Содержание

Обработка ошибок и исключения

Обработка ошибок — сложная задача, при решении которой программисту требуется вся помощь, которая только может быть предоставлена.

— Бьярн Страуструп (Bjarne Stroustrup), [Stroustrup94] §16.2

Имеется три способа написать программу без ошибок; но работает только третий способ.

— Алан Перлис (Alan Perlis)

Вопрос не в том, будем ли мы делать программные ошибки. Вопрос в том, будем ли мы что-либо предпринимать, чтобы позволить компилятору и другим используемым инструментам их обнаружить.

В этом разделе документированы добытые трудом множества программистов знания и наилучшие практические подходы, некоторые из которых стоили многих лет работы. Следуйте приведенным правилам и рекомендациям. Когда мы пишем сложную, надежную и безопасную программу — нам требуется вся помощь, которую мы только в состоянии получить.

В этом разделе мы считаем наиболее значимой рекомендацию 69 — “Определите разумную стратегию обработки ошибок и строго ей следуйте”.

68. Широко применяйте assert для документирования внутренних допущений и инвариантов

Резюме

Используйте assert или его эквивалент для документирования внутренних допущений в модуле (т.е. там, где вызываемый и вызывающий код поддерживаются одним и тем же программистом или командой), которые должны всегда выполняться (в противном случае они являются следствием программной ошибки; например, нарушение постусловий функции, обнаруженное вызывающим кодом). (См. также рекомендацию 70.) Убедитесь, что использование assert не приводит к побочным действиям.

Обсуждение

Очень трудно найти ошибку в своем коде, когда вы ищете ее;
но во сто крат труднее найти ее, если вы считаете,
что ее там нет.

— Стив Мак-Коннелл (Steve McConnell)

Трудно переоценить всю мощь assert. Этот макрос и его альтернативы, такие как шаблоны проверки времени компиляции (или, что несколько хуже, времени выполнения), представляют собой неоценимый инструментарий для обнаружения и отладки программных ошибок при работе над проектами. Среди прочих инструментов у них, пожалуй, наилучшее отношение сложность/эффективность.

Рассматриваемые проверки обычно генерируют код только в режиме отладки (когда не определен макрос NDEBUG), так что от них можно освободиться при сборке окончательной версии программы. Широко используйте проверки в своих программах, но никогда не пишите выражений в assert, которые могут иметь побочное действие. При построении окончательной версии, когда будет определен макрос NDEBUG, проверки не будут генерировать никакого кода:

```
assert( ++i < limit ); // Плохо: i увеличивается только в
                       // отладочном режиме
```

Согласно теории информации, количество информации, заключающееся в событии, обратно пропорционально вероятности данного события. То есть чем менее вероятно, что какая-то проверка сработает, тем больше информации вы получите, когда она сработает.

Избегайте применения assert(false), лучше использовать assert(!"информационное сообщение"). Большинство компиляторов вставят строку в вывод сообщения об ошибке. Подумайте также о добавлении &&"информационное сообщение" к более сложным проверкам вместо комментария.

Рассмотрим определение вашего собственного assert. Стандартный макрос assert просто бесцеремонно завершает вашу программу с выводом сообщения в стандартный поток вывода. Ваша среда, вероятно, обладает расширенными возможностями отладки; пусть, например, она в состоянии автоматически запустить интерактивный отладчик. В этом случае вы можете определить собственный макрос MYASSERT и использовать его. Может также оказаться полезным оставить большинство проверок даже в окончательной версии программы (лучше не отключать проверки по соображениям эффективности, пока необходимость этого

отключения не будет точно доказана; см. рекомендацию 8), так что существенные преимущества может предоставить наличие различных “уровней проверки”, некоторые из которых могут оставаться активными и в окончательной версии программы.

Проверки зачастую связаны с условиями, которые можно было бы протестировать во время компиляции, если бы язык был достаточно выразителен для этого. Например, ваш проект может полагаться на то, что каждый объект класса `Employee` имеет ненулевой идентификатор `id_`. В идеале компилятор мог бы анализировать конструктор `Employee` и его члены и доказать при помощи статического анализа, что указанное условие всегда выполняется. В реальной ситуации вы можете использовать `assert(id_!=0)` в реализации `Employee`:

```
unsigned int Employee::GetID() {
    assert(id_!=0 && "Employee ID должен быть ненулевым") ;
    return id_;
}
```

Не используйте `assert` для сообщения об ошибках времени выполнения (см. рекомендации 70 и 72). Например, не следует применять `assert`, чтобы убедиться в корректной работе `malloc`, успешном создании окна или запуске потока программы. Однако можно использовать `assert`, чтобы убедиться, что API работает так, как документировано. Например, если вы вызываете некоторую функцию API, в документации на которую сказано, что она всегда возвращает положительное значение, но вы подозреваете наличие в ней ошибки — после вызова этой функции можно воспользоваться `assert` для проверки выполнения постусловия.

Не рекомендуется вместо проверок генерировать исключения, несмотря на то, что именно для этой цели был разработан стандартный класс `std::logic_error`. Главный недостаток использования исключений для сообщения о программных ошибках состоит в том, что при этом не требуется свертка стека — желательно вызвать отладчик именно в той строке, где обнаружено нарушение, с полным сохранением состояния программы.

Резюмируя: имеются ошибки, о которых вы знаете, что они могут произойти (см. рекомендации с 69 по 75). Все остальные ошибки произойти не должны, и если это все же случается — то это ошибка программиста. Для таких ошибок имеется `assert`.

Примеры

Пример. Проверка базовых допущений. Все мы сталкивались с ситуациями, когда происходило что-то, что “ну никак не может произойти”. Но часто даже собственный опыт мало чему учит, и через некоторое время опять начинается — “это значение может быть только положительным!”, “совершенно очевидно, что этот указатель не нулевой!”… Разработка программного обеспечения — работа сложная, и в программе, в которую вносятся изменения, может произойти все, что угодно. Проверки предназначены для того, чтобы убедиться в справедливости ваших предположений. Не стесняйтесь проверять тавтологии, которые не в состоянии обеспечить система:

```
string Date::DayOfWeek() const {
    // Проверка инвариантов
    assert( day_ > 0 && day_ <= 31 );
    assert( month_ > 0 && month_ <= 12 );
    // ...
}
```

Ссылки

[Abrahams01b] • [Alexandrescu03b] • [Alexandrescu03c] • [Allison98] §13 • [Cargill92] pp. 34-35 • [Cline99] §10.01-10 • [Dewhurst03] §28 • [Keffer95] pp. 24-25 • [Lakos96] §2.6, §10.2.1 • [McConnell93] §5.6 • [Stroustrup00] §24.3.7, §E.2, §E.3.5, §E.6 • [Sutter00] §47

69. Определите разумную стратегию обработки ошибок и строго ей следуйте

Резюме

Еще на ранней стадии проектирования разработайте практичную, последовательную и разумную стратегию обработки ошибок и строго следуйте ей. Убедитесь, что ваша стратегия включает следующее.

- *Идентификация*: какие условия являются ошибкой.
- *Строгость*: насколько важна каждая ошибка.
- *Обнаружение*: какой код отвечает за обнаружение ошибки.
- *Распространение*: какой механизм используется для описания и распространения уведомления об ошибке в каждом модуле.
- *Обработка*: какой код отвечает за выполнение действий, связанных с ошибкой.
- *Уведомление*: каким образом информация об ошибке вносится в журнальный файл или производится уведомление пользователя программы.

Изменяйте механизмы обработки ошибок только в пределах границ модуля.

Обсуждение

В этой рекомендации мы рассматриваем ошибки времени выполнения, возникновение которых не связано с неверным кодированием (таким ошибкам посвящена рекомендация 68).

Определите стратегию сообщения об ошибках и их обработки для вашего приложения и для каждого модуля или подсистемы, и строго следуйте ей. Стратегия должна включать, как минимум, следующие пункты.

Везде.

- *Определение ошибок*. Для каждой сущности (например, для каждой функции, класса, модуля) документируйте внутренние и внешние инварианты.

Для каждой функции.

- *Определение ошибок*. Для каждой функции документируйте ее пред- и постусловия, инварианты, за которые она отвечает, и гарантии безопасности, которые она поддерживает (см. рекомендации 70 и 71). Заметим, что деструкторы и функции освобождения ресурсов должны всегда поддерживать гарантию бессбойности, поскольку в противном случае часто невозможно надежно и безопасно выполнить освобождение захваченных ресурсов (см. рекомендацию 51).

Для каждой ошибки (см. определение “ошибки” в рекомендации 70).

- *Серьезность ошибки и категоризация*. Для каждой ошибки определите уровень серьезности. Желательно предоставить способ тонкой настройки диагностики для определенных категорий и уровней ошибок.
- *Обнаружение ошибок*. Для каждой ошибки документируйте, какой именно код отвечает за ее обнаружение, следуя советам рекомендации 70.
- *Обработка ошибок*. Для каждой ошибки определите код, который отвечает за ее обработку, следуя советам рекомендации 74.

- *Уведомление об ошибках*. Для каждой ошибки определите соответствующий метод уведомления. Сюда входят запись на диск в журнальный файл, распечатка или даже отправка SMS на мобильный телефон администратора.

Для каждого модуля.

- *Передача ошибки*. Для каждого модуля (обратите внимание: для каждого модуля, а не для каждой ошибки) определите механизм, который будет использоваться для передачи информации об ошибке (например, исключения C++, исключения COM, исключения CORBA, коды возврата).

Мы уже подчеркивали, что стратегия обработки ошибок может изменяться только на границах модулей (см. рекомендации 62 и 63). Каждый модуль должен последовательно использовать единую стратегию обработки ошибок внутри модуля (например, модули, написанные на C++, должны использовать исключения; см. рекомендацию 72), и последовательно пользоваться единой, хотя, возможно, иной стратегией обработки ошибок для своего интерфейса (например, модуль может предоставлять обычный API на языке C, чтобы обеспечить возможность его использования кодом, написанном на разных языках программирования; или использовать оболочку COM и, соответственно, исключения COM).

Все функции, являющиеся точками входа в модуль, непосредственно отвечают за преобразование между внутренней и внешней стратегиями, если они различны. Например, в модуле, который внутренне использует исключения C++, но предоставляет интерфейс в стиле C API, все функции интерфейса должны содержать перехват `catch(...)` всех исключений и преобразовывать их в коды ошибок.

Обратите внимание, в частности, на то, что функции обратного вызова и функции потоков по определению являются (или могут быть) границами модуля. Тело каждой функции обратного вызова или функции потока должно преобразовывать внутренний механизм ошибок в механизм, использующийся стратегией интерфейса (см. рекомендацию 62).

Ссылки

[Abrahams01b] • [Allison98] §13 • [McConnell93] §5.6 • [Stroustrup94] §16.2, §E.2 • [Stroustrup00] §14.9, §19.3.1 • [Sutter04b]

70. Отличайте ошибки от ситуаций, не являющихся ошибками

Резюме

Функция представляет собой единицу работы. Таким образом, сбои следует рассматривать либо как ошибки, либо как штатные ситуации, в зависимости от их влияния на функции. В функции `f` сбой является ошибкой тогда и только тогда, когда он нарушает одно из предусловий `f`, не позволяет выполнить предусловие вызываемой ею функции, препятствует достижению собственных постусловий `f` или сохранению инварианта, за поддержку которого отвечает функция `f`.

В частности, в этой рекомендации мы исключаем внутренние программные ошибки (т.е. те, где за вызывающий и вызываемый код отвечает один и тот же человек или команда, например, в пределах одного модуля). Они представляют собой отдельную категорию ошибок, для работы с которой используется такое средство, как проверки (см. рекомендацию 68).

Обсуждение

Очень важно четко различать ошибки и ситуации, не являющиеся ошибками в плане их влияния на работу функций, в особенности в целях определения гарантий безопасности (см. рекомендацию 71). Ключевыми словами данной рекомендации являются *предусловие*, *постусловие* и *инвариант*.

Функция представляет собой базовую единицу работы, независимо от того, программируете ли вы на C++ в структурном, объектно-ориентированном или обобщенном стиле. Функция делает определенные предположения о начальном состоянии (предусловия, за выполнение которых несет ответственность вызывающий код, а за проверку — вызываемый) и выполняет одно или несколько действий (документируемых, как результат выполнения функции, или ее постусловия, за выполнение которых несет ответственность данная функция). Функция может (наряду с другими функциями) нести ответственность за поддержание одного или нескольких инвариантов. В частности, не закрытая неконстантная функция-член по определению представляет собой единицу работы над объектом, и должна перевести объект из одного корректного, сохраняющего инвариант состояния в другое. Во время выполнения тела функции-члена инвариант объекта может (и практически всегда должен) нарушаться, и это вполне нормальная ситуация, лишь бы по окончании работы функции-члена инвариант вновь выполнялся. Функции более высокого уровня объединяют функции более низкого уровня в бо́льшие единицы работы.

Ошибкой является любой сбой, который не дает функции успешно завершиться. Имеется три вида ошибок.

- *Нарушение или невозможность достижения предусловия*. Функция обнаруживает нарушение одного из своих собственных предусловий (например, ограничения, накладываемого на параметр или состояние) или сталкивается с условием, которое не позволяет достичь выполнения предусловия для некоторой другой неотъемлемой функции, которая должна быть вызвана.
- *Неспособность достичь постусловия*. Функция сталкивается с ситуацией, которая не позволяет ей выполнить одно из ее собственных постусловий. Если функция возвращает значение, получение корректного возвращаемого значения является ее постусловием.

- *Неспособность восстановления инварианта*. Функция сталкивается с ситуацией, которая не позволяет ей восстановить инвариант, за поддержку которого она отвечает. Это частный случай постусловия, который в особенности относится к функциям-членам. Важнейшим постусловием всех не закрытых функций-членов является восстановление инварианта класса (см. [Stroustrup00] §E.2.)

Все прочие ситуации ошибками не являются, и, следовательно, уведомлять о них, как об ошибках, не требуется (см. примеры к данной рекомендации).

Код, который может вызвать ошибку, отвечает за ее обнаружение и уведомление о ней. В частности, вызывающий код должен обнаружить и уведомить о ситуации, когда он не в состоянии выполнить предусловия вызываемой функции (в особенности если для вызываемой функции документировано отсутствие проверок с ее стороны; так, например, оператор `vector::operator[]` не обязан выполнять проверку попадания аргумента в корректный интервал значений). Однако поскольку вызываемая функция не может полагаться на корректность работы вызывающего кода, желательно, чтобы она выполняла собственные проверки предусловий и уведомляла об обнаруженных нарушениях, генерируя ошибку (или, если функция является внутренней для (т.е. вызываемой только в пределах) модуля, то нарушение предусловий по определению является программной ошибкой и должно обрабатываться при помощи `assert` (см. рекомендацию 68)).

Добавим пару слов об определении предусловий функций. Условие является предусловием функции `f` тогда и только тогда, когда имеются основания ожидать, что весь вызывающий код проверяет выполнение данного условия перед вызовом функции `f`. Например, было бы неверно полагать предусловием нечто, что может быть проверено только путем выполнения существенной работы самой функцией, либо путем доступа к закрытой информации. Такая работа должна выполняться в функции и не дублироваться вызывающим ее кодом.

Например, функция, которая получает объект `string`, содержащий имя файла, обычно не должна делать условие существования файла предусловием, поскольку вызывающий код не в состоянии надежно гарантировать, что данный файл существует, не используя блокировки файла (при проверке существования файла без блокировки другой пользователь или процесс могут удалить или переименовать этот файл в промежутке между проверкой существования файла вызывающим кодом и попыткой открытия вызываемым кодом). Единственный корректный способ сделать существование файла предусловием — это потребовать, чтобы вызывающий код открыл его, а параметром функции сделать `ifstream` или его эквивалент (что к тому же безопаснее, поскольку работа при этом выполняется на более высоком уровне абстракции; см. рекомендацию 63), а не простое имя файла в виде объекта `string`. Многие предусловия таким образом могут быть заменены более строгим типизированием, которое превратит ошибки времени выполнения в ошибки времени компиляции (см. рекомендацию 14).

Примеры

Пример 1. `std::string::insert` (ошибка предусловия). При попытке вставить новый символ в объект `string` в определенной позиции `pos`, вызывающий код должен проверить корректность значения `pos`, которое не должно нарушать документированные требования к данному параметру; например, чтобы не выполнялось соотношение `pos > size()`. Функция `insert` не может успешно выполнить свою работу, если для нее не будут созданы корректные начальные условия.

Пример 2. `std::string::append` (ошибка постусловия). При добавлении символа к объекту `string` сбой при выделении нового буфера, если заполнен существующий, не позволит функции выполнить документированные действия и получить документированные же постусловия, так что такой сбой является ошибкой.

Пример 3. Невозможность получения возвращаемого значения (ошибка постусловия). Получение корректного возвращаемого объекта является постусловием для функции, которая возвращает значение. Если возвращаемое значение не может быть корректно создано (например, если функция возвращает `double`, но значение `double` с требуемыми математическими свойствами не существует), то это является ошибкой.

Пример 4. `std::string::find_first_of` (не ошибка в контексте `string`). При поиске символа в объекте `string`, невозможность найти искомый символ — вполне законный итог поиска, ошибкой не являющийся. Как минимум, это не ошибка при работе с классом `string` общего назначения. Если владелец данной строки предполагает, что символ должен наличествовать в строке, и его отсутствие, таким образом, является ошибкой в соответствии с высокоуровневым инвариантом, то высокоуровневый вызываемый код должен соответствующим образом уведомить об ошибке инварианта.

Пример 5. Различные условия ошибок в одной функции. Несмотря на увеличивающуюся надежность дисковых носителей, запись на диск традиционно сопровождается ожиданием ошибок. Если вы разрабатываете класс `File`, в одной-единственной функции `File::Write(const char*buffer, size_t size)`, которая требует, чтобы файл был открыт для записи, а указатель `buffer` имел ненулевое значение, вы можете предпринимать следующие действия.

- *Если `buffer` равен `NULL`*: сообщить об ошибке нарушения предусловия.
- *Если файл открыт только для чтения*: сообщить об ошибке нарушения предусловия.
- *Если запись выполнена неуспешно*: сообщить об ошибке нарушения постусловия, поскольку функция не в состоянии выполнить свою работу.

Пример 6. Различный статус одного и того же условия. Одно и то же условие может быть корректным предусловием для одной функции и не быть таковым для другой. Выбор зависит от автора функции, который определяет семантику интерфейса. В частности, `std::vector` предоставляет два пути для выполнения индексированного доступа: оператор `operator[]`, который не выполняет проверок выхода за пределы диапазона, и функцию `at`, которая такую проверку выполняет. И оператор `operator[]`, и функция `at` требуют выполнения предусловия, состоящего в том, что аргумент не должен выходить за пределы диапазона. Поскольку от оператора `operator[]` не требуется проверка его аргумента, должно быть четко документировано, что вызывающий код отвечает за то, чтобы аргумент оператора находился в допустимом диапазоне значений; понятно, что данная функция небезопасна. Функция же `at` в той же ситуации вполне безопасна, поскольку документировано, что она проверяет принадлежность своего аргумента к допустимому диапазону значений, и если аргумент выходит за пределы допустимого диапазона значений, то она сообщает об ошибке (путем генерации исключения `std::out_of_range`).

Ссылки

[Abrahams01b] • [Meyer00] • [Stroustrup00] §8.3.3, §14.1, §14.5 • [Sutter04b]

71. Проектируйте и пишите безопасный в отношении ошибок код

Резюме

В каждой функции обеспечивайте наиболее строгую гарантию безопасности, какой только можно добиться без дополнительных затрат со стороны вызывающего кода, не требующего такого уровня гарантии. Всегда обеспечивайте, как минимум, базовую гарантию безопасности.

Убедитесь, что при любых ошибках ваша программа всегда остается в корректном состоянии (в этом и заключается базовая гарантия). Остерегайтесь ошибок, нарушающих инвариант (включая утечки, но не ограничиваясь ими).

Желательно дополнительно гарантировать, что конечное состояние либо является исходным состоянием (в результате отката после происшедшей ошибки), либо корректно вычисленным целевым состоянием (если ошибок не было). Это — строгая гарантия безопасности.

Еще лучше гарантировать, что сбой в процессе операции невозможен. Хотя для большинства функций это невозможно, такую гарантию следует обеспечить для таких функций, как деструкторы и функции освобождения ресурсов. Данная гарантия — гарантия бессбойности.

Обсуждение

Базовая, строгая гарантии и гарантия бессбойности (известная также как гарантия отсутствия исключений) впервые были описаны в [Abrahams96] и получили широкую известность благодаря публикациям [GotW], [Stroustrup00, §E.2] и [Sutter00], посвященным вопросам безопасности исключений. Эти гарантии применимы к обработке любых ошибок, независимо от конкретного использованного метода, так что мы можем воспользоваться ими при описании безопасности обработки ошибок в общем случае. Гарантия бессбойности является строгим подмножеством строгой гарантии, а строгая гарантия, в свою очередь, является строгим подмножеством базовой гарантии.

В общем случае каждая функция должна обеспечивать наиболее строгую гарантию, которую она в состоянии обеспечить без излишних затрат для вызывающего кода, которому не требуется такая степень гарантии. Там, где это возможно, следует дополнительно обеспечить достаточную функциональность для того, чтобы требующий более строгую гарантию код мог получить ее (см. в примерах к данной рекомендации случай `vector::insert`).

В идеале следует писать функции, которые всегда успешно выполняются и, таким образом, могут обеспечить гарантию бессбойности. Некоторые функции должны всегда обеспечивать такую гарантию, в частности, деструкторы, функции освобождения ресурсов и функции обмена (см. рекомендацию 51).

Однако в большинстве функций могут произойти сбои. Если ошибка возможна, наиболее безопасным будет гарантировать транзакционное поведение функции: либо функция выполняется успешно и программа переходит из начального корректного состояния в корректное целевое состояние, либо — в случае сбоя — программа остается в том же состоянии, в котором находилась перед вызовом функции, т.е. видимые состояния всех объектов после сбойного вызова оказываются теми же, что и до него (например, значение глобальной целой переменной не может измениться с 42 на 43), и любое действие, которое вызывающий код мог предпринять до сбойного вызова, должно остаться возможным (с тем же смыслом) и после сбойного вызова (например, ни один итератор контейнера не должен стать недействительным; применение оператора ++ к упомянутой глобальной целой переменной даст значение 43, а не 44). Такая гарантия безопасности называется строгой.

Наконец, если обеспечить строгую гарантию оказывается слишком сложно или излишне дорого, следует обеспечить, по крайней мере, базовую гарантию: функция либо выполняется успешно и достигает целевого состояния, либо она неуспешна и оставляет программу в состоянии корректном (сохраняя инварианты, за которые отвечает данная функция), но не предсказуемом (это может быть исходное состояние, но может и не быть таковым; некоторые из постусловий могут быть не выполнены; однако все инварианты должны быть восстановлены). Ваше приложение должно быть спроектировано таким образом, чтобы суметь соответствующим образом обработать такой результат работы функции.

Вот и все; более низкого уровня гарантии не существует. Функция, которая не дает даже базовой гарантии — это просто ошибка программиста. Корректная программа должна отвечать, как минимум, базовой гарантии для всех функций. Даже те немногие корректные программы, которые сознательно идут на утечку ресурсов, в частности, в ситуациях, когда программа аварийно завершается, поступают так с учетом того, что ресурсы будут освобождены операционной системой. Всегда разрабатывайте код таким образом, чтобы корректно освобождались все ресурсы, а данные находились в согласованном состоянии даже при наличии ошибок, если только ошибка не приводит к немедленному аварийному завершению программы.

При принятии решения о том, какой уровень гарантии следует поддерживать, надо не забывать о перспективе развития проекта. Всегда проще усилить гарантию в последующих версиях, в то время как снижение степени гарантии ведет к необходимости переделки вызывающего кода, полагающегося на предоставление функцией более строгой гарантии.

Помните, что “небезопасность в отношении ошибок” и “плохое проектирование” идут рука об руку: если некоторую часть кода сложно сделать обеспечивающей базовую гарантию, то почти всегда это говорит о плохом проектировании. Например, если функция отвечает за выполнение нескольких несвязанных задач, ее трудно сделать безопасной в отношении ошибок (см. рекомендацию 5).

Остерегайтесь оператора копирующего присваивания, которому для корректной работы требуется проверка, не выполняется ли присваивание объекта самому себе. Безопасный в отношении ошибок оператор копирующего присваивания автоматически безопасен и в плане присваивания самому себе. Использовать проверку присваивания самому себе можно только в качестве оптимизации, для того чтобы избежать излишней работы (см. рекомендацию 55).

Примеры

Пример 1. Повторная попытка после сбоя. Если ваша программа включает команду для сохранения данных в файл и во время записи произошел сбой, убедитесь, что вы вернулись к состоянию, когда операция может быть повторена. В частности, не освобождайте никакие структуры данных до тех пор, пока данные не будут полностью сброшены на диск. Это не теоретизирование — нам известен один текстовый редактор, который не позволяет изменить имя файла для сохранения данных после ошибки записи.

Пример 2. Текстуры. Если вы пишете приложение, у которого можно менять внешний вид, загружая новые текстуры, то учтите, что не следует уничтожать старые текстуры до тех пор, пока не будут полностью загружены и применены новые. В противном случае при сбое во время загрузки новых текстур ваше приложение может оказаться в нестабильном состоянии.

Пример 3. `std::vector::insert`. Поскольку внутреннее представление `vector<T>` использует непрерывный блок памяти, вставка элемента в средину требует перемещения ряда имеющихся значений на одну позицию для освобождения места для вставляемого элемента. Перемещение выполняется с использованием копирующего конструктора `T::T(const T&)` и оператора присваивания `T::operator=`, и если одна из этих операций может сбоить (генерировать исключение), то единственный способ обеспечить строгую гарантию — это

сделать полную копию контейнера, выполнить операцию над копией, а затем обменять оригинал и копию с использованием бессбойной функции `vector<T>::swap`.

Однако выполнение всех указанных действий обходится излишне дорого как в плане требуемой памяти, так и процессорного времени. Таким образом, обеспечение строгой гарантии даже тогда, когда она не является необходимой, оказывается чрезмерно расточительным. Поэтому строгую гарантию вызывающий код должен при необходимости обеспечивать самостоятельно — для этого шаблон `vector` предоставляет все необходимые инструменты. (В лучшем случае, если тип содержимого контейнера не генерирует исключений в копирующем конструкторе и копирующем операторе присваивания, никаких дополнительных действий не требуется. В худшем следует создать копию вектора, выполнить вставку в копию и после успешного завершения данной операции обменять копию и исходный вектор.)

Пример 4. Запуск спутника. Рассмотрим функцию `f`, в которой частью ее работы является запуск спутника, и используемую ею функцию `LaunchSatellite`, обеспечивающую гарантию не ниже строгой. Если функция `f` может выполнить всю работу, при которой может произойти сбой, до запуска спутника, то `f` способна обеспечить строгую гарантию. Но если `f` должна выполнить некоторые операции, в процессе которых может произойти сбой, уже после запуска спутника, то обеспечение строгой гарантии оказывается невозможным — вернуть запущенный спутник на стартовую площадку уже нельзя. (Такую функцию `f` следует разделить по крайней мере на две, поскольку одна функция не должна даже пытаться выполнить несколько различных действий такой важности; см. рекомендацию 5.)

Ссылки

[Abrahams96] • [Abrahams01b] • [Alexandrescu03d] • [Josuttis99] §5.11.2 • [Stroustrup00] §14.4.3, §E.2-4, §E.6 • [Sutter00] §8-19, §40-41, §47 • [Sutter02] §17-23 • [Sutter04] §11-13 • [Sutter04b]

72. Для уведомления об ошибках следует использовать исключения

Резюме

Для уведомления об ошибках лучше использовать механизм исключений, а не коды ошибок. Применять коды состояния (например, коды ошибок, переменную `errno`) следует только тогда, когда нельзя использовать исключения (см. рекомендацию 62), а также для ситуаций, которые не являются ошибками. К другим методам, таким как экстренное завершение программы (или плановое завершение с освобождением ресурсов и т.п. действиями), следует прибегать только в ситуациях, когда восстановление после ошибки невозможно (или не требуется).

Обсуждение

То, что современные языки программирования, созданные в течение последних 20 лет, используют в качестве основного механизма сообщения об ошибках исключения, — не случайность. Практически по определению исключения предназначены для уведомления об исключениях в нормальном процессе — известных также как "ошибки", которые определены в рекомендации 70 как нарушения предусловий, постусловий и инвариантов. Так же, как и все другие механизмы уведомления об ошибках, исключения не должны генерироваться при нормальной успешной работе.

Далее мы будем использовать термин "коды состояния" для всех видов сообщения об ошибках посредством кодов (включая коды возврата, `errno`, функцию `GetLastError` и прочие стратегии возврата или получения кодов), а термин "коды ошибок" — для тех кодов состояния, которые означают ошибки. В C++ сообщение об ошибках посредством исключений имеет явные преимущества перед уведомлением посредством кодов ошибок.

- *Исключения невозможно проигнорировать*. Самое слабое место кодов ошибок заключается в том, что по умолчанию они игнорируются; чтобы уделить хотя бы минимальное внимание кодам ошибок, вы должны явно писать код, который опрашивает код ошибки и отвечает на него. Весьма распространенная среди программистов практика — случайно (или из-за лени) забыть опросить код ошибки. Исключения же невозможно просто проигнорировать; чтобы проигнорировать исключение, вы должны явно перехватить его (даже если вы сделаете это при помощи единственной инструкции `catch(...)`) и никак на него не отреагировать.
- *Исключения распространяются автоматически*. Коды ошибок по умолчанию за пределы области видимости не распространяются; для того, чтобы информировать высокоуровневую вызывающую функцию о низкоуровневой ошибке, программист должен написать промежуточный код, который передаст информацию об ошибке. Исключения автоматически распространяются за пределы области видимости до тех пор, пока они не будут перехвачены. ("*Это не самое разумное — пытаться сделать из каждой функции брандмауэр*". — [Stroustrup94, §16.8])
- *Исключения выносят обработку ошибок и восстановление после них из основного потока управления*. Проверка кода ошибки и ее обработка перемежается с основным потоком управления программы, тем самым запутывая его. Это затрудняет понимание и сопровождение как основного кода программы, так и кода обработки ошибок. Обработка исключений естественным образом перемещает обнаружение ошибок и восстановление после них в отдельные `catch`-блоки, т.е. делают обработку ошибок существенно более

модульной. Тем самым основной код программы, как и код обработки ошибок, оказывается более понятным и легче сопровождаемым.

- *Исключения оказываются наилучшим способом уведомления об ошибках в конструкторах и операторах*. Копирующие конструкторы и операторы имеют предопределенные сигнатуры, в которых просто нет места для кодов возврата. В частности, конструкторы вообще не имеют возвращаемого типа (даже `void`), а, например, каждый оператор `operator+` должен получать в точности два параметра и возвращать только один объект (предопределенного типа; см. рекомендацию 26). В случае операторов использование кодов ошибок, как минимум, возможно, если не желательно; для этого можно воспользоваться глобальной переменной наподобие `errno` или несколько более худшим методом внедрения кода состояния в состав объекта. Но для конструкторов использование кодов ошибок неприменимо, поскольку в языке C++ исключения в конструкторе и сбои в конструкторе настолько тесно связаны, что по сути являются синонимами. Если вы попытаетесь использовать подход с глобальной переменной наподобие

```
SomeType anObject; // Конструирование объекта
if( SomeType::ConstructionWasOk() ) { // Проверка результата
    // ...                            // конструирования
```

то результат оказывается не только уродливым и подверженным ошибкам, но и ведет к “незаконнорожденным” объектам, которые признаются корректными, но на самом деле не удовлетворяют инвариантам типа. Это связано с возможными условиями гонки при использовании функции `SomeType::ConstructionWasOk` в многопоточных приложениях (см. [Stroustrup00] §E.3.5).

Главный потенциальный недостаток обработки исключений заключается в том, что требует от программиста хороших знаний о некоторых идиомах, возникающих в результате вынесения исключений из основного потока управления. Например, деструкторы и функции освобождения ресурсов не должны генерировать исключения (см. рекомендацию 51), а промежуточный код должен быть корректен при наличии исключений (см. рекомендацию 71 и ссылки); распространенная идиома для достижения этого заключается в отдельном выполнении всей работы, которая может привести к генерации исключений, и только после успешного завершения результаты работы принимаются, и состояние программы модифицируется с использованием только тех операций, которые предоставляют гарантию бессбойности (см. рекомендацию 51 и [Sutter00] §9-10, §13). Однако использование кодов ошибок также имеет собственные идиомы. Просто эти идиомы более старые и их знает большее количество программистов — но при этом, к сожалению, зачастую их просто игнорирует…

Обычно использование обработки исключений не приводит к снижению производительности. Для начала заметим, что вы всегда должны включать поддержку обработки исключений в вашем компиляторе, даже если по умолчанию она отключена; в противном случае вы не сможете получить предусмотренное стандартом поведение и уведомление об ошибках от таких операций C++, как оператор `new` или операции стандартной библиотеки наподобие вставок в контейнер (см. раздел исключений в данной рекомендации).

[Небольшое отступление. Включение поддержки обработки исключений может быть реализовано так, что оно увеличит размер выполнимого файла (что неизбежно), но при этом практически не повлияет на производительность приложения при отсутствии сгенерированных исключений, причем некоторые компиляторы именно так и поступают. Другие компиляторы приводят к определенным накладным расходам, в особенности при обеспечении режима безопасности для предотвращения атак посредством переполнения буфера в механизме обработки исключений. Однако имеются ли накладные расходы, связанные

с механизмом обработки исключений, или нет — в любом случае, вы должны включить его, иначе вы не сможете получать сообщения об ошибках от языка программирования и стандартной библиотеки.]

При включенной поддержке обработки исключений компилятором разница в производительности между генерацией исключений и возвратом кода ошибки при нормальной работе (в отсутствие ошибок) обычно незначительна. Определенную разницу в производительности можно заметить только при ошибках, но если ошибки происходят так часто, что становится ощутимой разница в производительности — скорее всего, вы используете исключения для ситуаций, которые ошибками не являются, а значит, вы не смогли корректно вычленить ошибки из прочих ситуаций (см. рекомендацию 70). Если же это действительно ошибки, которые нарушают пред-, постусловия или инварианты, и они встречаются настолько часто — значит, у вашей программы имеются гораздо более серьезные проблемы, чем снижение производительности из-за обработки исключений.

Один из симптомов неверного употребления кодов ошибок — когда коду приложения приходится постоянно проверять тривиальные истинные условия, либо (что еще хуже) когда приложение не проверяет коды ошибок, которые должно проверять.

Симптом злоупотребления исключениями — когда код приложения генерирует и перехватывает исключения настолько часто, что количества успешных и неуспешных выполнений `try`-блока оказываются величинами одного порядка. Такой `catch`-блок либо в действительности обрабатывает не истинные ошибки (которые нарушают пред-, постусловия или инварианты), либо у вашей программы имеются серьезные проблемы.

Примеры

Пример 1. Конструкторы (ошибка инварианта). Если конструктор не способен успешно создать объект своего типа (что означает, что он не способен установить все инварианты нового объекта), он должен сгенерировать исключение. Верно и обратное — исключение, сгенерированное конструктором, означает, что конструирование объекта не выполнено и что время жизни объекта никогда не начиналось.

Пример 2. Успешный рекурсивный поиск в дереве. При поиске в дереве с использованием рекурсивного алгоритма может показаться заманчивой идея вернуть результат — и легко свернуть стек — генерируя исключение с результатом поиска. Но так делать не следует: исключение означает ошибку, а результат поиска ошибкой не является (см. [Stroustrup00]). (Заметим, что, конечно, неуспешный поиск также не является ошибкой в контексте функции поиска; см. пример с функцией `find_first_of` в рекомендации 70.)

Обратитесь также к примерам рекомендации 70, заменяя в их тексте “сообщение об ошибке” на “генерация исключения”.

Исключения

В редких случаях можно рассмотреть возможность использования кодов ошибок, если оказываются выполнены следующие условия.

- *Не применимы преимущества исключений*. Например, вы знаете, что практически всегда ошибка будет обрабатываться непосредственно вызывающим кодом, так что распространение ошибки никогда (или практически никогда) не будет востребовано. Это весьма редкая ситуация, поскольку обычно вызываемая функция не имеет достаточной информации о том, какой код будет ее вызывать.

- *Измерения показывают наличие реального снижения производительности при использовании исключений по сравнению с кодами ошибок*. Обычно такое снижение производительности становится заметным при частой генерации исключений во внутреннем цикле; следует напомнить, что частая генерация исключений обычно говорит о том, что в качестве ошибки рассматривается ситуация, которая ошибкой не является.

В некоторых очень редких случаях некоторые программы, работающие в реальном времени, могут собираться с отключенной обработкой исключений из-за того, что механизм этой обработки, предлагаемый компилятором, имеет в худшем случае время работы, которое не позволяет ключевым операциям выполняться в реальном времени. Конечно, такое отключение обработки исключений означает, что язык и стандартная библиотека не будут уведомлять об ошибках стандартным способом (или не будут уведомлять об ошибках вообще — см. документацию на ваш компилятор), и механизм уведомления об ошибках в вашем проекте должен быть основан не на исключениях, а на кодах ошибок. Трудно преувеличить всю нежелательность принятия такого решения. Перед тем как решиться на такой шаг, вы должны детально проанализировать, каким образом будет выполняться уведомление об ошибках в конструкторах и операторах и как предложенная схема будет работать на вашем компиляторе. Если после серьезного и глубокого анализа вы все же решите отключить механизм обработки исключений, то не делайте это во всем проекте; постарайтесь ограничиться как можно меньшим количеством модулей, собрав в них наиболее важные и чувствительные ко времени выполнения операции.

Ссылки

[Alexandrescu03d] • [Allison98] §13 • [Stroustrup94] §16 • [Stroustrup00] §8.3.3, §14.1, §14.4-5, §14.9, §E.3.5 • [Sutter00] §8-19, §40-41, §47 • [Sutter02] §17-23 • [Sutter04] §11-16 • [Sutter04b]

73. Генерируйте исключения по значению, перехватывайте — по ссылке

Резюме

Генерируйте исключения по значению (не через указатель) и перехватывайте их как ссылки (обычно константные). Эта комбинация наилучшим образом соответствует семантике исключений. При повторной генерации перехваченного исключения предпочтительно использовать просто оператор `throw;`, а не инструкцию `throw e;`.

Обсуждение

При генерации исключения генерируйте его по значению. Избегайте использовать исключение-указатель, поскольку в этом случае вам придется столкнуться с массой вопросов управления памятью: вы не можете генерировать исключение-указатель на значение в стеке, поскольку до того, как указатель достигнет точки назначения, стек будет свернут. Вы можете генерировать исключение-указатель на динамически выделенную память (если, конечно, ошибка, о которой вы сообщаете, не состоит в нехватке памяти), но при этом вы возлагаете на `catch`-блок задачу по освобождению выделенной памяти. Если вы уверены, что вам надо генерировать именно указатель, подумайте, нельзя ли заменить его интеллектуальным указателем типа `shared_ptr<T>` вместо обычного `T*`.

При генерации по значению компилятор сам отвечает за запутанный процесс управления памятью, выделенной генерируемому объекту. Все, что требуется от вас, — это принять меры для гарантии того, что копирующий конструктор класса вашего исключения не может генерировать исключений (см. рекомендацию 32).

Если только вы не генерируете интеллектуальный указатель, который добавляет уровень косвенности для сохранения полиморфизма, перехватывать исключения следует по ссылке. Перехват по значению приведет к срезке в точке перехвата (см. рекомендацию 54), что лишит вас обычно весьма важного качества полиморфизма объекта исключения. Перехват по ссылке сохраняет полиморфизм объекта исключения.

При повторной генерации исключения `e` лучше просто писать оператор `throw;` вместо инструкции `throw e;`. Дело в том, что первый способ всегда сохраняет полиморфизм объекта повторно генерируемого исключения.

Примеры

Пример. Повторная генерация измененного исключения. При повторной генерации перехваченного исключения предпочтительно использовать простой оператор `throw;`.

```
catch( MyException& e ) {      // Перехват неконстантной ссылки
  e.AppendContext("Перехват");// Внесение изменения
  throw;                       // Повторная генерация
                               // модифицированного исключения
}
```

Ссылки

[Dewhurst03] §64-65 • [Meyers96] §13 • [Stroustrup00] §14.3 • [Vandevoorde03] §20

74. Уведомляйте об ошибках, обрабатывайте и преобразовывайте их там, где следует

Резюме

Сообщайте об ошибках в тот момент, когда они обнаружены и идентифицированы как ошибки. Обрабатывайте или преобразовывайте их на самом нижнем уровне, на котором это можно сделать корректно.

Обсуждение

Сообщайте об ошибке (т.е. пишите `throw`) там, где функция обнаруживает ошибку, которую не может разрешить самостоятельно и которая делает невозможным продолжение выполнения функции (см. рекомендацию 70).

Обрабатывайте ошибку (т.е. пишите `catch`, который не генерирует повторно то же или иное исключение и не использует иной способ для дальнейшего уведомления об ошибке, например, код ошибки) там, где вы обладаете достаточной информацией, чтобы ее обработать, в том числе для обеспечения границ, определенных стратегией обработки ошибок (например, границ функции `main` или потоков; см. рекомендацию 62) и для поглощения исключений в телах деструкторов и функций освобождения ресурсов.

Преобразовывайте ошибку (т.е. пишите `catch`, который будет генерировать иное исключение или использовать иной способ для дальнейшего уведомления об ошибке, например, код ошибки) в следующих обстоятельствах.

- Для добавления высокоуровневого семантического значения. Например, в текстовом редакторе функция `Document::Open` может принимать низкоуровневую ошибку “неожиданное окончание файла” и преобразовывать ее в ошибку “неверный или поврежденный документ”, добавляя соответствующую семантическую информацию.
- *Для изменения механизма обработки ошибок*. Например, в модуле, который внутренне использует исключения, но чей API в стиле C сообщает об ошибках посредством кодов ошибок, функции API должны перехватывать исключения и возвращать документированные коды ошибки, понятные вызывающему коду.

Код не должен перехватывать ошибку, если контекст не позволяет ему сделать с ней что-либо полезное. Если функция не может самостоятельно обработать ошибку (возможно, преобразовать или сознательно поглотить ее), то она не должна мешать ошибке распространяться далее, чтобы достичь вызывающего кода, который сможет ее обработать.

Исключения

Иногда оказывается полезным перехватить и повторно генерировать ту же ошибку (т.е. воспользоваться `catch` с последующим `throw`) для контроля за ошибками, несмотря на то, что в действительности обработки ошибки при этом не происходит.

Ссылки

[Stroustrup00] §3.7.2, §14.7, §14.9 • [Sutter00] §8 • [Sutter04] §11 • [Sutter04b]

75. Избегайте спецификаций исключений

Резюме

Не пишите спецификаций исключений у ваших функций, если только вас не заставляют это делать внешние обстоятельства (например, код, который вы не можете изменить, уже ввел их; см. исключения к данному разделу).

Обсуждение

Если говорить коротко — не беспокойтесь о спецификациях исключений. Основная проблема заключается в том, что спецификации исключений всего лишь "якобы" являются составной частью системы типов. Они не делают того, что от них ожидает большинство программистов, и вам почти никогда не надо то, что они на самом деле делают.

Спецификации исключений не являются частью типа функции за исключением тех случаев, когда они таки ею являются. Это не шутка — они по сути образуют "теневую" систему типов, из-за чего написание спецификаций исключений в разных местах программы оказывается:

- *неверным* — в инструкции `typedef` для указателя на функцию;
- *разрешенным* — в точно таком же коде, но без использования `typedef`;
- *необходимым* — в объявлении виртуальной функции, которая перекрывает виртуальную функцию базового класса со спецификацией исключений;
- *неявным и автоматическим* — в объявлении конструкторов, операторов присваивания, операторов присваивания и деструкторов, когда они неявно генерируются компилятором.

Весьма распространенным, но тем не менее не верным является убеждение о том, что спецификации исключений статически гарантируют, что функции будут генерировать только перечисленные исключения (возможно, не будут генерировать исключения вообще), что позволяет компилятору выполнить оптимизацию на основе данной гарантии.

В действительности же спецификации исключений делают нечто вроде бы и незначительное, но фундаментально отличающееся: они заставляют компилятор ввести в программу дополнительный код в форме неявных блоков `try/catch` вокруг тела функции для обеспечения проверки времени выполнения того, что функция действительно генерирует только перечисленные исключения или не генерирует их вообще, кроме тех случаев, когда компилятор может статически доказать, что спецификация исключений никогда не будет нарушена. В последнем случае компилятор может оптимизировать код, убрав описанную проверку. Спецификации исключений, кроме того, могут помешать дальнейшей оптимизации кода компилятором — так, например, некоторые компиляторы не могут делать встраиваемыми функции, имеющие спецификации исключений.

Однако хуже всего то, что спецификации исключений — очень "тупой" инструмент: при нарушении они по умолчанию немедленно прекращают выполнение программы. Вы можете зарегистрировать функцию-обработчик данного нарушения, но этот вряд ли вам поможет, поскольку такой обработчик только один на всю программу, и все, что он может сделать для того, чтобы избежать немедленного завершения программы, — это сгенерировать допустимое исключение. Но, напомним, такой обработчик — единственный для всего приложения, так что трудно представить, как можно сделать что-то полезное в такой ситуации или как узнать, какое исключение можно повторно сгенерировать в нем без вмешательства во все спецификации исключений (например, определить, что все спецификации исключений должны

включать некоторое общее исключение `UnknownException`, генерируемое обработчиком. Заметим, что это автоматически уничтожит все преимущества от использования спецификаций исключений).

Обычно вы не можете писать спецификации исключений для шаблонов функций, поскольку вы не можете в общем случае сказать заранее, какие типы исключений могут сгенерировать типы, с которыми будет работать шаблон.

Снижение производительности в обмен на введение ограничений практически всегда бесполезно. Это отличный пример преждевременной пессимизации (см. рекомендацию 9).

Нет простого пути решения описанных в данной рекомендации проблем. В частности, проблемы не решаются путем перехода к статическим проверкам. Программисты часто предлагают перейти от динамической проверки спецификаций исключений к статической, как это делается в Java и других языках программирования. Коротко говоря, это просто означает поменять шило на мыло, т.е. один ряд проблем на другой. Пользователи языков со статической проверкой спецификаций исключений не менее часто предлагают перейти к динамической проверке…

Исключения

Если вы перекрываете виртуальную функцию базового класса, которая уже имеет спецификацию исключений, и у вас нет возможности внести изменения в базовый класс и убрать спецификации исключений (или убедить сделать это автора класса), то вы должны использовать совместимую спецификацию исключений в вашей перекрывающей функции, причем следует сделать ее не менее ограничивающей, чем в базовой версии, чтобы минимизировать возможность нарушений спецификации исключений:

```
class Base { // ...     // В чужом классе имеется
    virtual f()         // спецификация исключений,
        throw(X, Y, Z); // и если вы не можете
};                      // ее удалить...

class MyDerived
: public Base { // ...  // ... то в вашем классе при
    virtual f()         // перегрузке функции она должна
        throw(X, Y, Z); // иметь совместимую (желательно
};                      // идентичную) спецификацию исключений
```

Из опыта [BoostLRG] следует, что только пустые спецификации исключений (т.е. `throw()`) у невстраиваемых функций “могут давать некоторое преимущество у некоторых компиляторов”. Не слишком оптимистичное заявление для одного из наиболее продвинутых, разрабатываемого экспертами мирового уровня проекта…

Ссылки

[BoostLRG] • [Stroustrup00] §14.1, §14.6 • [Sutter04] §13