Const Correctness

Автор: Алекс Аллейн

Ключевое слово const позволяет указать, является ли переменная изменяемой. Вы можете использовать const для предотвращения изменений переменных и указателей const, а ссылки const предотвращают изменение данных, на которые указывают (или на которые ссылаются).

Но почему тебя это волнует?

Const дает вам возможность более четко документировать вашу программу и фактически применять эту документацию. Ключевое слово const обеспечивает пользователям гарантии, позволяющие оптимизировать производительность без угрозы повреждения их данных. Например, ссылки const позволяют указать, что данные, на которые ссылаются, не будут изменены; это означает, что вы можете использовать ссылки const как простой и непосредственный способ повышения производительности для любой функции, которая в данный момент принимает объекты по значению не беспокоясь о том, что ваша функция может изменить данные. Даже если это произойдет, компилятор предотвратит компиляцию кода и предупредит вас о проблеме. С другой стороны, если бы вы не использовали ссылки const, у вас не было бы простого способа убедиться, что ваши данные не были изменены.

Документация и безопасность

Основная цель constness-предоставить документацию и предотвратить ошибки программирования. Const позволяет дать понять себе и другим, что что-то менять не стоит. Более того, у него есть дополнительное преимущество: все, что вы объявляете const, на самом деле остается const, если не использовать силовые методы (о которых мы поговорим позже). Особенно полезно объявлять ссылочные параметры функций как ссылки const:

```
1 bool verifyObjectCorrectness (const myO
```

Здесь объект myObj передается по ссылке в verifyObjectCorrectness. В целях безопасности const используется для обеспечения того, чтобы verifyObjectCorrectness не могла изменить объект-в конце концов, он просто должен убедиться, что объект находится в допустимом состоянии. Это может предотвратить глупые ошибки программирования, которые в противном случае могут привести к повреждению объекта (например, путем установки поля класса для целей тестирования, что может привести к тому, что поле никогда не будет сброшено). Более того, объявив аргумент const, пользователи функции могут будьте уверены, что их объект не будет изменен и не нужно беспокоиться о возможных побочных эффектах вызова функции.

Примечание по синтаксису

При объявлении переменной const можно поставить const либо перед типом, либо после него: то есть и то, и другое

```
1 int const x = 5;
```

и

```
1 const int x = 4;
```

в результате x будет постоянным целым числом. Обратите внимание, что в обоих случаях значение переменной указывается в объявлении; нет никакого способа установить его позже!

Указатели const

Мы уже видели продемонстрированные ссылки const, и они довольно естественны: когда вы объявляете ссылку const, вы делаете только данные, на которые ссылаются const. Ссылки по самой своей природе не могут изменить то, на что они ссылаются. С другой стороны, указатели можно использовать двумя способами: изменить данные, на которые они указывают, или изменить сам указатель. Следовательно, существует два способа объявления указателя const: тот, который не позволяет изменить то, на что указано, и тот, который не позволяет изменить указанные данные.

Синтаксис объявления указателя на постоянные данные достаточно естественен:

```
1 const int *p_int;
```

Вы можете думать об этом как о том, что *p_int-это "const int". Таким образом, указатель может быть изменчивым, но вы определенно не можете коснуться того, на что указывает p_int. Ключ здесь в том, что const появляется перед *.

С другой стороны, если вы просто хотите, чтобы адрес, хранящийся в самом указателе, был const, то вы должны поставить const после *:

```
1 int x;
2 int * const p_int = &x;
```

Лично я нахожу этот синтаксис уродливым, но другого, очевидно, лучшего способа сделать это нет. Можно подумать, что "* const p_int" - это обычное целое число, и что значение, хранящееся в самом p_int, не может измениться, поэтому вы просто не можете изменить указанный адрес. Заметьте, кстати, что этот указатель должен был быть инициализирован при его объявлении: поскольку сам указатель является const, мы не можем изменить то, на что он указывает позже! Это правила игры.

Как правило, первый тип указателя, где данные неизменяемы, - это то, что я буду называть "указателем const" (отчасти потому, что это тот тип, который встречается чаще, поэтому у нас должен быть естественный способ его описания).

Const Функции

Последствия объявления переменной const распространяются по всей программе. Если у вас есть объект const, он не может быть назначен неконстантной ссылке или использовать функции, которые, как известно, способны изменять состояние объекта. Это необходимо для обеспечения постоянства объекта, но это означает, что вам нужен способ указать, что функция не должна вносить изменения в объект. В не объектно-ориентированном коде это так же просто, как использовать ссылки const, как показано выше.

Однако в C++ существует проблема классов с методами. Если у вас есть объект const, вы не хотите вызывать методы, которые могут изменить объект, поэтому вам нужен способ сообщить компилятору, какие методы можно безопасно вызывать. Эти методы называются "const-функциями" и являются единственными функциями, которые могут быть вызваны для объекта const. Обратите внимание, кстати, что только методы-члены имеют смысл как методы const. Помните, что в C++ каждый метод объекта получает неявный указатель this на объект; методы const эффективно получают указатель const this.

Способ объявить, что функция безопасна для объектов const, - это просто пометить ее как const; синтаксис функций const немного своеобразен, потому что есть только одно место, где вы действительно можете поместить const: в конце функции:

```
<return-value> <class>::<member-functic
{
        // ...
}
```

Например,

```
int Loan::calcInterest() const
{
        return loan_value * interest_ra
}
```

Обратите внимание, что только потому, что функция объявлена const, это не запрещает неконстантным функциям использовать ее; правило таково:

- Функции Const всегда можно вызвать
- Неконстантные функции могут вызываться только неконстантными объектами

Это имеет смысл: если у вас есть функция const, все это означает, что она гарантирует, что не изменит объект. Поэтому тот факт, что он const, не означает, что неконстантные объекты не могут его использовать.

На самом деле функции const имеют несколько более сильное ограничение, чем просто то, что они не могут изменять данные. Они должны быть сделаны так, чтобы их нельзя было использовать таким образом, чтобы вы могли использовать их для изменения данных const. Это означает, что когда функции const возвращают ссылки или указатели на члены класса, они также должны быть const.

Перегрузка Const

Во многом потому, что const-функции не могут возвращать неконстантные ссылки на данные объектов, во многих случаях может показаться целесообразным иметь как const, так и неконстантные версии функции. Например, если вы возвращаете ссылку на некоторые данные-члены (обычно это не очень хорошо, но есть исключения), то вам может понадобиться неконстантная версия функции, которая возвращает неконстантную ссылку:

```
int& myClass::getData()
{
        return data;
}
```

С другой стороны, вы не хотите, чтобы кто-то использовал константную версию вашего объекта,

```
myClass constDataHolder;
```

от получения данных. Вы просто хотите, чтобы этот человек не изменил его, вернув ссылку const. Но вы, вероятно, не хотите, чтобы имя функции менялось только потому, что вы меняете, является ли объект const или нет-среди прочего, это означало бы, что очень много кода может измениться только потому, что вы меняете способ объявления переменной-переходя от неконстантного к постоянному версия переменной была бы настоящей головной болью.

К счастью, C++ позволяет осуществлять перегрузку в зависимости от постоянства метода. Таким образом, вы можете иметь как константные, так и неконстантные методы, и будет выбрана правильная версия. Если вы хотите вернуть неконстантную ссылку в некоторых случаях, вам просто нужно объявить вторую, постоянную версию метода, который возвращает метод const:

```
// called for const objects only since
const int& myData::getData() const
{
        return data;
}
```

Const итераторы

Как мы уже видели, чтобы обеспечить постоянство, C++ требует, чтобы функции const возвращали только указатели const и ссылки. Поскольку итераторы также могут использоваться для изменения базовой коллекции, когда коллекция STL объявлена const, то любые итераторы, используемые над коллекцией, должны быть итераторами const. Они похожи на обычные итераторы, за исключением того, что их нельзя использовать для изменения базовых данных. (Поскольку итераторы являются обобщением идеи указателей, это имеет смысл.)

Итераторы Const в STL достаточно просты: просто добавьте "const_" к нужному вам типу итератора. Например, мы могли бы итератор над вектором следующим образом:

```
std::vector<int> vec;
vec.push_back( 3 );
vec.push_back( 4 );
vec.push_back( 8 );

for ( std::vector<int>::const_iterator
      itr != end;
      ++itr )
{
        // just print out the values..
        std::cout<< *itr <<std::endl;
}
```

Обратите внимание,что я использовал константный итератор для перебора неконстантной коллекции. Зачем это делать? По той же причине, по которой мы обычно используем const: он предотвращает возможность глупых ошибок программирования ("ой, я хотел сравнить два значения, а не назначить их!") и документирует, что мы никогда не собираемся использовать итератор для изменения коллекции.

Const cast

Иногда у вас есть переменная const, и вы действительно хотите передать ее в функцию, которая, как вы уверены, не изменит ее. Но эта функция не объявляет свой аргумент как const. (Это может произойти, например, если библиотечная функция C, такая как strlen, была объявлена без использования const.) К счастью, если вы знаете, что можете безопасно передавать переменную const в функцию, которая явно не указывает, что она не изменит данные, то вы можете использовать const_cast, чтобы временно удалить константу объекта.

Приведения Const выглядят как обычные типы в C++, за исключением того, что они могут использоваться только для отбрасывания постоянства (или изменчивости), но не для преобразования между типами или приведения иерархии классов.

```
// a bad version of strlen that doesn'
int bad_strlen (char *x)
{
        strlen( x );
```

```
}

// note that the extra const is actual
// string literals are constant
const char *x = "abc";

// cast away const-ness for our strler
bad_strlen( const_cast<char *>(x) );
```

Обратите внимание, что вы также можете использовать const_cast, чтобы пойти другим путем-добавить const-ness-если вы действительно хотите.

Повышение эффективности? Заметка о концептуальном против Побитовая константа

Одно из распространенных оправданий константной корректности основано на неправильном представлении о том, что константа может быть использована в качестве основы для оптимизации. К сожалению, это, как правило, не так-даже если переменная объявлена const, она не обязательно останется неизменной. Во-первых, можно отбросить константу, используя const_cast. Это может показаться глупым, когда вы объявляете параметр функции как const, но это возможно. Вторая проблема заключается в том, что в классах даже классы const могут быть изменены из-за ключевого слова mutable.

Изменяемые данные в классах Const

Во-первых, зачем вам вообще нужна возможность изменять данные в классе, объявленном const? Это лежит в основе того, что означает постоянство, и есть два способа думать об этом. Одна из идей-это "побитовая константа", которая в основном означает, что класс const должен иметь точно такое же представление в памяти во все времена. К сожалению (или к счастью), это не парадигма, используемая стандартом C++; вместо этого C++ использует "концептуальную константу". Концептуальная константа относится к идее, что выходные данные класса const всегда должны быть одинаковыми. Это означает, что базовые данные могут меняться до тех пор, пока фундаментальное поведение остается неизменным. (В сущности, "концепция" постоянна, но представление может меняться.)

Зачем нужна концептуальная постоянство?

Почему вы предпочитаете концептуальную константу побитовой константе? Одна из причин-эффективность: например, если в вашем классе есть функция, которая полагается на значение, вычисление которого занимает много времени, было бы более эффективно вычислить значение один раз, а затем сохранить его для последующих запросов. Это не изменит поведение функции-она всегда будет возвращать одно и то же значение. Однако это изменит представление класса, потому что у него должно быть какое-то место для кэширования значения.

C++ Поддержка концептуальной константы

C++ обеспечивает концептуальную константу с помощью ключевого слова mutable: при объявлении класса вы можете указать, что некоторые поля являются изменяемыми:

```
mutable int my_cached_result;
```

это позволит функциям const изменять поле независимо от того, был ли сам объект объявлен как const.

Другие способы достижения тех же выгод

Если вы планируете использовать const для повышения эффективности, подумайте о том, что это будет означать на самом деле-это было бы сродни использованию исходных данных без их копии. Но если бы вы хотели это сделать, самым простым подходом было бы просто использовать ссылки или указатели (предпочтительно ссылки const или указатели). Это дает вам реальный прирост эффективности, не полагаясь на оптимизацию компилятора, которой, вероятно, нет.

Опасность слишком большого постоянства

Остерегайтесь слишком большого использования const; например, то, что вы можете вернуть ссылку const, не означает, что вы должны вернуть ссылку const. Наиболее важным примером является то, что если у вас есть локальные данные в функции, вы действительно не должны возвращать ссылку на нее вообще (если она не статична), так как это будет ссылка на память, которая больше не действительна.

В другой раз, когда возврат ссылки const может быть не очень хорошей идеей, это когда вы возвращаете ссылку на данные-члены объекта. Хотя возврат ссылки const не позволяет кому-либо изменять данные с ее помощью, это означает, что у вас должны быть постоянные данные для поддержки ссылки-на самом деле это должно быть поле объекта, а не временные данные, созданные в функции. Как только вы сделаете ссылочную часть интерфейса классом, вы исправите детали реализации. Это может быть неприятно, если позже вы захотите измениться личные данные вашего класса, поэтому результат функции вычисляется при вызове функции, а не хранится в классе постоянно.

Краткие сведения

Не смотрите на const как на средство повышения эффективности, а скорее как на способ документировать свой код и гарантировать, что некоторые вещи не могут измениться. Помните, что константа распространяется по всей вашей программе, поэтому вы должны использовать функции const, ссылки const и итераторы const, чтобы гарантировать, что невозможно будет изменить данные, которые были объявлены const.

Части этой статьи были основаны на материалах более эффективного C++: 35 новых способов улучшить ваши программы и проекты Скотта Мейерса и исключительного стиля C++: 40 новых инженерных головоломок, проблем программирования и решений Херба Саттера.