C++

Лекция 15

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. облегчает 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:
  - отбрасываются крайние ref и cv-qualifiers
  - в т.ч. `auto&&` — работает по правилам универсальных ссылок
- исключение 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;
```

```cpp
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;
```

```cpp
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&
```

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

auto a = rx;
const auto b = rx;

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

```cpp
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&
```

```cpp
int x = 42;
const int& crx = x;

auto a = crx;
const auto b = crx;

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

```cpp
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;
```

```cpp
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();
```

```cpp
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();
```

```cpp
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();
```

```cpp
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&
```

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

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

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

- `foo`, `bar` - шаблонные функции

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>` ⇒ 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);
```

Проблемы:

- если f возвращает ссылочный тип (с `auto` теряется ссылочность)
- варианты `auto&`, `auto&&` в return type у invoke — не поддерживают соответственно rvalue и lvalue результатах `f(...)`

decltype

Syntax

```
decltype( entity )
```

```
decltype( expression )
```

- позволяет вывести точный тип
- entity — переменная или поле класса
  - результат — тип, с которым объявлены переменная/поле
- expression — другие варианты (не будут вычислятся!)
  - результат — тип на основании типа и value category выражения

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 (!)
```

entity vs 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` ⇒ `T&`

decltype w/ expression, prvalue

```
struct T { /* ... */ };

T makeT();

using Type = decltype(makeT());
```

- выражение типа `T` категории `prvalue` ⇒ `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` ⇒ `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

in C++ & C

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

применение

- 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
```

See example

teches

- include guards: `#ifndef … #define … #endif`
  - похожая директива: `#pragma once`
- посмотреть выхлоп препроцессора: `$ clang++ -E …`
- определить в командной строке: `$ clang++ -DVALUE=42 …`

макро-функции

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

- нет готового механизма перегрузок (*)
- не работает с рекурсией
- токен , — разделитель аргументов: в `FOO(std::map<int, int>)` их два
- результат раскрытия — подстановка

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

как мимикрировать под обычные функции

- при использовании в коде в конце ожидать обязательный ;
- поддержить корректную работу нескольких выражений в макросе

```
// 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

писать макросы-функции в определенном стиле

```
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(0)
// 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 (подстановки значения нет; нужна еще одна фаза: доп вызов макроса)

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

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

#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 (если нужно поддержать макросы-аргументы)

```
#define CONCAT(X, Y) 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();                     \
} } while (0)
```

See example on godbolt

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

// что хочется писать ("фасад"):
#define FOO(x,y) ...      // -> use FOO2
#define FOO(x,y,z) ...    // -> use FOO3
```

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
```