**Правила вывода типа шаблона.**

Пусть имеется некоторая шаблонная функция:

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

Тогда ее вызов можно записать так:

f(**expr**);

В процессе компиляции компилятор использует expr для вывода двух типов: T и ParamType. Последний чаще не совпадает с T, т.к. дополнительно может содержать cv-квалификаторы и квалификатор ссылки.

|  |  |  |
| --- | --- | --- |
|  | ParamType является | Правила вывода типа |
| **1** | Указатель или lvalue-ссылка | Если expr является ссылкой, ссылочная часть игнорируется. |
| **2** | универсальная ссылка (в шаблоне записывается как rvalue-ссылка) | Если expr является lvalue, как T, так и ParamType выводятся как lvalue-ссылки.  Если expr является rvalue, действует правило, аналогичное правилу 1. |
| **3** | Ни указатель, ни ссылка (по значению) | Если expr является ссылкой, ссылочная часть игнорируется.  cv-квалификаторы также игнорируются. |

*Пример*. Вывод типа по правилу 1.

|  |  |  |
| --- | --- | --- |
| template<typename T> void f(T& param); | | |
| **expr** | **ParamType** | **T** |
| int x = 27; f(x); | int& | int |
| const int cx = x; f(cx); | const int& | const int |
| const int& rx = x; f(rx); | const int& | const int |

|  |  |  |
| --- | --- | --- |
| template<typename T> void f(const T& param); | | |
| **expr** | **ParamType** | **T** |
| int x = 27; f(x); | const int& | int |
| const int cx = x; f(cx); | const int& | int |
| const int& rx = x; f(rx); | const int& | int |

|  |  |  |
| --- | --- | --- |
| template<typename T> void f(T\* param); | | |
| **expr** | **ParamType** | **T** |
| int x = 27; f(&x); | int\* | int |
| const int \*px = &x; f(px); | const int\* | const int |

*Пример*. Вывод типа по правилу 2.

|  |  |  |
| --- | --- | --- |
| template<typename T> void f(T&& param); | | |
| **expr** | **ParamType** | **T** |
| int x = 27; f(x); | int& | int& |
| const int cx = x; f(cx); | const int& | const int& |
| const int& rx = x; f(rx); | const int& | const int& |
| f(27); | int&& | int |

*Пример*. Вывод типа по правилу 3.

|  |  |  |
| --- | --- | --- |
| template<typename T> void f(T param); | | |
| **expr** | **ParamType** | **T** |
| int x = 27; f(x); | int | int |
| const int cx = x; f(cx); | int | int |
| const int& rx = x; f(rx); | int | int |

|  |  |  |
| --- | --- | --- |
| template<typename T> void f(T param); | | |
| **expr** | **ParamType** | **T** |
| const char\* const p = “text”; f(p); | const char\* (NB: const для p игнорируется, а для строки остается) | const char\* |

*Пример*. Вывод типа шаблона при передачи ссылки на массив.

|  |  |  |
| --- | --- | --- |
| template<typename T> void f(T& param); | | |
| **expr** | **ParamType** | **T** |
| const char name[] = “Briggs”;  f(name); | const char(&)[13] (NB: ссылка на массив) | const char[13] |

Возможность объявлять ссылки на массивы позволяет создать шаблон, который выводит количество элементов, содержащихся в массиве:

template<typename T, std::size\_t N> constexpr std::size\_t arraySize(T (&)[N]) noexcept

{ return N; }

Объявление этой функции как constexpr делает ее результат доступным во время компиляции. Это позволяет объявить, например, массив с таким же количеством элементов, как и у второго массива, размер которого вычисляется из инициализатора в фигурных скобках.

*Пример*. Вывод типа шаблона при передачи ссылки на функцию.

|  |  |  |
| --- | --- | --- |
| template<typename T> void f(T& param); | | |
| **expr** | **ParamType** | **T** |
| void someFunc(int, double);  f(someFunc); | void(&)(int, double) | void(int, double) |

Здесь работает также, как и в случае вывода типа для массивов. Если функция передается по значению, она преобразуется в указатель, а если по ссылке – то в ссылку на функцию.

Таким образом, для аргументов массивов и функций можно сформулировать правило №4:

В процессе вывода типа шаблона аргументы, являющиеся именами массивов или функций, преобразуются в указатели, если только они не использованы для инициализации ссылок (NB: в таком случае тип аргумента шаблона остается ссылкой).

**Вывод типа auto.**

Вывод типа auto представляет собой вывод типа шаблона. Когда переменная объявлена с использованием ключевого слова auto, оно играет роль T в шаблоне, а спецификатор типа переменной действует как ParamType. Для вывода типов компилятор действует так, как если бы для данного объявления имелся шаблон, а также вызов этого шаблона с соответствующим инициализирующим выражением expr.

*Пример*.

|  |  |  |  |
| --- | --- | --- | --- |
| Объявление | Концептуальный шаблон | ParamType | T |
| auto x = 27;  func\_for\_x(x);  // правило 3 | template<typename T>  void func\_for\_x(T param) | int | int |
| const auto cx = x;  func\_for\_cx(cx);  // правило 3 | template<typename T>  void func\_for\_cx(const T param);  // const будет проигнорирован | int | int |
| const auto &rx = x;  func\_for\_rx(rx);  // правило 1 | template<typename T>  void func\_for\_rx(const T& param); | conts int& | int |

NB: Чтобы не было путаницы важно уяснить: ParamType, записанный для шаблона (в т.ч. для концептуального) может не совпадать с фактическим типом ParamType, полученным в процессе вывода типа. Т.е. это понятие в разных контекстах имеет разное значение: в шаблоне записали одно, но компилятор выведет другой тип по одному из трех правил вывода типа. Программисту важно уметь выводить именно фактический ParamType.

Таким образом, можно видеть, что вывод типа auto работает подобно выводу типа шаблона. По сути это две стороны одной медали. Они отличаются только в одном: если при инициализации использовались фигурные скобки, выведенным типом является std::initializer\_list.

C++14 допускает применения auto для указания того, что возвращаемый тип функции должен быть выведен, а кроме того, лямбда выражения C++14 могут использовать auto в объявлениях параметров. Однако, такое применение auto использует вывод типа шаблона, а не вывод типа auto. Таким образом, функция с возвращаемым типом auto, которая возвращает инициализатор в фигурных скобках, компилироваться не будет. То же самое справедливо и тогда, когда auto используется в спецификации типа параметра в лямбда-выражении C++14.

**decltype**.

Основное применение decltype в C++11 – объявление шаблонов функций, в которых возвращаемый тип функции зависит от ее параметров. Т.е. decltype применяется для вычисления возвращаемого типа.

*Пример*.

template<typename Container, typename Index>

auto authAndAccess (Container& с, Index i)

- > decltype (c[i])

{

authenticateUser();

return c[i];

}

Использование auto перед именем функции не имеет ничего общего с выводом типа. На самом деле оно указывает, что использован синтаксис C++11 – завершающий возвращаемый тип (trailing return type), т.е. что возвращаемый тип функции будет объявлен после списка параметров (после «->»). Завершающий возвращаемый тип обладает тем преимуществом, что в спецификации возвращаемого типа могут использоваться параметры функции. Если бы возвращаемый тип, как обычно, предшествовал имени функции, c и i были бы в нем недоступны, поскольку в этот момент они еще не были объявлены. При таком объявлении authAndAccess возвращает тот тип, который возвращает operator[] при применении к переданному контейнеру.

C++14 расширяет эту возможность и позволяет опустить завершающий возвращаемый тип, оставляя только одно ведущее ключевое слово auto. При таком объявлении auto означает, что имеет место вывод типа. В частности, это означает, что компиляторы будут выводить возвращаемый тип функции из ее реализации.

template<typename Container, typename Index>

auto authAndAccess(Container &c, intx i)

{

authenticateUser();

return c[i];

}

Для большинства контейнеров с объектами типа T оператор operator[] возвращает T&, но как говорилось ранее, в процессе вывода типа шаблона «ссылочность» инициализирующего выражения игнорируется.

NB: инициализирующим выражением (expr) является c[i], который представляет собой ссылку. Мы помним, что при выводе типа строится концептуальный шаблон и при подстановке c[i] в этот шаблон его ссылочная часть будет отброшена.

В этом случае код вида

std::deque<int> d;

// …

authAndAccess(d, 5) = 10; // не скомпилируется, т.к. слева не lvalue-ссылка!

Чтобы заставить authAndAccess работать так, как мы хотим, нам надо использовать для ее возвращаемого типа вывод типа decltype, т.е. указать, что authAndAccess должна возвращать в точности тот же тип, что и выражение c[i]. Специально для таких случаев авторы C++ сделали это возможным в C++14 с помощью спецификатора decltype(auto). auto указывает, что тип должен быть выведен, а decltype говорит о том, что в процессе вывода типа следует использовать правила decltype (NB: который «попугайничает» и возвращает именно тот тип, который мы предсказываем).

NB: для дальнейшего изучения, следует изучить статью на хабре: [Секреты auto и decltype](https://habr.com/post/206458/).