Контрольные вопросы

24 ноября 2022 г.

1 Что обеспечивает идеальная передача, и как она реализуется?

Идеальная (прямая передача, perfect forwarding) передача обеспечивает обобщённый код, который передает фундаментальные свойства передаваемых аргументов:

- Модифицируемый объект должен оставаться модифицируемым.
- Константный объект должен быть передан как объект, предназначенный лишь для чтения.
- Перемещаемый объект должен передаваться как перемещаемый объект.

Оказывается, что достижение такой функциональности без шаблонов сталкивается с трудностями при работе с перемещаемыми объектами. Код для таких объектов отличается от реализации кода с lvalue: он должен использовать std::move(), т.к. согласно правилам языка семантика перемещения не передаётся. При объединении же всех трёх случаев в обобщённом коде (с шаблонной функцией) он не будет работать, когда передается перемещаемый объект.

Для достижения же требуемой функциональности C++11 вводит специальные правила для идеальной передачи параметров. Идиоматическая схема кода имеет следующий вид:

```
template<typename T>
void f (T&& val)
{
        g(std::forward<T>(val)); // perfect forwarding val to g()
}
```

При этом важно, что std::move() не имеет шаблонного параметра и "запускает"семантику перемещения для передаваемого аргумента, в то время как std::forward() "передаёт"потенциальную семантику перемещения в зависимости от переданного аргумента шаблона.

Кроме того, T&& для параметра шаблона T ведёт себя не так, как X&& для конкретного типа X (см. вопрос 2). Хотя, например, при передаче аргумента шаблонной функции по пробрасывающей ссылке, его копия не создаётся, как и в случае передачи аргумента по любой другой ссылке.

15.6.3.

2 Какая ссылка называется пробрасывающей или универсальной?

Пробрасывающей (forwarding reference) или универсальной (universal reference) ссылкой называется T&& для параметра шаблона T. Она может быть связана с изменяемым, неизменяемым (const) или перемещаемым объектом. В определении функции параметр может быть изменяемым, неизменяемым или указывать на объект, у которого можно переместить внутреннее содержимое. В этом и состоит отличие от X&&, который для конкретного типа X объявляет параметр как rvalue-ссылку. Она может быть связана только с перемещаемым объектом (prvalue, таким как временный объект, и xvalue, таким как объект, переданный с использованием std::move()). Данная rvalue-ссылка всегда изменяема, и всегда есть возможность переместить её значение. Можно также отметить, что тип наподобие X const&& корректен, но на практике не распространен, поскольку "кража"внутреннего представления перемещаемого объекта требует его изменения. Поэтому он может использоваться для обеспечения передачи только временных значений или объектов с примененным к ним std::move() без возможности их модифицировать.

3 В чём заключается идиома SFINAE применительно к шаблонам?

Идиома SFINAE (substitution failure is not an error – ошибка подстановки ошибкой не является) обеспечивает возможность выбора между различными реализациями шаблона функции для разных типов или разных ограничений. Иными словами, она позволяет игнорировать шаблоны функции для определённых ограничений, и код шаблонов при этих ограничениях становится недопустимым.

В C++ возможна перегрузка функций. Когда компилятор видит вызов перегруженной функции, он должен рассмотреть каждого кандидата отдельно, оценивая аргументы вызова и выбирая кандидата, который наилучшим образом соответствует передаваемым аргументам.

В тех случаях, когда набор кандидатов для вызова включает шаблоны функций, компилятор сначала должен определить, какие аргументы шаблона должны использоваться для этого кандидата, затем подставить эти аргументы в список параметров функции и возвращаемый тип, а потом оценить, насколько хорошо она соответствует аргументам. В процессе подстановки могут получаться конструкции, которые не имеют никакого смысла, или конструкции, выполнения которых не предполагалось. Вместо того, чтобы считать бессмысленную подстановку ошибкой, правила языка требуют просто игнорировать кандидатов с такими проблемами. А выполнение непредполагавшихся конструкций достигается отключением шаблонов для конкретных случаев. Этот принцип/идиома и именуется SFINAE.

Описанный процесс подстановки отличается от процесса инстанцирования по требованию: подстановка может быть выполнена даже для потенциальных инстанцирований, которые на самом деле не нужны. И эта подстановка выполняется непосредственно в объявлении функции (но не в её теле).

4 Как можно использовать вспомогательный шаблон enable_if?

Варианты использования вспомогательного шаблона enable_if:

- Упаковка возвращаемого типа функции (не применимо к конструкторам/операторам):

```
template< ... >
enable_if_t < Condition , Return_Type > f( args )
```

- Добавление параметра функции с аргументом по умолчанию (Dummy необязательно):

```
template < ... >
Return_Type f(args , enable_if_t < Condition , T > Dummy = v)
```

- Добавление парамаетра шаблона с аргументом по умолчанию (по умолчанию тип void):

```
template < ... , typename Dummy = enable_if_t < Condition > >
```

- Упаковка нешаблонного аргумента:

```
template < typename T >
void f(T & t, enable_if_t < is_something < T > ::value , string & > s)
```

- Выбор типа аргумента по умолчанию (добавление параметра функции с аргументом по умолчанию)

```
template < ... >
Return_Type f(args , enable_if_t < Condition , void ** > Dummy = NULL)
```

Свойство типа enable_if<> позволяет реализовать идиому SFINAE, отключив шаблон и просто заменив тип (в скобках <>) конструкцией с условием отключения шаблона.

Так, вспомогательный шаблон enable_if<> можно использовать для отключения объявления шаблонного конструктора

```
template<typename INIT>
Class(INIT&& init_value);
```

если переданный аргумент init_value представляет собой Class или выражение, которое может быть преобразовано в Class.

Проблема заключается в том, что согласно правилам разрешения перегрузки C++ для неконстантного lvalue шаблон специальной функции члена

```
template<typename INIT>
Class(INIT&& init_value);
```

оказывается лучшим соответствием, чем копирующий конструктор:

```
Class(Class const& obj);
```

INIT просто заменяется типом Class&, в то время как для копирующего конструктора необходимо преобразование в const. Кроме того, для объектов производного класса шаблон специальной функции-члена остается лучшим соответствием. Поэтому в действительности необходимо отключить шаблон члена для случая, когда переданный аргумент, как указывалось выше, представляет собой Class или выражение, которое может быть преобразовано в Class, чего и позволяет достичь конструкция std::enable_if<>.

На самом деле, чтобы отключить объявление шаблонного конструктора нужно воспользоваться ещё одним стандартным свойством типа std::is_convertible<FROM, TO>. Если тип INIT не может быть преобразован к типу init_value, то весь шаблон функции при использовании std::is_convertible<> будет игнорироваться, и копирующий конструктор оказывается доступным. Итого, соответствующее объявление, начиная с C++17, имеет вид (здесь использованы шаблонные псевдонимы для std::enable_if<> и std::is_convertible<>), если, например, необходимо игнорировать std::string:

```
template<
typename INIT,
typename = std::enable_if_t<std::is_convertible_v<Class, std::string>>
        >
Class(INIT&& init_value);
```

Помимо прочего, существует альтернатива использованию std::is_convertible<>, потому эта конструкция требует, чтобы типы были неявно преобразуемы. С помощью std:: is_constructible<> можно позволить использовать для инициализации явные преобразования. Однако в этом случае порядок аргументов будет противоположным.

Можно отметить, что, как правило, нельзя использовать enable_if<> для отключения стандартных копирующих/перемещающих конструкторов и операторов присваивания. Причина заключается в том, что шаблоны функций-членов никогда не учитываются как стандартные специальные функции-члены и игнорируются, когда, например, требуется копирующий конструктор.

Другое применение std::enable_if<> – запрет на передачу константных объектов по неконстантным ссылкам. Дело в том, при передаче const-аргументов шаблонной функции, вывод типа параметра шаблона может привести к тому, что передаваемый аргумент станет объявлением константной ссылки, что означает существование возможности передавать rvalue там, где ожидалось lvalue. Тогда дальнейшие попытки изменить значения (предполагаемой неконстантной ссылки) вызовут ошибку.

5 Какие правила вывода применяются при работе с шаблонами?

В действительности правил вывода при работе с шаблонами довольно много. Основная информация:

- В процессе вывода типы аргументов вызова функции сравниваются с соответствующими типами параметров шаблона функции, и компилятор пытается сделать вывод о том, что именно нужно подставить вместо одного или нескольких выведенных параметров. Анализ каждой пары (аргумент–параметр) проводится независимо, и, если выводы в конечном итоге отличаются, процесс вывода завершается неудачей.

- Пусть тип A (выведенный из типа аргумента) соответствует параметризованному типу P (выведенному из объявления параметра вызова). Если параметр объявлен как ссылка, считаем, что P – это тип, на который делается ссылка, а A – тип аргумента. Иначе P представляет собой объявленный тип параметра, а тип A получается из типа аргумента путем низведения типов массива или функции к указателю на соответствующий тип. При этом квалификаторы const и volatile игнорируются.

- Сложные объявления типов составляются из более простых конструкций (деклараторов указателей, ссылок, массивов и функций, идентификаторов шаблонов и т.д.). Процесс определения нужного типа происходит в нисходящем порядке, начиная с конструкций высокого уровня и рекурсивно продвигаясь к составляющим их элементам. Этим путем можно подобрать тип для большинства конструкций объявлений типов, и в этом случае они называются выводимым контекстом (deduced context). Однако некоторые конструкции выводимым контекстом не являются (поскольку в общем случае вывод может оказаться неоднозначным). Кроме того, возможны несколько ситуаций, в которых использующаяся для вывода пара (P, A) получается не из аргументов вызова функции и параметров шаблона функции.

- Когда в качестве аргумента функции выступает список инициализации, такой аргумент не имеет определенного типа, так что в общем случае вывод из данной пары (P, A) не выполняется за отсутствием A. Однако, если тип P параметра после удаления ссылочности и квалификаторов верхнего уровня const и volatile эквивалентен std::initializer_list$< P_0 >$ для некоторого типа $_0$, который имеет схему вывода, вывод продолжается путем сравнения P_0 с типом каждого элемента в списке инициализаторов, и успешно завершается, только если все элементы имеют один и тот же тип.

- При выполнении вывода аргумента шаблона для вариативных шаблонов один и тот же пакет параметров (P) сопоставляется нескольким аргументам (A), и каждое соответствие производит дополнительные значения для любого пакета параметров шаблонов в P.

- Когда параметр функции представляет собой пробрасываемую ссылку (forwarding reference), то вывод аргумента шаблона рассматривает не только тип аргумента вызова функции, но и выясняет, является этот аргумент lvalue или rvalue. В случаях, когда аргумент является lvalue, тип, определяемый выводом аргумента шаблона, представляет собой ссылку на lvalue на тип аргумента, а правила свёртки ссылок приводят к тому, что подставляемый параметр будет ссылкой на lvalue. В противном случае выведенный для параметра шаблона тип является просто типом аргумента (не ссылочным типом), и подставляемый параметр является ссылкой на rvalue на этот тип.

- Для вывода аргументов шаблона существуют ограничения.

- Стандарт C++11 включает возможность объявления переменной, тип которой выводится из ее инициализатора. Он также обеспечивает механизм для выражения типа именованной сущности (переменной или функции) или выражения. Эти возможности оказались очень удобными, и они получили дальнейшее развитие в C++14 и C++17:

  – Спецификатор типа auto может использоваться в ряде мест (главным образом, в областях видимости пространства имен и локальных) для вывода типа переменной из ее инициализатора. В таких случаях auto называется типом-заместителем (placeholder type). Вывод для auto использует тот же механизм, что и вывод аргумента шаблона.

    Выводимый тип-заместитель auto также может использоваться для определения типов возвращаемых значений функций.

  – Точный тип выражения или объявления может быть выражен с помощью типа заместителя decltype:

    * Если e является именем сущности (такой как переменная, функция, перечислитель или член-данные) или обращением к члену класса, то decltype(e) даёт объявленный тип этой сущности или упомянутого члена класса.
    * В противном случае, decltype(e) даёт тип, отражающий тип и категорию значения этого выражения следующим образом: если e – lvalue типа T, тогда decltype(e) даёт тип T&; если e xvalue типа T, decltype(e) даёт тип T&& и если e – prvalue типа T, decltype(e) даёт тип T.

  – decltype(auto) – тип-заместитель, при использовании которого фактический тип переменной, возвращаемый тип или аргумент шаблона определяется из типа связанного выражения (инициализатора, возвращаемого значения или аргумента шаблона) путём применения конструкции decltype непосредственно к выражению. Начиная с C++17, decltype(auto) может также использоваться для выводимых параметров, не являющихся типами.

- Шаблоны псевдонимов "прозрачны"по отношению к выводам. Это означает, что везде, где появляется шаблон псевдонима с некоторыми аргументами шаблона, выполняется подстановка аргументов в определение псевдонима, и получающийся в результате шаблон используется для вывода.

- Другие тонкости, например, вывод аргументов шаблонов классов...

Литература

[1] Вандевурд, Дэвид, Джосаттис, Николаи М., Грегор, Дуглас. В17 Шаблоны C++. Справочник разработчика, 2-е изд.: Пер. с англ. — СпБ.: ООО "Альфа-книга 2018. – 848 с.: ил. – Парал. гит. англ.