

Обзор стандарта C++ 11 и 14 версии

Выполнил: Ашихмин Н.А.,
студент ВМИ-301

Немного истории

C++98

- Первый настоящий стандарт
- Что обычно учатся в университетах

C++03

- Исправление ошибок 98

C++TR1

- Smart pointers
- Unsorted_map и прочая

C++11

- Мы изучаем сегодня

C++14

- Добеведение до ума 11

Features



Декларация типа с помощью auto

```
// с++03 решение
for (std::vector<std::map<int,
std::string>>::const_iterator it
= container.begin();
it != container.end();
++it)
{
// do something
}
```

```
// с++11 решение
for (auto it =
container.begin(); it !=
container.end(); ++it)
{
// do something
}
```

!Оператор auto обеспечивает автоматическое определение типа во время *компиляции*

Особенности auto

```
void foo()  
{  
  auto x = 5; // тип переменной x будет int  
  x = "foo"; // ошибка! не соответствие типов  
  
  auto y = 4, z = 3.14; // ошибка! нельзя объявлять  
  переменные разных типов  
}
```

Что такое decltype и с чем его едят?

```
int x = 5;  
double y = 5.1;  
  
decltype(x) foo;  
// int  
decltype(y) bar;  
// double  
  
decltype(x + y) baz;  
// double
```

decltype позволяет выбрать такой же тип, как у объекта в скобках

Финты ушами с auto, decltype и шаблонами

```
template <typename T, typename E>  
auto compose(T a, E b) -> decltype(a + b) {  
    return a + b;  
}  
auto c = compose(2, 3.14); // c - double
```

>> как закрытие вложенных шаблонов

//Как было раньше

```
std::vector<std::map<int, int>> foo;    // ошибка  
КОМПИЛЯЦИИ
```

```
std::vector<std::map<int, int> > foo;  // вполне  
корректный код
```


Range-based for

```
std::vector<int> foo;
```

```
// заполняем вектор
```

```
for (int x : foo)  
std::cout << x <<  
std::endl;
```

```
std::vector<std::pair<int,  
std::string>> container;
```

```
// ...
```

```
for (const auto& i : container)  
std::cout << i.second << std::endl;
```

Теперь есть foreach как в шарпе!

NULLPTR – полноценный указатель на пустой объект

//Раньше NULL был макросом языка C, который означал 0

Foo* foo = 0; // можно было писать так

```
void func(int x);
```

```
void func(const Foo* ptr);
```

```
// ...
```

```
func(0); //вызовется func(int x), хотя
```

мы могли подразумевать

```
//func(const Foo* ptr) с значением NULL
```

//Тут описана более серьезная проблема

```
std::vector<Foo*> foos;
```

```
// ...
```

```
std::fill(foos.begin(), foos.end(), 0); //страшная ошибка компиляции на полтора листа
```

//Решение C++ 11

```
std::vector<Foo*> foos;
```

```
// ...
```

```
std::fill(foos.begin(), foos.end(), nullptr); //все отлично!
```

Списки инициализации

```
//C++ 03
```

```
struct Struct  
{  
int x;  
std::string str;  
};
```

```
// инициализируем атрибуты  
структуры.
```

```
Struct s = { 4, "four" };
```

```
// инициализируем массив
```

```
int arr[] = { 1, 8, 9, 2,  
4 };
```

```
//C++ 11
```

```
std::vector<int> v = { 1, 5, 6,  
0, 9 };  
v.insert(v.end(), { 0, 1, 2, 3, 4  
});
```

```
class Foo
```

```
{  
public:  
// ...  
Foo(std::initializer_list<int>  
list);  
};
```

```
Foo::Foo(std::initializer_list<in  
t> list)  
{  
// do something  
}
```

Универсальная инициализация

```
class Foo
{
    public:
    // ...
    Foo(int x, double y,
        std::string z);
};
```

```
// ...
Foo::Foo(int x, double y,
std::string z)
{
    // do something
}
```

```
// ...
Foo one = { 1, 2.5, "one" };
Foo two{ 5, 3.14, "two" };
```

```
//эквивалентно вызову конструктора
Foo foo(1, 2.5, "one");
```

Не все только классам, не забудем и структуры

```
struct Foo
```

```
{
```

```
std::string str;
```

```
double x;
```

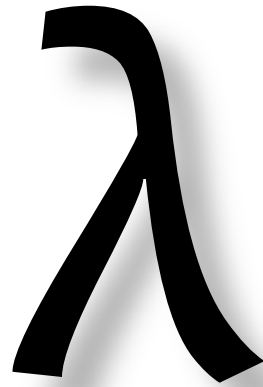
```
int y;
```

```
};
```

```
Foo foo{ "C++11", 4.0, 42 }; // {str, x, y}
```

```
Foo bar{ "C++11", 4.0 }; // {str, x}, y = 0
```

Lambdas

A large, bold, black Greek letter lambda symbol (λ) is centered on the page. The symbol has a slight shadow effect behind it, giving it a three-dimensional appearance.

Лямбда-функции

```
struct Comparator : public
std::binary_function<int, int,
bool>
{
    bool operator()(int lhs, int
rhs) const
    {
        if (lhs & 1 && rhs & 1)
            return lhs < rhs;
        return lhs & 1;
    }
};
```

```
std::sort(vec.begin(),
vec.end(),
Comparator());
```

```
std::sort(vec.begin(),
vec.end(), [](int lhs, int
rhs) -> bool {
    if (lhs & 1 && rhs & 1)
        return lhs < rhs;
    return lhs & 1;
});
```

Общий случай

```
[captures](arg1, arg2) -> result_type { /* code */ }
```

`arg1, arg2` - аргументы. То, что передается алгоритмом в функтор (лямбду).

`result_type` - тип возвращаемого значения. Это может показаться несколько непривычно, так как раньше тип всегда писали перед сущностью (переменной, функцией). Но к этому быстро привыкаешь.

`Captures` - список захвата: переменных внешней среды, которые стоит сделать доступными внутри лямбды. Эти переменные можно захватывать по значению и по ссылке.

Примеры использования captures

```
// по значению
```

```
std::sort(vec.begin(), vec.end(),  
[max](int lhs, int rhs) {  
    return lhs < max;  
});
```

```
// по ссылке
```

```
std::sort(vec.begin(), vec.end(),  
[&max](int lhs, int rhs) {  
    return lhs < max;  
});
```

```
// по значению
```

```
std::sort(vec.begin(), vec.end(),  
[=](int lhs, int rhs) {  
    return lhs < someVar;  
});
```

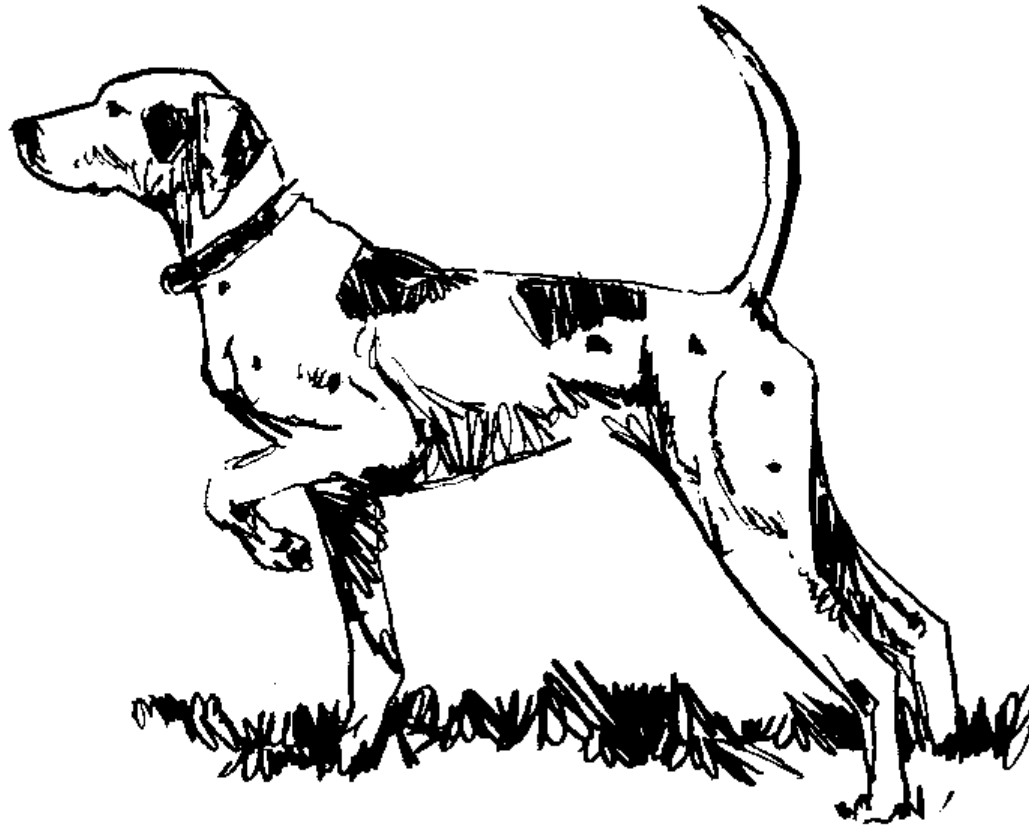
```
// по ссылке
```

```
std::sort(vec.begin(), vec.end(),  
[&](int lhs, int rhs) {  
    return lhs < otherVar;  
});
```

Пример использования лямбда функций

```
class Foo
{
    public:
    Foo() : _x(5) {}
    void doSomething() {
        // если вместо this поставить _x – будет ошибка!
        auto lambda = [this](int x) {
            std::cout << _x * x << std::endl;
        };
        lambda(4);
    }
    private:
    int _x;
};
```

Smart pointers



Проблемы обычных указателей

- Указатель не управляет временем жизни объекта, ответственность за удаление объекта целиком лежит на программисте. Проще говоря, указатель не «владеет» объектом.
- Указатели, ссылающиеся на один и тот же объект, никак не связаны между собой. Это создаёт проблему «битых» указателей – указателей, ссылающихся на освобождённые или перемещённые объекты.
- Нет никакой возможности проверить, указывает ли указатель на корректные данные, либо «в никуда».
- Указатель на единичный объект и указатель на массив объектов никак не отличаются друг от друга.
- Рассмотрим по порядку эти недостатки, а также инструменты, позволяющие от них избавиться.

Smart pointers

Smart pointer — это объект, работать с которым можно как с обычным указателем, но при этом, в отличие от последнего, он предоставляет некоторый дополнительный функционал (например, автоматическое освобождение закрепленной за указателем области памяти).

Умные указатели призваны для борьбы с утечками памяти, которые сложно избежать в больших проектах.

Auto_ptr

- Самый первый стандартный умный указатель в C++
- Позволяет бороться с утечками памяти
- Реализует разрушающее копирование, т.е. при $a=b$, b становится равно NULL

Auto_ptr

```
std::auto_ptr<int> x_ptr(new int(42));  
std::auto_ptr<int> y_ptr;  
  
// вот это нехороший и неявный момент  
// права владения ресурсов уходят в y_ptr и x_ptr  
начинает  
// указывать на null pointer  
y_ptr = x_ptr;  
  
// segmentation fault  
std::cout << *x_ptr << std::endl;
```

Unique_ptr

- Запрещает копирование.
- Изменение прав владения ресурсом осуществляется с помощью вспомогательной функции `std::move` (которая является частью механизма перемещения).
- Как `auto_ptr`, так и `unique_ptr` обладают методами `reset()`, который сбрасывает права владения, и `get()`, который возвращает сырой (классический) указатель.

Unique_ptr

```
std::unique_ptr<int> x_ptr(new int(42));  
std::unique_ptr<int> y_ptr;
```

```
// ошибка при компиляции
```

```
y_ptr = x_ptr;
```

```
// ошибка при компиляции
```

```
std::unique_ptr<int> z_ptr(x_ptr);
```

```
std::unique_ptr<int> x_ptr(new int(42));
```

```
std::unique_ptr<int> y_ptr;
```

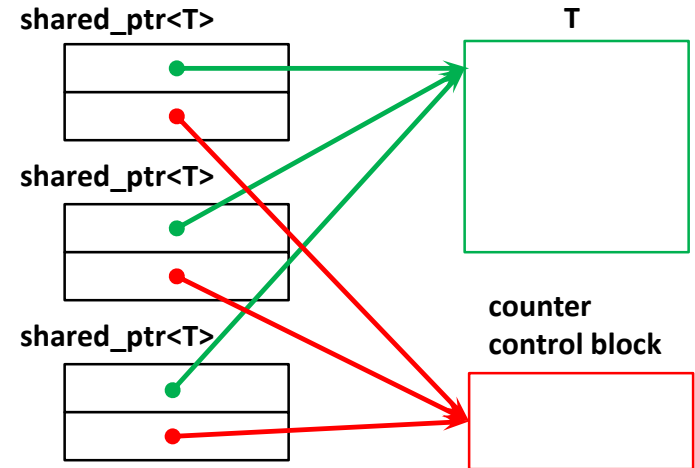
```
// также, как и в случае с ``auto_ptr``, права владения  
переходят
```

```
// к y_ptr, а x_ptr начинает указывать на null pointer
```

```
y_ptr = std::move(x_ptr);
```

Shared_ptr

- реализует подсчет ссылок на ресурс
- Также как и `unique_ptr`, и `auto_ptr`, данный класс предоставляет методы `get()` и `reset()`.



Shared_ptr

```
std::shared_ptr<int> x_ptr(new int(42));
std::shared_ptr<int> y_ptr(new int(13));
// после выполнения данной строчки, ресурс
// на который указывал ранее y_ptr (int(13)) освободится,
// а на int(42) будут ссылаться оба указателя
y_ptr = x_ptr;
std::cout << *x_ptr << "\t" << *y_ptr << std::endl;

// int(42) освободится лишь при уничтожении последнего
// ссылающегося
// на него указателя
auto ptr = std::make_shared<Foo>();
Foo *foo = ptr.get();
foo->bar();
ptr.reset();
```

Weak_ptr

- Слабый указатель
- Необходим для разрушения циклических зависимостей
- Используется в связке с shared_ptr

Weak_ptr

// пример циклической
зависимости

```
class Bar;
class Foo
{
public:
    Foo() { std::cout <<
"Foo()" << std::endl; }
    ~Foo() { std::cout <<
"~Foo()" << std::endl;
}

    std::shared_ptr<Bar>
    bar;
};
```

```
class Bar
{
public:
    Bar() { std::cout << "Bar()" <<
std::endl; }
    ~Bar() { std::cout << "~Bar()"
<< std::endl; }

    std::shared_ptr<Foo> foo;
};

int main()
{
    auto foo =
std::make_shared<Foo>();

    foo->bar =
std::make_shared<Bar>();
    foo->bar->foo = foo;

    return 0;
}
```

```
std::shared_ptr<Foo> ptr = std::make_shared<Foo>();  
std::weak_ptr<Foo> w(ptr);
```

```
if (std::shared_ptr<Foo> foo = w.lock())  
{  
    foo->doSomething();  
}
```

Спецификаторы



Спецификатор override

```
class Base
{
    public:
    virtual void doSomething(int
x);
};
// ...
class Derived : public Base
{
    public:
    virtual void doSomething(long
x);
};
```

```
class Base
{
    public:
    virtual void doSomething(int x);
};
// ...
class Derived : public Base
{
    public:
    virtual void doSomething(long x)
override;
};
```

Спецификатор `override` позволяет проверять существование метода с данной сигнатурой в базовом классе

Спецификатор final

```
class Base
{
public:
virtual void doSomething(int x) final;
};
// ...
class Derived : public Base
{
public:
virtual void doSomething(int x);
// ошибка!
};
class Base final {};
class Derived : public Base {};
// ошибка!
```

Спецификатор final позволяет запрещать в классах-наследниках переопределение определенных методов.

Спецификатор default

```
class Foo
{
public:
    Foo() = default;
    Foo(int x) { /* ... */ }
};
```

```
Foo obj;
```

Спецификатор default позволяет указать компилятору, что данный конструктор, деструктор он должен писать сам.

Спецификатор delete

```
class Foo
{
public:
    Foo() = default;
    Foo(const Foo&) = delete;
    void bar(int) = delete;
    void bar(double) {}
};
// ...
Foo obj;
obj.bar(5);          // ошибка!
obj.bar(5.42);      // ok
```

Спецификатор delete призван пометить те методы, работать с которыми нельзя. То есть, если программа ссылается явно или неявно на эту функцию — ошибка на этапе компиляции. Запрещается даже создавать указатели на такие функции.

Многопоточность



ПОТОКИ

```
#include <thread>

void threadFunction()
{
    // do smth
}

int main()
{
    std::thread thr(threadFunction);
    thr.join();
    return 0;
}
```

В C++11, работа с потокам осуществляется по средствам класса `std::thread` (доступного из заголовочного файла `<thread>`), который может работать с регулярными функциями, лямбдами и функторами. Кроме того, он позволяет вам передавать любое число параметров в функцию потока.

```
void threadFunction(int i, double
    d, const std::string &s)
{
    std::cout << i << ", " << d << ",
        " << s << std::endl;
}

int main()
{
    std::thread thr(threadFunction,
        1, 2.34, "example");
    thr.join();
    return 0;
}
```

В этом примере, thr — это объект, представляющий поток, в котором будет выполняться функция threadFunction(). Вызов join блокирует вызывающий поток (в нашем случае — поток main) до тех пор, пока thr (а точнее threadFunction()) не выполнит свою работу. Если функция потока возвращает значение — оно будет проигнорировано. Однако принять функция может любое количество параметров.

<thred>

Данная библиотека представляет следующие полезные функции:

- [get_id](#): возвращает id текущего потока
- [yield](#): говорит планировщику выполнять другие потоки, может использоваться при активном ожидании
- [sleep_for](#): блокирует выполнение текущего потока в течение установленного периода
- [sleep_until](#): блокирует выполнение текущего потока, пока не будет достигнут указанный момент времени

Блокировки

- [mutex](#): обеспечивает базовые функции [lock\(\)](#) и [unlock\(\)](#) и не блокируемый метод [try lock\(\)](#)
- [recursive mutex](#): может войти «сам в себя»
- [timed mutex](#): в отличие от обычного мьютекса, имеет еще два метода: [try lock for\(\)](#) и [try lock until\(\)](#)
- [recursive timed mutex](#): это комбинация `timed_mutex` и `recursive_mutex`


```
std::mutex g_lock;
void threadFunction(){
g_lock.lock();
std::cout << "entered thread " <<
std::this_thread::get_id() << std::endl;
std::this_thread::sleep_for(std::chrono::sec
onds(rand() % 10));
std::cout << "leaving thread " <<
std::this_thread::get_id() << std::endl;
g_lock.unlock();
}
int main() {
srand((unsigned int)time(0));
std::thread t1(threadFunction);
std::thread t2(threadFunction);
std::thread t3(threadFunction);
t1.join();
t2.join();
t3.join();
return 0;
}
```

Вывод:

```
entered thread 10144
leaving thread 10144
entered thread 4188
leaving thread 4188
entered thread 3424
leaving thread 3424
```

C++ 14

- C++14 — неофициальное название последней версии стандарта C++ ISO/IEC. C++14 можно рассматривать как небольшое расширение над C++11, содержащее в основном исправления ошибок и небольшие улучшения. Комитет разработки нового стандарта опубликовал черновик 15 мая 2013. Рабочая версия была опубликована 2 марта 2014 года, заключительный период голосования закрыт 15 августа 2014 года, а результат (единогласное одобрение) был объявлен 18 августа 2014 года.

Вывод типа возвращаемого значения для функций

```
auto DeduceReturnType(); // тип возвращаемого значение будет  
определён позже.
```

```
auto Correct(int i) {
```

```
if (i == 1)
```

```
return i; // в качестве типа возвращаемого значения выводится  
int
```

```
else
```

```
return Correct(i - 1) + i; // теперь можно вызывать
```

```
}
```

```
auto Wrong(int i) {
```

```
if (i != 1)
```

```
return Wrong(i - 1) + i; // неподходящее место для рекурсии. Нет  
предшествующего возврата.
```

```
else
```

```
return i; // в качестве типа возвращаемого значения выводится  
int
```

```
}
```

Разделители разрядов

```
auto integer_literal = 1'000'000;  
auto floating_point_literal = 0.000'015'3;  
auto binary_literal = 0b0100'1100'0110;  
auto silly_example = 1'0'0'000'00;
```

C++ 17

- В 2017 году ожидается новый стандарт, он еще не описан, и можно только гадать, что в него войдет. Но ожидаются следующие вещи:
- Поддержка utf-8 литералов
- Удаление некоторых старых функций и типов (auto_ptr, random_shuffle)
- Переработка unordered_map, unordered_set
- Исключение слова trigraphs
- Улучшение auto
- Окончательная унификация доступа к контейнерам
- И многое еще могут придумать...

First X3J16
meeting
Somerset, NJ, USA
(1990)



Completed
C++11
Madrid, Spain
(2011)



Completed
C++14
Issaquah, WA, USA
(2014)



Photo: Chandler Carruth and Olivier Giroux. License: tinyurl.com/9wn439f