Содержание

ГЛАВА 2

Объявление auto

Концептуально объявление auto настолько простое, насколько может быть, но все же сложнее, чем выглядит. Его применение экономит исходный текст, вводимый программистом, но при этом предупреждает появление вопросов корректности и производительности, над которыми вынужден мучиться программист при ручном объявлении типов. Кроме того, некоторые выводы типов auto, хотя и послушно соблюдают предписанные алгоритмы, дают результаты, некорректные с точки зрения программиста. Когда такое происходит, важно знать, как привести auto к верному ответу, поскольку возврат к указанию типов вручную — альтернатива, которой чаще всего лучше избегать.

В этой короткой главе описаны основы работы с auto.

2.1. Предпочитайте auto явному объявлению типа

Легко и радостно написать

```
int x;
```

Стоп! #@$! Я забыл инициализировать x, так что эта переменная имеет неопределенное значение. Может быть. Но она может быть инициализирована и нулем — в зависимости от контекста. Жуть!

Ну, ладно. Давайте лучше порадуемся объявлению локальной переменной, инициализированной разыменованием итератора:

```
template<typename It> // Некий алгоритм, работающий с
void dwim(It b, It e) // элементами из диапазона от b до e
{
    while (b != e) {
        typename std::iterator_traits<It>::value_type
            currValue = *b;
        …
    }
}
```

Жуть. typename std::iterator_traits<It>::value_type — просто чтобы записать тип значения, на которое указывает итератор? Нет, я такой радости не переживу… #@$! Или я это уже говорил?..

Ладно, третья попытка. Попробую объявить локальную переменную, тип которой такой же, как у лямбда-выражения. Но его тип известен только компилятору. #@$! (Это становится привычкой…)

Да что же это такое — никакого удовольствия от программирования на C++!

Так не должно быть. И не будет! Мы дождались C++11, в котором все эти проблемы решены с помощью ключевого слова `auto`. Тип переменных, объявленных как `auto`, выводится из их инициализатора, так что они обязаны быть инициализированными. Это значит — прощай проблема неинициализированных переменных:

```
int x1;       // Потенциально неинициализированная переменная
auto x2;      // Ошибка! Требуется инициализатор
auto x3 = 0; // Все отлично, переменная x корректно определена
```

Нет проблем и с объявлением локальной переменной, значением которой является разыменование итератора:

```
template<typename It>  // Все, как и ранее
void dwim(It b, It e)
{
    while (b != e) {
        auto currValue = *b;
        …
    }
}
```

А поскольку `auto` использует вывод типов (см. раздел 1.2), он может представлять типы, известные только компиляторам:

```
auto derefUPLess =                      // Функция сравнения
[](const std::unique_ptr<Widget>& p1, // объектов Widget, на
   const std::unique_ptr<Widget>& p2) // которые указывают
   { return *p1 < *p2; };              // std::unique_ptr
```

Просто круто! В C++14 все еще круче, потому что параметры лямбда-выражений также могут включать `auto`:

```
auto derefLess =          // Функция сравнения в C++14,
[](const auto& p1,        // для значений, на которые
   const auto& p2)        // указывает что угодно
   { return *p1 < *p2; }; // указателеобразное
```

Несмотря на всю крутость вы, вероятно, думаете, что можно обойтись и без `auto` для объявления переменной, которая хранит лямбда-выражение, поскольку мы можем использовать объект `std::function`. Это так, можем, но, возможно, это не то, что вы на самом деле подразумеваете. А может быть, вы сейчас думаете "А что это такое — объект `std::function`?" Давайте разбираться.

`std::function` — шаблон стандартной библиотеки C++11, который обобщает идею указателя на функцию. В то время как указатели на функции могут указывать только

на функции, объект std::function может ссылаться на любой вызываемый объект, т.е. на все, что может быть вызвано как функция. Так же как при создании указателя на функцию вы должны указать тип функции, на которую указываете (т.е. сигнатуру функции, на которую хотите указать), вы должны указать тип функции, на которую будет ссылаться создаваемый объект std::function. Это делается с помощью параметра шаблона std::function. Например, для объявления объекта std::function с именем func, который может ссылаться на любой вызываемый объект, действующий так, как если бы его сигнатура была

```
bool(const std::unique_ptr<Widget>&, // Сигнатура C++11 для
     const std::unique_ptr<Widget>&) // функции сравнения
                                     // std::unique_ptr<Widget>
```

следует написать следующее:

```
std::function<bool(const std::unique_ptr<Widget>&,
                   const std::unique_ptr<Widget>&)> func;
```

Поскольку лямбда-выражения дают вызываемые объекты, замыкания могут храниться в объектах std::function. Это означает, что можно объявить C++11-версию derefUPLess без применения auto следующим образом:

```
std::function<bool(const std::unique_ptr<Widget>&,
                   const std::unique_ptr<Widget>&)>
    derefUPLess = [](const std::unique_ptr<Widget>& p1,
                     const std::unique_ptr<Widget>& p2)
        { return *p1 < *p2; };
```

Важно понимать, что, даже если оставить в стороне синтаксическую многословность и необходимость повторения типов параметров, использование std::function — не то же самое, что использование auto. Переменная, объявленная с использованием auto и хранящая замыкание, имеет тот же тип, что и замыкание, и как таковая использует только то количество памяти, которое требуется замыканию. Тип переменной, объявленной как std::function и хранящей замыкание, представляет собой конкретизацию шаблона std::function, которая имеет фиксированный размер для каждой заданной сигнатуры. Этот размер может быть не адекватным для замыкания, которое требуется хранить, и в этом случае конструктор std::function будет выделять для хранения замыкания динамическую память. В результате объект std::function использует больше памяти, чем объект, объявленный с помощью auto. Кроме того, из-за деталей реализации это ограничивает возможности встраивания и приводит к косвенным вызовам функции, так что вызовы замыкания через объект std::function обычно выполняются медленнее, чем вызовы посредством объекта, объявленного как auto. Другими словами, подход с использованием std::function в общем случае более громоздкий, требующий больше памяти и более медленный, чем подход с помощью auto, и к тому же может приводить к генерации исключений, связанных с нехваткой памяти. Ну и, как вы уже видели в примерах выше, написать "auto" — гораздо проще, чем указывать тип для инстанцирования std::function. В соревновании между auto

и std::function для хранения замыкания побеждает auto. (Подобные аргументы можно привести и в пользу предпочтения auto перед std::function для хранения результатов вызовов std::bind, но все равно в разделе 6.4 я делаю все, чтобы убедить вас использовать вместо std::bind лямбда-выражения…)

Преимущества auto выходят за рамки избегания неинициализированных переменных, длинных объявлений переменных и возможности непосредственного хранения замыкания. Кроме того, имеется возможность избежать того, что я называю проблемой "сокращений типа" (type shortcuts). Вот кое-что, что вы, вероятно, уже видели, а возможно, даже писали:

```
std::vector<int> v;
…
unsigned sz = v.size();
```

Официальный возвращаемый тип v.size() — std::vector<int>::size_type, но об этом знает не так уж много разработчиков. std::vector<int>::size_type определен как беззнаковый целочисленный тип, так что огромное количество программистов считают, что unsigned вполне достаточно, и пишут исходные тексты, подобные показанному выше. Это может иметь некоторые интересные последствия. В 32-разрядной Windows, например, и unsigned, и std::vector<int>::size_type имеют один и тот же размер, но в 64-разрядной Windows unsigned содержит 32 бита, а std::vector<int>::size_type — 64 бита. Это означает, что код, который работал в 32-разрядной Windows, может вести себя некорректно в 64-разрядной Windows. И кому хочется тратить время на подобные вопросы при переносе приложения с 32-разрядной операционной системы на 64-разрядную?

Применение auto гарантирует, что вам не придется этим заниматься:

```
auto sz = v.size(); // Тип sz — std::vector<int>::size_type
```

Все еще не уверены в разумности применения auto? Тогда рассмотрите следующий код.

```
std::unordered_map<std::string, int> m;
…
for (const std::pair<std::string, int>& p : m)
{
    … // Что-то делаем с p
}
```

Выглядит вполне разумно… но есть одна проблема. Вы ее не видите?

Чтобы разобраться, что здесь не так, надо вспомнить, что часть std::unordered_map, содержащая ключ, является константной, так что тип std::pair в хеш-таблице (которой является std::unordered_map) вовсе не std::pair<std::string, int>, а std::pair<const std::string, int>. Но переменная p в приведенном выше цикле объявлена иначе. В результате компилятор будет искать способ преобразовать объекты std::pair<const std::string, int> (хранящиеся в хеш-таблице) в объекты std::pair<std::string, int> (объявленный тип p). Этот способ — создание временного объекта типа, требуемого p, чтобы скопировать в него каждый объект из m с последующим

связыванием ссылки p с этим временным объектом. В конце каждой итерации цикла временный объект уничтожается. Если этот цикл написан вами, вы, вероятно, будете удивлены его поведением, поскольку почти наверняка планировали просто связывать ссылку p с каждым элементом в m.

Такое непреднамеренное несоответствие легко лечится с помощью auto:

```
for (const auto& p : m)
{
    … // Как и ранее
}
```

Это не просто более эффективно — это еще и менее многословно. Кроме того, этот код имеет очень привлекательную особенность — если вы возьмете адрес p, то можете быть уверены, что получите указатель на элемент в m. В коде, не использующем auto, вы получите указатель на временный объект — объект, который будет уничтожен в конце итерации цикла.

Два последних примера — запись unsigned там, где вы должны были написать std::vector<int>::size_type, и запись std::pair<std::string, int> там, где вы должны были написать std::pair<const std::string, int>, — демонстрируют, как явное указание типов может привести к неявному их преобразованию, которое вы не хотели и не ждали. Если вы используете в качестве типа целевой переменной auto, вам не надо беспокоиться о несоответствиях между типом объявленной переменной и типом инициализирующего ее выражения.

Таким образом, имеется несколько причин для предпочтительного применения auto по сравнению с явным объявлением типа. Но auto не является совершенным. Тип для каждой переменной, объявленной как auto, выводится из инициализирующего ее выражения, а некоторые инициализирующие выражения имеют типы, которые не предполагались и нежелательны. Условия, при которых возникают такие ситуации, и что при этом можно сделать, рассматриваются в разделах 1.2 и 2.2, поэтому здесь я не буду их рассматривать. Вместо этого я уделю внимание другому вопросу, который может вас волновать при использовании auto вместо традиционного объявления типа, — удобочитаемость полученного исходного текста.

Для начала сделайте глубокий вдох и расслабьтесь. Применение auto — возможность, а не требование. Если, в соответствии с вашими профессиональными представлениями, ваш код будет понятнее или легче сопровождаемым или лучше в каком-то ином отношении при использовании явных объявлений типов, вы можете продолжать их использовать. Но имейте в виду, что C++ — не первый язык, принявший на вооружение то, что в мире языков программирования известно как *вывод типов* (type inference). Другие процедурные статически типизированные языки программирования (например, C#, D, Scala, Visual Basic) обладают более или менее эквивалентными возможностями, не говоря уже о множестве статически типизированных функциональных языков (например, ML, Haskell, OCaml, F# и др.). В частности, это объясняется успехом динамически типизированных языков программирования, таких как Perl, Python и Ruby, в которых явная типизация переменных — большая редкость. Сообщество разработчиков программного

обеспечения имеет обширный опыт работы с выводом типов, и он продемонстрировал, что в такой технологии нет ничего мешающего созданию и поддержке крупных приложений промышленного уровня.

Некоторых разработчиков беспокоит тот факт, что применение `auto` исключает возможность определения типа при беглом взгляде на исходный текст. Однако возможности IDE показывать типы объектов часто устраняют эту проблему (даже если принять во внимание обсуждавшиеся в разделе 1.4 вопросы, связанные с выводом типов в IDE), а во многих случаях абстрактный взгляд на тип объекта столь же полезен, как и точный тип. Зачастую достаточно, например, знать, что объект является контейнером, счетчиком или интеллектуальным указателем, не зная при этом точно, каким именно контейнером, счетчиком или указателем. При правильном подборе имен переменных такая абстрактная информация о типе почти всегда оказывается под рукой.

Суть дела заключается в том, что явно указываемые типы зачастую мало что дают, кроме того что открывают возможности для ошибок — в плане как корректности, так и производительности программ. Кроме того, типы `auto` автоматически изменяются при изменении типов инициализирующих их выражений, а это означает облегчение выполнения рефакторинга при использовании `auto`. Например, если функция объявлена как возвращающая `int`, но позже вы решите, что `long` вас больше устраивает, вызывающий код автоматически обновится при следующей компиляции (если результат вызова функции хранится в переменной, объявленной как `auto`). Если результат хранится в переменной, объявленной как `int`, вы должны найти все точки вызова функции и внести необходимые изменения.

Следует запомнить

- Переменные, объявленные как `auto`, должны быть инициализированы; в общем случае они невосприимчивы к несоответствиям типов, которые могут привести к проблемам переносимости или эффективности; могут облегчить процесс рефакторинга; и обычно требуют куда меньшего количества ударов по клавишам, чем переменные с явно указанными типами.
- Переменные, объявленные как `auto`, могут быть подвержены неприятностям, описанным в разделах 1.2 и 2.2.

2.2. Если `auto` выводит нежелательный тип, используйте явно типизированный инициализатор

В разделе 2.1 поясняется, что применение `auto` для объявления переменных предоставляет ряд технических преимуществ по сравнению с явным указанием типов, но иногда вывод типа `auto` идет налево там, где вы хотите направо. Предположим, например, что у меня есть функция, которая получает `Widget` и возвращает `std::vector<bool>`, где каждый `bool` указывает, обладает ли `Widget` определенным свойством:

```
std::vector<bool> features(const Widget& w);
```

Предположим далее, что пятый бит указывает наличие высокого приоритета у `Widget`. Мы можем написать следующий код.

```
Widget w;
…
bool highPriority = features(w)[5]; // Имеет ли w высокий
                                    // приоритет?
…
processWidget(w, highPriority);     // Обработка w в соответ-
                                    // ствии с приоритетом
```

В этом коде нет ничего неверного. Он корректно работает. Но если мы внесем кажущееся безобидным изменение и заменим явный тип `highPriority` типом `auto`

```
auto highPriority = features(w)[5]; // Имеет ли w высокий
                                    // приоритет?
```

то ситуация изменится. Код будет продолжать компилироваться, но его поведение больше не будет предсказуемым:

```
processWidget(w, highPriority); // Неопределенное поведение!
```

Как указано в комментарии, вызов `processWidget` теперь имеет неопределенное поведение. Но почему? Ответ, скорее всего, вас удивит. В коде, использующем `auto`, тип `highPriority` больше не является `bool`. Хотя концептуально `std::vector<bool>` хранит значения `bool`, `operator[]` у `std::vector<bool>` не возвращает ссылку на элемент контейнера (то, что `std::vector::operator[]` возвращает для всех типов *за исключением* `bool`). Вместо этого возвращается объект типа `std::vector<bool>::reference` (класса, вложенного в `std::vector<bool>`).

Тип `std::vector<bool>::reference` существует потому, что `std::vector<bool>` определен как хранящий значения `bool` в упакованном виде, по одному биту на каждое значение. Это создает проблему для оператора `operator[]` класса `std::vector<bool>`, поскольку `operator[]` класса `std::vector<T>` должен возвращать `T&`, но C++ запрещает ссылаться на отдельные биты. Будучи не в состоянии вернуть `bool&`, `operator[]` класса `std::vector<bool>` возвращает объект, который *действует подобно* `bool&`. Для успешной работы объекты `std::vector<bool>::reference` должны быть применимы по сути во всех контекстах, где применим `bool&`. Среди прочих возможностей `std::vector<bool>::reference` обладает неявным преобразованием в `bool`. (Не в `bool&`, а именно в `bool`. Пояснение всего набора методов, используемых `std::vector<bool>::reference` для эмуляции поведения `bool&`, завело бы нас слишком далеко, так что я просто замечу, что это неявное преобразование является только одним из камней в существенно большей мозаике.)

С учетом этой информации посмотрим еще раз на следующую часть исходного кода:

```
bool highPriority = features(w)[5]; // Явное объявление типа
                                    // highPriority
```

Здесь `features` возвращает объект `std::vector<bool>`, для которого вызывается `operator[]`. Этот оператор возвращает объект типа `std::vector<bool>::reference`, который затем неявно преобразуется в значение типа `bool`, необходимое для инициализации `highPriority`. Таким образом, `highPriority` в конечном итоге получает значение пятого бита из `std::vector<bool>`, возвращенного функцией `features`, так, как и предполагалось.

Но что же произойдет, если переменная `highPriority` будет объявлена как `auto`?

```
auto highPriority = features(w)[5]; // Вывод типа highPriority
```

Функция `features`, как и ранее, возвращает объект типа `std::vector<bool>`, и, как и ранее, выполняется его `operator[]`. Оператор возвращает объект типа `std::vector<bool>::reference`, но дальше привычный ход событий изменяется, так как `auto` приводит к выводу типа переменной `highPriority`. Теперь переменная `highPriority` не получает значение пятого бита `std::vector<bool>`, возвращенного вызовом `features`.

Полученное ею значение зависит от того, как реализован тип `std::vector<bool>::reference`. Одна из реализаций таких объектов состоит в том, чтобы содержать указатель на машинное слово с интересующим нас битом и смещение этого бита в слове. Рассмотрим, что это означает для инициализации `highPriority`, в предположении, что имеет место именно такая реализация `std::vector<bool>::reference`.

Вызов `features` возвращает временный объект `std::vector<bool>`. Этот объект не имеет имени, но для упрощения нашего рассмотрения я буду называть его `temp`. Для `temp` вызывается `operator[]`, в результате чего возвращается объект `std::vector<bool>::reference`, содержащий указатель на слово в структуре данных, хранящей интересующий нас бит (эта структура находится под управлением `temp`), плюс смещение в слове, соответствующее пятому биту. Переменная `highPriority` представляет собой копию этого объекта `std::vector<bool>::reference`, так что `highPriority` тоже содержит указатель на слово в `temp` плюс смещение, соответствующее пятому биту. В конце инструкции объект `temp` уничтожается, так как это объект временный. В результате переменная `highPriority` содержит висячий указатель, что и дает неопределенное поведение при вызове `processWidget`:

```
processWidget(w, highPriority); // Неопределенное поведение!
                                // highPriority содержит
                                // висячий указатель!
```

Класс `std::vector<bool>::reference` является примером *прокси-класса* (proxy class), т.е. класса, цель которого — эмуляция и дополнение поведения некоторого другого типа. Прокси-классы применяются для множества разных целей. Например, `std::vector<bool>::reference` нужен для того, чтобы создать иллюзию, что `operator[]` класса `std::vector<bool>` возвращает ссылку на бит, а интеллектуальные указатели стандартной библиотеки (см. главу 4, “Интеллектуальные указатели”) являются прокси-классами, которые добавляют к обычным указателям управление ресурсами. Полезность прокси-классов — давно установленный и не вызывающий сомнения факт. Фактически

шаблон проектирования "Прокси" — один из наиболее давних членов пантеона шаблонов проектирования программного обеспечения.

Одни прокси-классы спроектированы так, чтобы быть очевидными для клиентов. Это, например, такие классы, как `std::shared_ptr` и `std::unique_ptr`. Другие прокси-классы спроектированы для более-менее невидимой работы. Примером такого "невидимого" прокси-класса является `std::vector<bool>::reference`, как и его собрат `std::bitset::reference` из класса `std::bitset`.

В этом же лагере находятся и некоторые классы библиотек C++, применяющих технологию, известную как *шаблоны выражений* (expression templates). Такие библиотеки изначально разрабатывались для повышения эффективности кода для числовых вычислений. Например, для заданного класса `Matrix` и объектов `m1`, `m2`, `m3` и `m4` класса `Matrix`, выражение

```
Matrix sum = m1 + m2 + m3 + m4;
```

может быть вычислено более эффективно, если `operator+` для объектов `Matrix` возвращает не сам результат, а его прокси-класс. Иначе говоря, `operator+` для двух объектов `Matrix` должен возвращать объект прокси-класса, такого как `Sum<Matrix,Matrix>`, а не объект `Matrix`. Как и в случае с `std::vector<bool>::reference` и `bool`, должно иметься неявное преобразование из прокси-класса в `Matrix`, которое позволит инициализировать `sum` прокси-объектом, полученным из выражения справа от знака "=". (Тип этого объекта будет традиционно кодировать все выражение инициализации, т.е. быть чем-то наподобие `Sum<Sum<Sum<Matrix, Matrix>, Matrix>, Matrix>`. Определенно, это тип, от которого следует защитить клиентов.)

В качестве общего правила "невидимые" прокси-классы не умеют хорошо работать вместе с `auto`. Для объектов таких классов зачастую не предусматривается существование более длительное, чем одна инструкция, так что создание переменных таких типов, как правило, нарушает фундаментальные предположения проекта библиотеки. Это справедливо для `std::vector<bool>::reference`, и мы видели, как нарушение предположений ведет к неопределенному поведению.

Следовательно, надо избегать кода следующего вида:

```
auto someVar = выражение с типом "невидимого" прокси-класса;
```

Но как распознать, когда используется прокси-объект? Программное обеспечение, использующее невидимый прокси, вряд ли станет его рекламировать. Ведь эти прокси-объекты должны быть *невидимыми*, по крайней мере концептуально! И если вы обнаружите их, то действительно ли следует отказываться от `auto` и массы преимуществ, продемонстрированных для него в разделе 2.1?

Давайте сначала зададимся вопросом, как найти прокси. Хотя "невидимые" прокси-классы спроектированы таким образом, чтобы при повседневном применении "летать вне досягаемости радара программиста", использующие их библиотеки часто документируют такое применение. Чем лучше вы знакомы с основными проектными решениями

используемых вами библиотек, тем менее вероятно, что вы пропустите такой прокси незамеченным.

Там, где документация слишком краткая, на помощь могут прийти заголовочные файлы. Возможность сокрытия прокси-объектов в исходном коде достаточно редка. Обычно прокси-объекты возвращаются из функций, которые вызываются клиентами, так что сигнатуры этих функций отражают существование прокси-объектов. Например, вот как выглядит `std::vector<bool>::operator[]`:

```
namespace std {    // Из стандарта C++
    template <class Allocator>
    class vector<bool, Allocator> {
    public:
        …
        class reference { … };

        reference operator[](size_type n);
        …
    };
}
```

В предположении, что вы знаете, что `operator[]` у `std::vector<T>` обычно возвращает `T&`, необычный возвращаемый тип у `operator[]` в данном случае должен навести вас на мысль о применении здесь прокси-класса. Уделяя повышенное внимание используемым интерфейсам, часто можно выявить наличие прокси-классов.

На практике многие разработчики обнаруживают применение прокси-классов только тогда, когда пытаются отследить источник таинственных проблем при компиляции или отладить никак не проходящий тесты модуль. Независимо от того, как вы его обнаружили, после того как выясняется, что `auto` определен как выведенный тип прокси-класса вместо "проксифицируемого" типа, решение не требует отказа от `auto`. Само по себе ключевое слово `auto` проблемой не является. Проблема в том, что `auto` выводит не тот тип, который вам нужен. Решение заключается в том, чтобы обеспечить вывод другого типа. Способ достижения этого заключается в том, что я называю *идиомой явной типизации инициализатора*.

Идиома явной типизации инициализатора включает объявление переменной с использованием `auto`, но с приведением инициализирующего выражения к тому типу, который должен вывести `auto`. Например, вот как можно использовать эту идиому, чтобы заставить `highPriority` стать переменной типа `bool`:

```
auto highPriority = static_cast<bool>(features(w)[5]);
```

Здесь `features(w)[5]` продолжает, как и ранее, возвращать объект типа `std::vector<bool>::reference`, но приведение изменяет тип выражения на `bool`, который `auto` затем выводит в качестве типа переменной `highPriority`. Во время выполнения программы объект `std::vector<bool>::reference`, который возвращается вызовом `std::vector<bool>::operator[]`, преобразуется в значение `bool` и в качестве

части преобразования выполняется разыменование все еще корректного указателя на `std::vector<bool>`, возвращенного вызовом `features`. Это позволяет избежать неопределенного поведения, с которым мы сталкивались ранее. Затем к битам, на которые указывает указатель, применяется индексация с индексом 5 и полученное значение типа `bool` используется для инициализации переменной `highPriority`.

В примере с `Matrix` идиома явно типизированного инициализатора выглядит следующим образом:

```
auto sum = static_cast<Matrix>(m1 + m2 + m3 + m4);
```

Применение идиомы не ограничивается инициализаторами, производимыми проксиклассами. Она может быть полезной для того, чтобы подчеркнуть, что вы сознательно создаете переменную типа, отличного от типа, генерируемого инициализирующим выражением. Предположим, например, что у вас есть функция для вычисления некоторого значения отклонения:

```
double calcEpsilon(); // Возвращает значение отклонения
```

Очевидно, что `calcEpsilon` возвращает значение `double`, но предположим, что вы знаете, что для вашего приложения точности `float` вполне достаточно и для вас существенна разница в размерах между `float` и `double`. Вы можете объявить переменную типа `float` для хранения результата функции `calcEpsilon`

```
float ep = calcEpsilon(); // Неявное преобразование
                          // double -> float
```

но это вряд ли выражает мысль “я намеренно уменьшаю точность значения, возвращенного функцией”. Зато это делает идиома явной типизации инициализатора:

```
auto ep = static_cast<float>(calcEpsilon());
```

Аналогичные рассуждения применяются, если у вас есть выражение с плавающей точкой, которое вы преднамеренно сохраняете как целочисленное значение. Предположим, что вам надо вычислить индекс элемента в контейнере с итераторами произвольного доступа (например, `std::vector`, `std::deque` или `std::array`) и вы получаете значение типа `double` между `0.0` и `1.0`, указывающее, насколько далеко от начала контейнера расположен этот элемент (`0.5` указывает на середину контейнера). Далее, предположим, что вы уверены в том, что полученный индекс можно разместить в `int`. Если ваш контейнер — `c`, а значение с плавающей точкой — `d`, индекс можно вычислить следующим образом:

```
int index = d * (c.size() - 1);
```

Но здесь скрыт тот факт, что вы преднамеренно преобразуете `double` справа от знака “=” в `int`. Идиома явно типизированного инициализатора делает этот факт очевидным:

```
auto index = static_cast<int>(d * (c.size() - 1));
```

Следует запомнить

- “Невидимые” прокси-типы могут привести `auto` к выводу неверного типа инициализирующего выражения.
- Идиома явно типизированного инициализатора заставляет `auto` выводить тот тип, который нужен вам.