# Type deduction
# Lamda functions

# Perfect Forwarding

# Перегрузка метода по категории значения объекта

Для перегрузки метода по категории значения объекта используются символы & и &&

```
class Sample
{
public:
    void categoryCheck() &
    {
        std::cout << "Lvalue" << std::endl;
    }

    void categoryCheck() &&
    {
        std::cout << "Rvalue" << std::endl;
    }
};
```

# Перегрузка метода по категории значения объекта

```
Sample getObject()
{
    return Sample();
}

int main()
{
    Sample object;

    object.categoryCheck();
    getObject().categoryCheck();
}
```

# Вывод типов шаблонов

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

f(expr);
```

Компилятор использует `expr` для вывода двух типов: `T` и `ParamType`

Например,

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

int x = 0;
f(x);
```

`T` будет выведен как `int`, `ParamType` как `const int&`

# Три возможные ситуации при выводе типа шаблона

Тип выводимый для T зависит не только от expr, но и от ParamType

- ParamType – указатель или ссылка, но не универсальная ссылка
- ParamType – универсальная ссылка
- ParamType – не указатель и не ссылка

# Случай 1: ParamType ссылка или указатель, но не универсальная ссылка

Правила вывода:

1. Если тип expr – ссылка, то ссылочная часть игнорируется
2. Затем тип expr сопоставляется с типом ParamType и выводится тип T

Например:

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

int x = 27;
const int cx = x;
const int& crx = x;

f(x); // T - int, ParamType - int&
f(cx); // T - const int, ParamType - const int&
f(crx); // T - const int, ParamType - const int&
```

# Случай 1: ParamType ссылка или указатель, но не универсальная ссылка

Другой пример:

```
template <typename T>
void f(const T& param); // param теперь const ссылка

int x = 27;           // Как и раньше
const int cx = x;     // Как и раньше
const int& rx = x;    // Как и раньше

f(x);  // T - int, ParamType const int&
f(cx); // T - int, ParamType const int&
f(rx); // T - int, ParamType const int&
```

# Случай 1: ParamType ссылка или указатель, но не универсальная ссылка

С указателями все работает точно также:

```
template <typename T>
void f(T* param); // param теперь указатель

int x = 27;           // как и раньше
const int* px = &x; // px - указатель на const int

f(&x);  // T - int, ParamType int*
f(px); // T - const int, ParamType const int*
```

# Путаница с T&&

В C++ существует небольшая путаница насчет T&&, так как в разных контекстах оно может обозначать rvalue – ссылки и универсальную ссылку. Например:

```
void f(Widget&& param);              // rvalue reference
Widget&& var1 = Widget();            // rvalue reference
auto&& var2 = var1;                  // not rvalue reference

template <typename T>
void f(std::vector<T>&& param);      // rvalue reference

template <typename T>
void f(T&& param);                   // not rvalue reference
```

# Универсальные ссылки / краткий обзор

Если T&& является универсальной ссылкой, то она может быть как lvalue – ссылкой, так и rvalue – ссылкой. Такая ссылка может возникнуть только в шаблонном коде, либо в auto при выводе типов.

Правила вывода для универсальных ссылок:

1. Если expr – lvalue, то и T, и ParamType выводятся как lvalue – ссылки. Это единственная ситуация, где T может быть ссылкой.
2. Если expr – rvalue, то применяются «обычные» правила из ситуации 1

# Случай 2: ParamType универсальная ссылка

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

int x = 27;
const int cx = x;
const int& crx = x;

f(x);   // x - lvalue, T - int&, ParamType - int&
f(cx);  // cx - lvalue, T - const int&, ParamType - const int&
f(crx); // crx - lvalue, T - const int&, ParamType - const int&
f(27);  // 27 - rvalue, T - int, ParamType - int&&
```

# Случай 3: ParamType не ссылка и не указатель

Правила вывода:

1. Если тип expr – ссылка, то ссылочная часть игнорируется
2. Если expr – const, игнорировать константность

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

int x = 27;
const int cx = x;
const int& crx = x;

f(x);   // T - int, ParamType - int
f(cx);  // T - int, ParamType - int
f(crx); // T - int, ParamType - int
```

# Случай 3: ParamType не ссылка и не указатель

Для переданных указателей игнорируется только const, который говорит, что указатель не может указывать ни на что другое, второй const сохраняется

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


const char* const ptr = "Fun with pointers";

f(ptr);   // T - const char*, ParamType - const char*
```

# Запомнить

- При выводе типа в шаблонах, ссылочные фактические параметры трактуются как не ссылочные
- При выводе типа с формальным параметром – универсальной ссылкой lvalue аргументы трактуются не обычным путем
- При выводе типа для формального параметра «по значению» модификатор const игнорируется

# Что будет выведено на экран и почему?

```
void increase(int& r) { r++; }

template <typename Function, typename Parameter>
void apply(Function f, Parameter p)
{
    f(p);
}

int main()
{
    int i = 0;

    apply(increase, i);
    std::cout << i << std::endl;
}
```

# Reference Wrapper

- std::ref(T&) – находится в <functional> и может неявно приводится к (T&)

```
void increase(int& r) { r++; }

template <typename Function, typename Parameter>
void apply(Function f, Parameter p)
{
    f(p);
}

int main() {
    int i = 0;

    apply(increase, std::ref(i));
    std::cout << i << std::endl;
}
```

# Вывод типа для auto

Правила вывода типа для auto точно такие же, как и для шаблонов с одним исключением. Посмотрим примеры:

```
auto x = 27;
const auto cx = x;
const auto& crx = x;

auto&& uref1 = x;
auto&& uref2 = cx;
auto&& uref3 = 27;
```

# Исключение для вывода типа auto

Вспомним варианты синтаксиса инициализации

```
int x1 = 27;
int x2(27);
int x3 = { 27 };
int x4 {27};
```

Четыре варианта – один результат

```
auto x1 = 27;       // int
auto x2(27);        // int
auto x3 = { 27 };   // !!! std::initializer_list <int>
auto x4 {27};       // !!! std::initializer_list <int>
```

Шаблонная функция не скомпилируется с { 27 }

# Синтаксис λ - функции / замыкания

1. [capture] (params) mutable exception_attribute → ret_type { body }
2. [capture] (params) → ret_type { body }
3. [capture] (params) { body }
4. [capture] { body }

1. Полное определение
2. Константное определение замыкания: объекты, захваченные по значению не могут быть изменены
3. Опущен возвращаемый тип, компилятор сам его выведет.
4. Опущен список параметров, может использоваться только без спецификаторов

# Capture λ - функции

- Этот раздел λ - функции позволяет захватывать внешние переменные как по значению, так и по ссылке
- Возможные варианты:

1. [a, &b] – a захвачено по значению, b – по ссылке
2. [this] – захватывает указатель this текущего объекта
3. [&] – захватывает все локальные переменные по ссылке
4. [=] – захватывает все локальные переменные по значению

# Правила вывода возвращаемого значения λ - функции

- (до C++14)
  Если функция состоит из одной строчки return, то компилятор выводит тип возвращаемого значения по этой строчке, иначе – тип возвращаемого значения void.
- (с C++14)
  Компилятор находит строчку с return и выводит тип возвращаемого значения из неё

# Тип λ - функции

Тип λ - функции знает только компилятор, но это не значит, что мы не можем хранить её в переменной, type - deduce позволяет нам работать с ним, не зная его.

```
auto lambda1 = []{};
auto lambda2 = [](int left, int right) mutable noexcept -> bool { return left < right;
};
```

# Пример λ - функции

```
#include <iostream>

template <typename T, typename Comparator>
bool logCompare(const T& left, const T& right, Comparator comp) {
    static std::uint64_t count = 0;
    std::cout << "compare " << ++count << " times" << std::endl;
    return comp(left, right);
}

class Comparator {
public:
    bool operator()(int left, int right) { return left < right; }
};

bool compare(int left, int right) {
    return left < right;
}
```

# Пример λ - функции

```
int main()
{
    std::cout << logCompare(3, 2, compare) << std::endl;
    std::cout << logCompare(2, 2, Comparator()) << std::endl;
    std::cout << logCompare(2, 3, [](int left, int right) noexcept { return left < right; })
    << std::endl;
}
```

# Пример захвата переменных

```
class Example {
    float field1;
    int field2;
    char field3;
public:
    Example() noexcept : field1(0), field2(0), field3('a') {}

    void logThroughLambda() const noexcept {
        auto logLamda = [this] { std::cout << field1 << ' ' << field2 << ' ' << field3
        << std::endl; };

        logLamda();
    }
};

int main() {
    Example example; example.logThroughLambda();
}
```

# Пример захвата переменных

```
void assign(int& y, int x) noexcept
{
    [x, &y]() noexcept { y = x; }();
}

int main()
{
    int a = 3, b = 4;

    std::cout << "before: " << a << ' ' << b << std::endl;
    assign(a, b);
    std::cout << "after:" << a << ' ' << b << std::endl;
}
```

# Для C++11 сказочка с выводом auto - типов закончилась

А вот C++14 расширяет возможность использования auto, позволяя использовать auto в возвращаемых значениях функций и в формальных параметрах λ – функций.

Причем в этих контекстах вывод типа для auto совсем ничем не отличается от шаблонного вывода

```
auto get4()
{
    return {4};   <-- ошибка компиляции, как в шаблонном deduce type
}
```

# auto в параметрах λ – функций

```
int main()
{
    int v = 0;
    auto resetV = [&v](auto newValue) { v = newValue; };

    resetV(4);
}
```

# Decltype вывод типов

Decltype объявляет тип, как auto, но по выражению, переданному в него
Синтаксис:

decltype(expr)

```
int x = 2;
decltype(x) y = 3;
```

# Примеры очевидного поведения
## decltype

```
bool f(const Widget& w); // decltype(w) - const Widget&
                         // decltype(f) - bool (const Widget&)

const int i = 0; // decltype(i) - const int

struct Point { int x, y; }; // decltype(Point::x) - int
                            // decltype(Point::y) - int


Widget w; // decltype(w) - Widget
          // decltype(f(w)) - bool

template<typename T> // simplified version of std::vector
class vector {
public:
    T& operator[](std::size_t index);
};

vector <int> v; // decltype(v) - vector<int>
                // decltype(v[0]) - int&
```

# Правила вывода decltype

- 1) Насколько возможно не изменять тип своего аргумента
- 2) Для lvalue выражения типа T отличного от простого имени объекта всегда выводится T&

# Использование decltype в C++11

В C++11 decltype наиболее часто использовался в шаблонных функциях, где тип возвращаемого значения зависел от передаваемых им аргументов.

# Trailing return type syntax

```
auto имяФункции(Параметры...) -> тип_возвращаемого значения
```

```
template <typename Container, typename Index>
auto authAndAccess(Container& c, Index i) -> decltype(c[i])
{
    authUser();
    return c[i];
}
```

# В C++14 мы можем избежать такого синтаксиса

```
template <typename Container, typename Index>
auto authAndAccess(Container& c, Index i)
{
    authUser();
    return c[i];
}
```

Но, к сожалению, следующий вызов не скомпилируется (почему?)

```
std::vector <int> d;
authAndAccess(d, 5u) = 10;
```

# Исправляем: auto с правилами decltype

```
template <typename Container, typename Index>
decltype(auto) authAndAccess(Container& c, Index i) // Почти хорошо
{
    authUser();
    return c[i];
}
```

```
std::vector <int> d;
authAndAccess(d, 5u) = 10; <-- теперь компилятор вернет ссылочный тип
```

## А как быть с таким вариантом? (Не Visual Studio)

```
std::vector <int> makeVector() noexcept;
// ...
authAndAccess(makeVector(), 5) = 10;
```

# Исправляем: добавляем универсальную ссылку

```
template <typename Container, typename Index>
decltype(auto) authAndAccess(Container&& c, Index i) // Почти отлично
{
    authUser();
    return c[i];
}
```

# Perfect forwarding

```
template <typename Function, typename Arg>
void apply(Function f, Arg&& arg)
{
    f(arg);
}
```

Какой недостаток у данной функции?

# Perfect forwarding

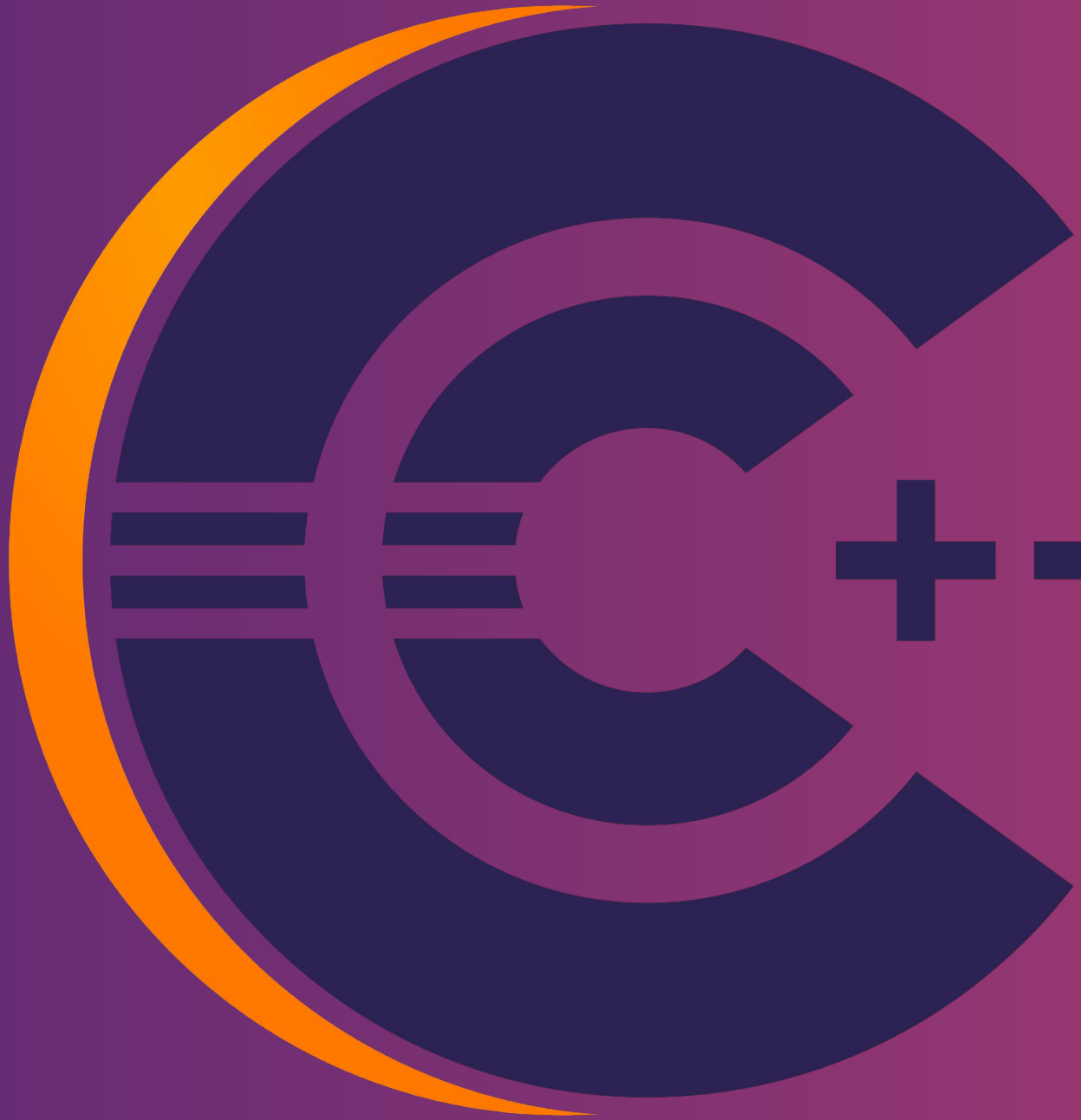
```
template <typename Function, typename Arg>
void apply(Function f, Arg&& arg)
{
    f(std::forward <Arg> (arg));
}
```

- Если arg lvalue – перемещения не будет
- Если arg rvalue – перемещение будет

std::forward – находится в <functional>

# Теперь вернемся к примеру с authAndAccess

```
template <typename Container, typename Index>
decltype(auto) authAndAccess(Container&& c, Index i) // Отлично
{
    authUser();
    return std::forward <Container>(c)[i];
}
```



C++

На этом всё