



*Spring 2021*

*[compscicenter.ru](http://compscicenter.ru)*

Башарин Егор, [t.me/egorbasharin](https://t.me/egorbasharin)

Грабовой Филипп, [t.me/phil-grab](https://t.me/phil-grab)

# Lecture III

auto, decltype, macros

# auto as type specifier

Use cases:

```
void foo(std::map<int, int>& m) {  
    std::map<int, int>::iterator it = m.begin();  
    auto similarIt = m.begin();  
    /* ... */  
}  
  
template<typename T, typename S>  
void bar(T lhs, S rhs) {  
    auto prod = lhs * rhs;  
    /* ... */  
}
```

1. упрощает написание конструкций для типов
2. позволяет использовать тип, зависимый от шаблонных параметров

# auto type deduction

- вывод типов происходит по тем же правилам, что и для шаблонных аргументов функции
  - упрощенное правило: отбрасываются крайние ref и cv-qualifiers
  - в деталях: [cprreference](#)
  - отличие: при использовании {, }

```
auto var = {1, 2, 3}; // -> std::initializer_list<int>

template<typename T>
void foo(T t);

foo({1, 2, 3}); // cannot be deduced
```

# Examples

```
int x = 42;
```

```
auto a = x;
```

```
auto& b = x;
```

```
const auto c = x;
```

```
const auto& d = x;
```

```
int x = 42;
```

```
auto a = x;           // int
```

```
auto& b = x;          // int&
```

```
const auto c = x;     // const int
```

```
const auto& d = x;    // const int&
```

```
const int x = 42;
```

```
auto a = x;
```

```
auto& b = x;
```

```
const auto c = x;
```

```
const auto& d = x;
```



```
const int x = 42;
```

```
auto a = x;           // int
```

```
auto& b = x;          // const int& (!)
```

```
const auto c = x;    // const int
```

```
const auto& d = x;   // const int&
```

```
int x = 42;  
int& rx = x;
```

```
auto a = rx;  
const auto b = rx;
```

```
auto& c = rx;  
const auto& d = rx;
```

```
int x = 42;  
int& rx = x;  
  
auto a = rx;           // int  
const auto b = rx;     // const int  
  
auto& c = rx;          // int&  
const auto& d = rx;    // const int&
```

```
int x = 42;  
const int& crx = x;  
  
auto a = crx;  
const auto b = crx;  
  
auto& c = crx;  
const auto& d = crx;
```

```
int x = 42;  
const int& crx = x;  
  
auto a = crx;           // int  
const auto b = crx;     // const int  
  
auto& c = crx;           // const int&  
const auto& d = crx;    // const int&
```

```
struct T { int i; };  
const T* t = new T{42};
```

```
auto a = t->i;  
const auto b = t->i;
```

```
auto& c = t->i;  
const auto& d = t->i;
```

```
struct T { int i; };  
const T* t = new T{42};  
  
auto a = t->i;           // int  
const auto b = t->i;     // const int  
  
auto& c = t->i;           // const int&  
const auto& d = t->i;    // const int&
```

```
int func();
```

```
auto a = func();
```

```
const auto b = func();
```

```
auto& c = func();
```

```
const auto& d = func();
```

```
auto&& e = func();
```



```
int func();

auto a = func();           // int
const auto b = func();     // const int

auto& c = func();           // compile error (!)
const auto& d = func();    // const int&

auto&& e = func();          // int&&
```

```
int& func();
```

```
auto a = func();
```

```
const auto b = func();
```

```
auto& c = func();
```

```
const auto& d = func();
```

```
auto&& e = func();
```

```
int& func();
```

```
auto a = func(); // int
```

```
const auto b = func(); // const int
```

```
auto& c = func(); // int&
```

```
const auto& d = func(); // const int&
```

```
auto&& e = func(); // int&
```

```
const int& func();
```

```
auto a = func();
```

```
const auto b = func();
```

```
auto& c = func();
```

```
const auto& d = func();
```

```
auto&& e = func();
```

```
const int& func();
```

```
auto a = func();           // int  
const auto b = func();    // const int
```

```
auto& c = func();          // const int&  
const auto& d = func();    // const int&
```

```
auto&& e = func();         // const int&
```

Check point

# auto with universal ref?

```
template <class T>
void func(std::vector<T>& vs, const T& value) {
    for (auto& v : vs) {
        v = value;
    }
}

std::vector<int> vs{1, 2, 3};
func(vs, 4);
```

```

template <class T>
void func(std::vector<T>& vs, const T& value) {
    for (auto& v : vs) {
        v = value;
    }
}

std::vector<bool> vs(2, false);
func(vs, true);

```

- error: non-const lvalue reference to type 'std::\_Bit\_reference' cannot bind to a *temporary* of type 'std::\_Bit\_iterator::reference'



# using universal ref

```
template <class T>
void func(std::vector<T>& vs, const T& value) {
    for (auto && v : vs) { // universal ref!
        v = value;
    }
}

std::vector<bool> vs(2, false);
func(vs, true);
```

# auto in structured binding

*позже*

# auto in abbreviated function template (C++20)

```
void foo(auto arg) { /* ... */ }

template <class T, class U>
void bar(T x, U y, auto z) { /* ... */ }
```

- `foo`, `bar` определяют шаблонные функции
- последние параметры сигнатуры — также параметры шаблонов
  - т.е. всего их 1 у `foo` (не 0) и 3 у `bar` (не 2)

# auto in return type

```
auto gcd(int a, int b) {  
    if (b == 0) { return a; }  
    return gcd(b, a % b);  
}
```

- те же правила для вывода типов по `return`-выражениям
- если несколько `return <smth>`  $\Rightarrow$  везде должен выводиться одинаковый тип

# auto in return type in template function

```
template <class F, class Arg>
auto invoke(F&& f, Arg&& arg) {
    return f(std::forward<Arg>(arg));
}

int foo(int x) { return x; }
auto res = invoke(foo, 10);
```

- проблема с: `int& bar(int& x)`
- варианты `auto&`, `auto&&` в return type — потребуют конструкторов
- что делать?

**decltype**

# Syntax

```
decltype( entity )
```

```
decltype( expression )
```

- ПОЗВОЛЯЕТ ВЫВЕСТИ ТОЧНЫЙ ТИП
- `entity` — переменная или поле класса (без внешних скобок)
- `expression` — другие варианты (не будут вычисляться! `unevaluated`)

# decltype w/ entity

```
int x = 101;  
using T1 = decltype(x);    // int  
  
const int cx = 101;  
using T2 = decltype(cx);   // const int  
  
const int& crx = 101;  
using T = decltype(crx);   // const int&  
  
struct T { int i = 101; };  
const T* t = new T;  
using T4 = decltype(t->i); // int (!)
```



# decltype entity to expression

```
int x = 101;  
  
// entity  
using T1 = decltype(x);    // int  
  
// expression  
using T2 = decltype(x);    // int&
```

# decltype w/ expression, lvalue

```
int x = 101;  
using T2 = decltype( (x) );
```

- выражение типа  $T$  категории `lvalue`  $\Rightarrow T\&$

# decltype w/ expression, prvalue

```
struct T { /* ... */ };  
  
T makeT();  
  
using Type = decltype(makeT());
```

- выражение типа  $T$  категории  $\text{prvalue} \Rightarrow T$

# decltype w/ expression, xvalue

```
struct T { ... };  
T&& func();  
  
T t;  
  
using T1 = decltype(std::move(t));  
using T2 = decltype(func());
```

- выражение типа  $T$  категории  $xvalue \Rightarrow T\&\&$

**decltype + auto**

# in return type in template function

```
template <class F, class Arg>
auto invoke(F&& f, Arg&& arg)
    -> decltype( f(std::forward<Arg>(arg)) )
{
    return f(std::forward<Arg>(arg)) ;
}
```

# decltype(auto)

- КОМПИЛЯТОР ВЫВОДИТ ТИП, НО ПО ПРАВИЛАМ decltype
  - с выражением в инициализаторе

```
int& foo();  
decltype(auto) i = foo();    // i is int&  
                             // it is decltype(foo())  
  
// => use in invoke:  
  
template <class F, class Arg>  
decltype(auto) invoke(F&& f, Arg&& arg) {  
    return f(std::forward<Arg>(arg));  
}
```

# decltype(auto) caution

- действительно используется выражение из инициализатора

```
decltype(auto) getIntEntity() {  
    int val = 10;  
    return val;    // returns int  
}  
  
decltype(auto) getIntExpression() {  
    int val = 10;  
    return (val);  // returns int&  
}
```



**macros**

# notes

- в C++ есть инструменты помимо макросов (в C макросы решали похожие задачи):
  - константы
  - шаблоны (+variadic)
  - inline-функции
  - constexpr

# notes

- макросы обрабатываются препроцессором до компиляции
  - поверх содержимого файла, в отрыве от синтаксиса C++
  - работа над строками-токенами
  - нет информации о типах, о пространствах имен и др.

# usage

- как используют макросы:
  - директивы: `#include` (+ guards), `#pragma`
  - конкатенация токенов
  - отладка/тестирование
  - условная компиляция
  - повторяемый/переиспользуемый код
- рассмотрим примеры: макро-константы, директивы, макро-функции

# макро-константы

```
#define <identifier name> [value]
```

- символ # — первый в строке
  - МОЖНО вставлять пробелы до и между # и `define`
  - `[value]` — ОПЦИОНАЛЬНАЯ часть
  - результат вне инструкций препроцессора — подстановка

```
#define PI 3.1415926
#define PI_PLUS_1 PI + 1

auto x = PI_PLUS_1 * 5; // -> PI + 1 * 5 (!)
// => braces
#define PI_PLUS_2 (PI + 2)
```

# ДИРЕКТИВЫ ДЛЯ УСЛОВНОЙ КОМПИЛЯЦИИ

- удобно для кросс-платформенной разработки + разного окружения

```
#ifdef WIN32
# ifdef _MSC_VER > 1800
    // WIN32 platform + MS compiler specific code
# endif // _MSC_VER
#else
    // WIN32 platform specific code
#endif // WIN32

// boolean expressions
#if defined (__linux__) && !defined(NDEBUG)
    // linux specific code
#endif
```

# teches

- include guards: `#ifndef ... #define ... #endif`
  - похожая директива: `#pragma once`
- посмотреть выхлоп препроцессора: `$ clang++ -E ...`
- определить в командной строке: `$ clang++ -DVALUE=100500 ...`

# макро-функции

```
#define MACRO_NAME(arg1, arg2, ...) [code to expand to]
```

- Нет механизма перегрузок (\*)
- Не работает с рекурсией
- токен `,` — разделитель аргументов
  - $\Rightarrow$  В `FOO(std::map<int, int>)` два аргумента: `std::map<int` и `int>`
- результат раскрытия — подстановка
  - не вычисляет аргументы

```
#define MAX(a,b) a > b ? a : b
MAX(x + 4, x--);           // ->
x + 4 > x-- ? x + 4 : x--;  // wrap + omit side effects
```



# МИМИКРИРОВАНИЕ ПОД ОБЫЧНЫЕ ФУНКЦИИ

- при использовании в коде в конце требовать ;
- чтобы несколько выражений в макросе работали корректно

```
// any multiple statements macro
#define SWAP(x, y) (x) ^= (y); (y) ^= (x); (x) ^= (y);

if (x > y)
    SWAP(x, y); // check #1
do_something();

if (x > y)
    SWAP(x, y); // check #2
else
    SWAP(y, z); // check #3
```

# МИМИКРИРОВАНИЕ – ВОЗМОЖНЫЕ решения

```
if (a > b)
    SWAP(x, y);

if (a > b)
    SWAP(a, b);
else
    ...

// solutions:
#define SWAP(x, y) do { x ^= y; y ^= x; x ^= y; } while(
// or
#define SWAP(x, y) (x ^= y, y ^= x, x ^= y, (void)0)
```

# token to string literal

- #ARG в теле макроса → оборачивает *токен* в `"`, `"`
  - ⇒ токен, обозначающий макрос, не раскрывается (нужна еще одна фаза)

```
#define STR_IMPL(X) #X
#define STR(X) STR_IMPL(X)

#define SHOW(X) \
do{ cout << STR_IMPL(X) << ": " << (X) << endl; }while(0)

#define VALUE 2.718281828459045

int main() {
    int x(2), y(5);
    SHOW(x + y); // x + y: 7

    cout << STR_IMPL(VALUE) << endl; // VALUE
    cout << STR(VALUE) << endl; // 2.718281828459045
}
```

# tokens concatenation

- `Argi##Argj` в теле макроса — соединяет токены в одну строку (оборачивая в ", ")
  - также нужна двухфазность

```
#define CONCAT_IMPL(X, Y) X##Y
#define CONCAT(X, Y) CONCAT_IMPL(X, Y)

#define DECL_PTR(T) \
    using CONCAT(T, _ptr) = unique_ptr<T>

DECL_PTR(int);

int main() {
    int_ptr p;
}
```

# predefined defines

- `_LINE_` — номер текущей строки
  - `_FILE_` — название текущего файла
  - `_FUNCTION_` — название функции
  - `_linux_`, `WIN32` и т.д. — настройка платформы
  - `_COUNTER_` — самоинкрементирующаяся константа

# multiline verify

```
#define Verify(expr) \
do{ if (!(expr)) \
{ \
    std::stringstream ss; \
    ss << "Verification failed: " \
        << STR(expr) << ". \"\" \
        << __FUNCTION__ << "\" in \" << \
        __FILE__ << ":" << __LINE__;\
    LogError(ss.str()); \
    throw verify_error(ss.str()); \
} } while (0, 0)
```

# вариативный макрос

- все параметры макроса можно собрать в один
  - дальше их можно только пробросить, нельзя итерироваться и т.д.

```
#define DECL_PROCESS(...) \
void process(__VA_ARGS__ const& object) \
{ \
    using type = __VA_ARGS__; \
    /* ... */ \
}

DECL_PROCESS(map<int, double>);
```

# перегрузка для макросов

```
// что понадобится иметь на входе
#define F002(x,y) ...
#define F003(x,y,z) ...

// что хочется писать ("фасад"):
#define F00(x,y) ...           // -> use F002
#define F00(x,y,z) ...        // -> use F003
```



# перегрузка для макросов, реализация

```
#define FOO2(x,y) ...
#define FOO3(x,y,z) ...

#define GET_MACRO(_1,_2,_3,NAME,...) NAME
#define FOO(...) GET_MACRO(__VA_ARGS__, /
                           FOO3, FOO2)(__VA_ARGS__)

FOO(x,y) // -> FOO2
FOO(x,y,z) // -> FOO3
```