7 Новые горизонты

Новый стандарт C++ предлагает много нововведений. Часть из них относится к тому, чего все давно ждали и успели освоить, что перекочевало из библиотеки boost или из technical reports к C++98. Но есть и такие, при работе с которыми требуется оперировать совершенно новыми понятиями и заставлять себя мыслить иначе, мыслить в духе действительно новых возможностей. Весь написанный ниже код пробовался на GCC 4.9.2 и GCC 5.1 с опциями –std=c++11 и –std=c++14 (обычно код валидный в одном стандарте и невалидный в другом помечен в тексте явно).

7.1 Type inference

Вывод типов это базовое и очень впечатляющее усовершенствование, которое легко понять и легко использовать. Кроме того, некоторый навык работы с системой типов, в особенности с новыми type traits несомненно нужен для понимания нового стандарта. Поэтому с него имеет смысл начать.

7.1.1 Auto и Decltype

При работе со стандартной библиотекой C++, каждому программисту знакомы невероятные обозначения типов, как в левой части следующего выражения:

```
1 map<int,list<string>>::iterator i = m.begin();
2 map<int,list<string>>::iterator j = i++;
```

Самое обидное, что компилятор и так знает всё о статическом типе правой части (по крайней мере, грубая ошибка в левой части вызовет ошибку компиляции), так что необходимость писать столько букв выглядит напрасной. Новый стандарт позволяет мягко хлопнуть компилятор по плечу и сказать ему, чтобы он попробовал догадаться о типе сам:

```
1  map <int,list<string>> m {{10, {"Hello", "World"}}, {20, {"
       Eric", "BloodAxe"}}};
2  auto i = m.begin ();
3  cout << i->first << ": " << i->second.front() << std::endl;
4  auto j = i++;
5  cout << j->first << ": " << j->second.back() << std::endl;
```

О новых способах инициализации говорить пока рано, но, кажется, в этом примере инициализация достаточно очевидна.

Вопрос к студентам: Что будет на экране?

Правильный ответ: внезапно на экране будет Hello World.

Если компилятор может взять тип выражения, логично, что это должен уметь и пользователь. Предыдущий кусок кода может быть переписан так:

```
auto i = m.begin();
decltype(i) j = i++;
```

Здесь на второй строчке объявлено "сделать j такого же типа, как i". Иногда `decltype` может быть сопряжен с некоторыми концептуальными проблемами:

```
struct Point { int x, y; };
Point porig {1, 2};
/* ... */
const Point &p = porig;
decltype(p.x) x; /* int& or const int&? */
```

Для их разрешения, стандарт вводит (5.1.2.18) форму `decltype` с двумя круглыми скобками:

```
decltype(p.x) x; /* int */
decltype((p.x)) x; /* const int& */
```

Кстати, вторая строчка является ошибкой компиляции, так как объявляет неинициализированную ссылку.

На самом деле это различие более тонкое чем просто ещё одни скобки (это была бы слишком грешная магия даже для C++). Речь идёт о разнице между `decltype(id-expr)` и `decltype(expr)`. Первое возвращает тип, с которым было объявлено имя, второе – тип, который могло бы приобрести выражение при его вычислении. Как сделать имя выражением? Обернуть его скобками, очевидно. Это кажется некоторым переусложнением, но это на самом деле единственный разумный выход из ситуации. Увы, `decltype(expr)` имеет неприятную особенность – если тип под ним это lvalue, то он добавит ссылку (если её не было)

```
int x;
typedef decltype(x) xval; /* int */
typedef decltype((x)) xref; /* int& */
```

```
typedef decltype((x+1)) xval_again; /* int */

xval r0;
xref r1 = x;
xval_again r2;
```

Почему это делается станет ясно позже при рассмотрении новых типов ссылочных значений. Но интуитивно это уже ясно – подчеркивается возможность присваивания. Между прочим выражение "могло бы быть" зафиксировано в стандарте и это даёт забавные возможности. Скажем `decltype(10000000)` это "целый тип в котором могло бы поместится 10000000". Компилятор не обязан выводить наименьший целый тип, но обычно это происходит (в GCC это происходит всегда).

Вопрос к студентам: как насчет `decltype((x+y))` если оба `int`?

Правильный ответ: конечно же сумма это не lvalue.

Вывод типов в C++ работает гораздо более приблизительно, чем их прямая аннотация, поэтому `auto` часто выводит первый попавшийся (на самом деле – наименее общий о чем см. далее) тип, а вот `decltype` всегда старается сохранить даже cv-квалификацию

```
  const int s = 5;
  auto s1 = s;
  decltype (s) s2 = 3;
  s1 += 1;
  /* s2 += 1; */
```

Строчка `auto s1 = 3` ничего не изменит.

Разумное применение этих средств позволяет существенно улучшить читаемость вашего кода и сделать гораздо меньше тонких ошибок и опечаток в сложных именах типов. Но тут нужно учитывать, что всегда есть тонкости и волчьи ямы. Я рассказал как ведет себя `auto`, но `auto` ведет себя так не всегда. Если вы конкретизируете тип, то все квалификаторы сохранятся вместе с вашей конкретизацией:

```
const int c = 0;
auto& rc = c;
rc = 44; // error: const qualifier was not removed
```

Также, и это бывает неприятно, `auto` склонно пропускать модификаторы если на верхнем уровне оно встречает то, к чему они относятся:

```
int x = 42;
```

```
const int* p1 = &x;
auto p2 = p1; /* p2 is const int* */
p2 = &y; /* ok */
/* *p2 = 3; */
```

Язык C++ справедливо считает, что константность данных под указателем это слишком важная характеристика типа. В отличии от этого, константность самого указателя может быть сброшена легко:

```
const int* const p1 = &x;
auto p2 = p1; /* p2 is const int* */
p2 = &y; /* ok */
```

Разные правила для `decltype` и `auto` привели в стандарте C++14 к введению идиомы `decltype(auto)` которая позволяет вывести тип из правой части автоматически, но именно так, как его вывел бы `decltype`.

```
const int i;
auto j = i; /* typeof(j) == int */
decltype (auto) k = i; /* typeof (k) == const int */
```

Очень интересный случай это выражение справа от `decltype`, заключенное в круглые скобки.

```
int i;
auto x4a = (i); /* decltype(x4a) is int */
decltype(auto) x4d = (i); /* decltype(x4d) is int& */
```

Таким образом моделируется поведение двух круглых скобок у `decltype (expr)` с конкретным `t`.

В качестве вишенки на торте, `auto` на самом деле достаточно мощно, чтобы замещать тип во всех контекстах (пример из стандарта C++14 раздел 5.3.4.2)

```
auto x = new auto('a');
```

Здесь выведенным значение будет `char`. Интересно дальше:

```
auto x1 = { 'a', 'b' };
```

Хитрый вопрос к студентам: Что будет выведено здесь?

Ответ “массив” неверный. В реальности будет выведен `std::initializer_list <char>` – тот самый новый тип, который позволил так лихо инициализировать вектор в самом начале.

Интересно, что это – единственное место, где отличается вывод типов шаблонами и `auto`. Для шаблонного параметра здесь будет ошибка.

Менее хитрый вопрос к студентам: мы знаем, что в случае обычных типов мы можем перечислять их через запятую в одном определении. Как вы думаете, какой тип будет выведен здесь?

```
auto x = 5, *y = &x;
```

Правильный ответ: увы, это ошибка. Это, правда, создает некоторые проблемы. Неясно почему вот так можно:

```
auto x = 1;
auto y = 1.0;
```

А вот так нельзя:

```
auto x = 1, y = 1.0; /* ! */
```

Написано-то как бы одно и то же. Ждем исправления этого в C++17 или более поздних. С другой стороны, жить это не мешает.

7.1.2 Расширенный синтаксис функций

Возможности автоматического вывода типов подразумевают построение абстракций с зависимыми типами. Пусть необходим статический шаблонный контракт на любой тип `T`, поддерживающий функцию `T::makeobject`, возвращающую некий свой тип. Нельзя (в стандарте C++11) просто взять и написать:

```
template <typename T> auto /* Error! */
makeAndProcessObject (const T& builder)
{
    auto val = builder.makeObject();
    /* do stuff with val */
    return val;
}
```

Потому что компилятор в точке объявления функции не обладает информацией о типе, который вернет `T::makeobject`. Забегая вперед – иногда обладает, скажем в C++14 тут все хорошо. Точно так же не сработает вот такой выход:

```
template <typename T>
decltype(builder.makeObject()) /* Error again! */
```

```
3 makeAndProcessObject (const T& builder)
```

Потому что `builder` не может быть использован до точки своего объявления (которой является список аргументов функции). Конечно правильного прошаренного пацана это не остановит. Он использует тот факт, что значение под `decltype` не вычисляется, и сделает тонко:

```
1 template <typename T>
2 decltype (((T*)0)->makeObject()) /* painfull but works */
3 makeAndProcessObject (const T& builder)
```

Но нельзя требовать от программистов такое всерьез, всегда и везде. Комитет по стандартизации решил эту проблему изящно, предложив расширенный синтаксис для обобщённых функций, возвращающих зависимые типы:

```
1 template <typename T> auto
2 makeAndProcessObject (const T& builder) -> decltype (builder.
    makeObject())
3 {
4     auto val = builder.makeObject();
5     /* do stuff with val */
6     return val;
7 }
```

Внутри скобок `decltype` в данном случае вычисление выражения (в том числе вызов функции) не происходит – происходит только вывод типа.

Конечно, здесь есть возможные ошибки и засады:

```
1 template <typename T, typename S>
2 auto min(T x, S y) -> decltype(x < y ? x : y)
3 {
4   return x < y ? x : y;
5 }
```

Казалось бы все при всем, но так писать нельзя, поскольку `decltype` вокруг выражения работает как `decltype(expr)`, а значит результат может быть выведен как ссылка, что чревато. Так как же все таки правильно написать type-generic minimum? Эта проблема найдет свое решение позже (см. 7.2.3).

Вопрос к студентам: Представим, что `T` и `S` это простые типы без квалификаций, указателей ссылок и прочего добра. Просто `int`, `double`

и всё такое. В каких случаях `decltype` здесь выведет ссылку, а в каких нет?

Правильный ответ: ссылка будет если типы одинаковые. Если они разные, все будет хорошо.

Зато теперь можно понять почему же такое поведение `decltype(expr)` было выбрано комитетом по стандартизации. Рассмотрим упрощенную задачу – допустим речь идет о выводе типа для доступа к элементу массива:

```
template <typename T>
auto array_access(T& array, size_t pos) -> decltype(array[pos])
{
  return array[pos];
}
```

С текущим подходом можно использовать такую обертку прозрачно как если бы это действительно был доступ к элементу массива:

```
std::vector<int> vect = {42, 43, 44};
int* p = &vect[0];

array_access(vect, 2) = 45;
array_access(p, 2) = 46;
```

В противном случае пришлось бы идти на разнообразные хаки.

Позвольте еще маленькую вишенку на этот тортик:

```
auto main() -> int
```

Теперь является легитимной формой функции `main` и если вы хотите показать насколько вы круче остальных людей в этом мире, то вы теперь знаете, что делать.

Впрочем, это так, отвлечение. Последний стандарт привнес ещё больше радости.

7.1.3 Коррективы в вывод типов для C++14

В некоторых простых случаях компилятору действительно не составляет проблем вывести тип функции:

```
auto isquare (int x) -> decltype (x)
{
```

```
  return x*x;
}
```

Здесь указание `decltype` выглядит просто излишним и C++14 разрешает его убрать:

```
auto isquare (int x) { return x*x; }
```

Для таких простых вариантов все хорошо, но как быть с рекурсией? Здесь возникает проблема: тип должен быть выведен до того, как рекурсивный вызов произошел:

```
auto sum_to (int i)
{
  if (i < 2)
    return i; // return type deduced as int
  else
    return sum_to (i-1) + i; // ok to call it now
}

cout << sum_to (10) << endl;
```

Но если переставить возвраты в вышеприведенном коде, он не будет скомпилирован.

```
auto bad_sum_to (int i)
{
  if (i > 2)
    return bad_sum_to (i-1) + i;
  else
    return i;
}
```

Впрочем GCC 4.9.2 возвращает вполне человечное описание ошибки:

```
error: use of ‘auto bad_sum_to(int)’ before deduction of ‘auto’
     return bad_sum_to (i-1) + i;
```

Увы, как уже было сказано `auto` режет типы.

```
auto Example(int const& i)
{
    return i;
}
```

Здесь возвращаемый тип int. Конечно, в конкретном коде несложно вернуть квалификацию типа:

```
auto const& Example(int const& i)
{
    return i;
}
```

Но что делать в обобщенном коде?

В обобщенном коде для точного вывода возвращаемого типа может быть использован decltype (auto) например так:

```
template <typename Fun, typename Arg>
decltype(auto) example(Fun fun, Arg arg)
{
    return fun(arg);
}
```

Теперь тип возвращаемого значения будет проброшен точно. Ниже будет показано как написать совершенно прозрачную обертку: пробросив не только возвращаемое значение но и произвольные аргументы (см. 7.2.6).

В новом стандарте также нет проблем с тем, чтобы шаблонные методы выводили свой тип непосредственно из других методов того же класса без явного this (пример взят из стандарта C++14, 5.1.1.3)

```
struct A
{
  char g();
  template <class T> auto f(T t) -> decltype(t + g())
  { return t + g(); }
};
```

7.1.4 Decaying и минимальные общие типы

Decaying уже был рассмотрен, при рассмотрении деградации массивов в указатели (3), но если в старом стандарте это была built-in особенность для одной конкретной пары типов, то новый стандарт предлагает интересные варианты обобщения этого понятия на любые типы. Простой случай:

```
int foo (const int &s)
```

```
{
  return s + 2;
}
```

Здесь в выражении `s + 2`, `s` ведёт себя так, как будто его тип `int`. Тогда можно сказать, что `const int &` деградирует к `int` в том же смысле, в каком массив деградирует к указателю, etc. Новый стандарт позволяет вручную "деградировать" тип:

```
const int &i;
std::decay<decltype(i)>::type j; /* int j */
/* and btw */
auto k = i; /* int k = i; */
```

Поскольку `auto` также осуществляет деградацию, можно считать `decay + decltype` способом вывести тот тип, который вывело бы `auto`.

На механизме `decay` неявно построен механизм `common_type`, позволяющий вывести минимальный общий тип:

```
template <typename T, typename S>
void foo(T lhs, S rhs) {
{
  std::common_type<decltype(lhs), decltype(rhs)>::type k;
  // ...
}
```

В принципе минимальные общие типы не так уже и нужны

```
template <typename T, typename S>
void foo(T lhs, S rhs) {
  auto prod = lhs * rhs;
  //...
}
```

Устроит не менее качественную деградацию. Мало того, такое расширение GCC давно известное и во многих других компиляторах как `typeof`, позволяло делать это и в старом стандарте. В новом же можно использовать `decltype`.

```
typedef typeof(lhs * rhs) product_type;
typedef decltype(lhs * rhs) product_type;
```

В качестве полезного примера, можно привести смешанную арифметику для числового класса:

```
template <typename T>
struct Number { T n; };

template <typename T, class U>
Number<typename std::common_type<T, U>::type>
operator+(const Number<T>& lhs,
          const Number<U>& rhs)
{
    return {lhs.n + rhs.n};
}

int main()
{
    Number<int> i1 = {1}, i2 = {2};
    Number<double> d1 = {2.3}, d2 = {3.5};
    std::cout << "i1i2: " << (i1 + i2).n
              << "\ni1d2: " << (i1 + d2).n
              << "\nd1i2: " << (d1 + i2).n
              << "\nd1d2: " << (d1 + d2).n
              << '\n';
}
```

Домашняя наработка: проанализировать вывод этой программы и то, почему он ведет себя именно так.

7.1.5 Стой, кто идёт?

Новый стандарт предлагает большое количество удобных стандартных шаблонов для получения более детальной информации о типах на этапе выполнения.

Большинство из них должны как-то сообщать ответы “да” и “нет” на вопросы “является ли это тем-то и тем-то?”, скажем: “является ли тип аргумента указателем на функцию-член класса X с такой-то сигнатурой?”. Чтобы закодировать ответы, используется обертка над интегральными константами времени компиляции:

```
template <typename T, T v>
struct integral_constant;
```

Теперь можно определить `true_type` как `integral_constant<bool, true>` и `false_type` как `integral_constant<bool, false>`.

Можно продемонстрировать создание пользовательских констант:

```
typedef integral_constant<int, 2> two_t;
typedef integral_constant<int, 4> four_t;
static_assert (two_t::value * two_t::value == four_t::value, "
    2*2 != 4");
```

Используя закодированные таким образом истину и ложь, можно сварганить простейший из определителей типов: является ли анализируемый тип интегральным (это такие типы как `bool`, `char`, `short`, `int`, `long`, `long long` и все их cv-квалификации). Как пример использования: можно потребовать интегрального `T` в шаблонной функции (пока нет концептов, вполне себе концепт для бедных).

```
template <typename T>
T f(T i)
{
    static_assert(std::is_integral<T>::value,
                  "Integer required");
    return i;
}
```

Домашняя наработка: как могла бы выглядеть реализация `is_integral` и `is_floating_point`?

Имея два определителя, можно скомбинировать из них производные, скажем:

```
template <typename T>
struct is_arithmetic : std::integral_constant<bool,
  std::is_integral<T>::value ||
  std::is_floating_point<T>::value> {};
```

Можно потренироваться и определить является ли нечто указателем:

```
template <typename T>
struct is_pointer_helper : std::false_type {};
template <typename T> struct is_pointer_helper<T*> : std::
    true_type {};
template <typename T> struct is_pointer :
    is_pointer_helper <typename std::remove_cv<T>::type> {};
```

Обратите внимание на использование `remove_cv`, который сам является композитным хелпером из `remove_const` и `remove_volatile` которые по отдельности тоже не составляют проблем.

Вопрос к студентам: как бы вы реализовали `remove_const`?

Правильный ответ:

```
template <typename T>
struct remove_const { typedef T type; };
template <typename T>
struct remove_const<const T> { typedef T type; };
```

Полное рассмотрение всех возможных traits не нужно – они перечислены в стандарте и их несложно конструировать по мере необходимости. Можно запомнить (это примерно столь же полезная для запоминания информация как первый 21 знак числа пи), что все что угодно, что встречается в корректной программе на C++14 может быть отнесено к одному из 14 базовых классов traits и только к нему одному. И кстати 14 это первые две цифры номера стандарта. Совпадение? Не думаю.

7.1.6 Ваш третий лучший друг

До сих пор двумя лучшими (воображаемыми) друзьями программиста были `typedef` (2.2.2) и `typename` (4.6). Но теперь оказывается, что в новом стандарте у `typedef` есть младший братик `using`

```
typedef int MyInt;
using MyInt = int;
```

Эти две строчки совершенно эквивалентны. Зачем же нужно было вводить новое ключевое слово? Потому что `using` умеет больше:

```
template <typename T> using MyType = AnotherType< T,
    MyAllocatorType >;
template <typename T> using ptr = T*;
...
MyType<int> a;
ptr<int> x;
```

Очень удобно совмещать новый `using` с определителями типов. Например такое переопределение, как приведенное ниже:

```
template <typename T> using decay_t = typename decay<T>::type;
```

Позволяет сделать in-place сгнивалку для типов. Синонимы для всех распространенных определителей уже включены в стандарт.