,5};
size_t n = v.size()/2;
auto it = v.begin();
for(size_t i=0; i<n; i++)
{
        ++(*it);
        it = std::next(it,2);
 }
```

ЛЯМБДА-ВЫРАЖЕНИЯ

Лямбда-выражениями (функциями)

называются безымянные локальные функции, которые можно создавать прямо внутри какого-либо выражения. Используется для:

- в **C++** это краткая формы записи **анонимных функциональных объектов (функторов)**
- в Qt (в версиях, которые поддерживают C++11) это удобная форма задания короткого слота в функции connect()
- …

Синтаксис

[<список «захвата»>]

(<список параметров>)

-> <тип_возвращаемого_значения>

{<тело>}

Специфика лямбда-выражений

λ-выражение:

- всегда начинается с **[]** (скобки могут быть непустыми)
- затем идет необязательный список параметров
- параметры можно передавать разными способами (по ссылке, по значению)
- в некоторых (однозначных) случаях компилятор может сформировать тип возвращаемого λ-функцией значения неявно (в частности, если λ-функция ничего не возвращает) или программист всегда может указать возвращаемый тип явно
- затем непосредственно «тело функции»

Таким образом λ-функцию вряд ли стоит использовать:

int n = **[] (int x, int y) { return x + y; }**(5, 4);

Зато стоит подумать, что делает компилятор!

???

```
int ar[] = {5,-1,4,-7,3};
//распечатать куб каждого элемента
```

Использование λ -выражений в качестве предикатов

```
class PrintCube{
public: void operator ()(int x) const { cout<< (x*x*x)<<' '; }
};

int main()
{
        int ar[] = {5,-1,4,-7,3};
        for_each(ar, ar + sizeof(ar)/sizeof(int), PrintCube() );
        cout<<endl;
//Использование лямбда-выражений
        for_each(ar, ar + sizeof(ar)/sizeof(int),
                [](int x){cout<< (x*x*x)<<' ';} );
        cout<<endl;
}
```

Встречая лямбда-выражение, компилятор:

- генерирует анонимный класс (или структуру?),
- в котором перегружен operator()
- важно! метод operator() – **константный**!

В нашем примере метод:

  – ничего не возвращает (void),
  – принимает по значению очередной элемент последовательности

Как создать «переменную» типа λ –функции:

Тип лямбда-функций зависит от реализации => имя этого типа доступно только компилятору.

Тип переменной:

- можно попросить вывести посредством auto
- или «завернуть» в универсальную «обертку» - шаблон std::function

```
auto square = [](int x) { return x * x; };
int res = square(2);
```

Специфика реализации лямбда-функции без параметров

[] { <тело>};

эквивалентно

[] (){<тело>};

Тип возвращаемого λ-выражением значения

- Может формироваться компилятором неявно
- Программист может указать явно

Тип возвращаемого значения формируется компилятором неявно

```
//скопировать только отрицательные значения
int ar[] = {5,-1,4,-7,3};
vector<int> v;
copy_if(ar, ar + sizeof(ar)/sizeof(int), ???,
        [](int x){return x<0;}
        );
```

Тип возвращаемого значения указывается явно

```
int ar[] = {5,-1,4,-7,3};
vector<int> v;
copy_if(ar, ar + sizeof(ar)/sizeof(int),
        back_inserter (v),
        [](int x)-> bool{return x<0;}
        );
```

Неявный вывод типа возвращаемого значения

```
[](int x)      {if(0==x) return 1; else return 2.2;}
???
```

```
[](int x) -> double
        {if(0==x) return 1; else return 2.2;}
```

Замечания:

- C++11 для неявного формирования типа возвращаемого значения в лямбда-функции должна быть только **одна** инструкция return
- C++14 – несколько, но тип выражения должен быть одинаковым:

```
copy_if(ar, ar + sizeof(ar)/sizeof(int),
        back_inserter(v),
        [](int x)
        {if(x==0) return true; else return x<0;}
);
```

Формирование значений переменных анонимного функционального объекта

- Требуется скопировать только те значения, которые попадают в указанный диапазон

Функциональный объект

```
class Range{
        int lower, upper;
public:
        Range(int l, int u):lower(l), upper(u){}
        bool operator ()(int x) const
                { return (x > lower) && (x < upper); }
};
```

Лямбда-функция

```
int ar[] = {5,-1,4,-7,3};
int lower=0, upper=10;
vector<int> v;
copy_if(ar, ar + sizeof(ar)/sizeof(int),
        back_inserter (v),
         [lower, upper](int _x)-> bool
        {return _x>lower && _x<upper;}
);
```

Тип передаваемого параметра

```
//Увеличить все элементы на единицу:
int ar[] = {5,-1,4,-7,3};
for_each(ar, ar + sizeof(ar)/sizeof(int),
        [](int x){x++;}
       ); //???
for_each(ar, ar + sizeof(ar)/sizeof(int),
        [](int& x){x++;}
       ) ; //???
```

Модификация переменных анонимного функционального объекта внутри λ-функции

- Метод operator(), генерируемый компилятором, является **константным**!
- Как позволить модифицировать переменные класса в константном методе???

Пример - При каждом копировании увеличиваем верхнюю границу диапазона

```
int ar[] = {5,-1,4,-7,3,11};
int lower=0, upper=10;
vector<int> v;
copy_if(ar, ar + sizeof(ar)/sizeof(int), back_inserter (v),
        [lower, upper](int _x) -> bool
        {
        if(_x>lower && _x<upper){upper++; //ошибка!
                    return true;}
        else     return false;
        }
        );
```

mutable

```
int ar[] = {5,-1,4,-7,3,11};
int lower=0, upper=10;
vector<int> v;
copy_if(ar, ar + sizeof(ar)/sizeof(int), back_inserter (v),
        [lower, upper](int x)mutable -> bool
        {
        if(x>lower && x<upper){upper++; return true;}
        else    return false;
        }
); //??? значение upper??? (что будет изменяться?)
```

Формирование в анонимном объекте адресов внешних переменных (захват)

Посчитать количество скопированных элементов

Формирование в анонимном объекте адресов внешних переменных

```
int ar[] = {5,-1,4,-7,3,11};
int lower=0, upper=10, count = 0;
vector<int> v;
copy_if(ar, ar + sizeof(ar)/sizeof(int), back_inserter (v),
        [lower, upper,count](int _x)mutable /*-> bool*/
        {
        if(_x>lower && _x<upper){count++; return true;}
        else    return false;
        }
); //??? значение count???
```

Формирование в анонимном объекте адресов внешних переменных

```
int ar[] = {5,-1,4,-7,3};
int lower=0, upper=10, count = 0;
vector<int> v;
copy_if(ar, ar + sizeof(ar)/sizeof(int), back_inserter (v),
        [lower, upper, &count](int x) /*mutable*/       {
        if(x>lower && x<upper){count++;
                return true;}
                else return false;
        }
        ); //???
```

Захват по значению и по ссылке

```
[x, y]          // захват x и y по значению
[&x, &y]        // захват x и y по ссылке
[x, &y]         // захват x по значению, а y — по ссылке
[=]             // все из внешнего контекста по значению
[&]             // все из внешнего контекста все по ссылке
[=, &x]         // захват x по ссылке, все остальные по значению
[this]          //захват всех переменных объекта
```

Что можно/нужно захватывать:

- все локальные переменные текущей функции, определенные «выше»
- параметры текущей функции
- параметры “внешней” λ-функции
- указатель this
- глобальные – имеет смысл?

Пример «захвата» внешних локальных переменных

```
void CaptureExample(int& x)
{
        int external = 1;
        {
                int internal = 2;
                [&](){external++; internal++; x++; }();
        }
}
```

? Стоит ли злоупотреблять [&], а тем более [=] ?

C++14 Захват выражений

//требуется увеличить все элементы массива на указанное значение:

```
int ar[10] = { 1,2,3 };
std::for_each(ar, ar+10,
                [val = rand()%10](int& x){x += val; }
                  );
```

C++14 move семантика

```
std::string s("abc");
[str = std::move(s)]{ std::cout << str; }();
// s==???
```

Спецификация исключений в λ-функции

- Не генерирует исключения:
  [] (int x) **throw()** { … } - deprecated!!!
  [] (int x) **noexcept** { … }

- Генерирует bad_alloc:
  [] (int x) **throw(std::bad_alloc&)** { … } - deprecated!!!

Пример генерации и обработки исключения

```
vector<int> v = {1,2,3};
vector<int> ind = {0,-1, 2};
try{
   for_each(ind.begin(), ind.end(), [&v](int index) {
                                        v.at(index) = index;
                                        });
}
catch (out_of_range& e){
          cout << e.what();
};
```

Использование λ-функции в методе класса

- [**this**] // захват переменных текущего класса

```
class A {
        vector<int> m_v;
        int m_toApply;
public:
  void Apply() {
      for_each(m_v.begin(), m_v.end(), [this](int& x)
                                  { x+=m_toApply;});
       }
};
```

Захват переменных класса + переменных из внешнего контекста

```
class A {
        std::vector<int> m_v;
        int m_toApply;
public:
void Apply() {
        int n = 1;
        std::for_each(m_v.begin(), m_v.end(),
                [this, n](int& x) { x += m_toApply+n;});
}
};
```

Использование λ-функции с шаблонами

Сама λ-функция не может быть шаблонной, но ее можно использовать в шаблонных функциях:

```
template <typename T> void
                        print_all(const vector<T>& v) {
for_each(v.begin(), v.end(), [](const T& n) { cout<<n<<‘ ‘; });
}
```

Вложенные лямбда

```
int res = [](int x)
        { return [](int y)
                { return y * 2; }(x) + 3;
        }(5);
// 5*2 + 3
```

Обобщённые лямбда-функции (Generic lambdas) C++14

```
std::vector<int> v{1,2,3,4,5};
for_each( begin(v), end(v),
        [](const auto& x) { std::cout << x; } );
```

Обобщенные лямбда – аналог шаблона функтора

```
auto sum = [](auto x, auto y) {return x + y;};

int res1=sum(1, 2); //operator()<int,int>
double res2 = sum(1.1, 2.2); //operator()<double,double>
double res3 = sum(1.1, 2); //operator()<double,int>
```

auto – параметры лямбда функции C++14

```
std::vector<int> v = {1,2,3,4,5};

// C++11:
for_each( cbegin(v), cend(v),
[](decltype(*begin(v)) x) { x++; cout << x; } );

//C++14
for_each( begin(v), end(v),
[](auto& x) { x++; cout << x; } );
```

C++14 - **Generic lambdas**

```
std::vector<std::vector<int>> vv = { {1,2}, {1,2,3}, {1,2,3,4} };
//C++11
for_each(std::cbegin(vv), std::cend(vv),
        [](const std::vector<int>& v) {std::cout << v.size() << ' ';});
//только для вектора!!!

//C++14
auto lambdaSize = [](const auto& m) { return m.size(); }; //для любого
                                                  класса, в котором есть метод size()

for (const auto& x : vv){std::cout << lambdaSize(x) << ' ';}

std::list<std::list<int>> ll = { { 1,2 },{ 1,2,3 },{ 1,2,3,4 } };
for (const auto& x : ll) { std::cout << lambdaSize(x) << ' '; }
```