Контрольные вопросы

31 октября 2022 г.

1 Какие существуют способы обработки различных ошибок?

Существуют следующие способы обработки ошибок:

- Обработка ошибки "на месте"(in place), т.е. там где она возникла. Если это возможно, то данная стратегия является наилучшей, поскольку ошибку можно локализовать и исправить, не затрагивая какой-либо внешний код. В таком случае можно использовать два варианта: повторение попытки до успешного завершения (например, проверка наличия доступа к Интернету – когда подключение к Сети исчезает, возникает ошибка, но затем оно может восстановиться – тогда выполнение кода может быть продолжено) или отмена выполняемой операции. Реализации данных возможностей осущетсвляются с помощью условной конструкции if–then–else.

  Другой вариант обработки ошибки in place – остановление выполнения программы. Он используется, когда ошибка настолько серьёзна, что программа не может продолжать работать должным образом (non-recoverable/fatal error). В таких случаях лучше всего завершать программу (аварийно), причём:

  – если код находится в main() или функция вызывается непосредственно из main(), лучше всего позволить main() вернуть ненулевой код состояния.
  – в противном случае целесообразно использование halt statement (abort–exit–terminate).

  Наконец, ещё один вариант – использование asserts – выражений, которые будут истинными, если в программе нет ошибок. Если выражение оценивается как true, оператор утверждения assert ничего не делает. Если условное выражение оценивается как false, отображается сообщение об ошибке и программа завершается (через std::abort). Это сообщение об ошибке обычно содержит выражение, которое не удалось выполнить, в виде текста, а также имя файла кода и номер строки утверждения. Это позволяет очень легко сказать не только о том, в чем была проблема, но и о том, где она возникла в коде.

- Передача информации об ошибке по вызову для её устранения/обработки. Тогда в дальнейшем коде можно обнаружить ошибку и исправить/обработать её. Данный способ может быть реализован различными вариантами:

  – Использование кодов возврата, когда какой-либо ошибке соответствует некоторый код.
  – Проброс исключения, позволяющий отделить обработку ошибок или других исключительных обстоятельств от типичного потока управления кода.
  – И другие...

2 В чём заключаются недостатки механизма кодов возврата?

Недостатки механизма кодов возврата:

- Не всегда очевидно, является ли возвращённое значение допустимым (т.е. оно могло быть получено в результате выполнения функции) или нет.
- Функции могут возвращать лишь одно значение, поэтому возникают трудности с возвращением значения и кода возврата одновременно. Поэтому либо результат, либо код возврата должны быть переданы обратно в качестве ссылочного параметра, что делает код менее удобным в использовании.
- Если в коде может случаться много ошибок, необходимо постоянно проверять коды возврата.
- Коды возврата не сочетаются с конструкторами и операторами.
- Когда вызывающей стороне возвращается код ошибки, вызывающая сторона не всегда может справиться с ней. Если вызывающая сторона не хочет обрабатывать ошибку, она должна либо проигнорировать её (в этом случае ошибка будет потеряна навсегда), либо вернуть ошибку вверх по стеку в функцию, которая её вызвала. Это может привести ко многим из проблем, отмеченных выше.

Итак, основная проблема с механизма кодов возврата заключается в том, что код обработки ошибок оказывается неразрывно связанным с обычным потоком управления кода. Это ограничивает как компоновку кода, так и разумную обработку ошибок.

3 Какими особенностями обладает механизм исключений?

Механизм исключений – механизм, предназначенный не только для обработки ошибок, но и для обработки особых ситуаций. Особенности механизма исключений:

- Многоуровневость и модульность. При возникновении исключения посредством throw, выполнение программы немедленно переходит к ближайшему охватывающему блоку try (распространяясь вверх по стеку, если необходимо найти охватывающий блок try). Тогда:

  – Если какой-либо из обработчиков catch, прикрепленных к блоку try, обрабатывает исключение этого типа, этот обработчик выполняется, и исключение считается обработанным.

  – Если подходящих обработчиков catch не существует, выполнение программы распространяется на следующий охватывающий блок try. Если до конца программы не удается найти подходящие обработчики перехвата, программа завершится с ошибкой исключения: вызывается std::terminate(), и приложение завершается. В таких случаях стек вызовов может быть раскручен, а может и нет. Если стек не раскручен, локальные переменные не будут уничтожены, и никакой очистки, ожидаемой при уничтожении указанных переменных, не произойдёт.

  Таким образом, блоки try перехватывают исключения не только из инструкций внутри блока try, но и из функций, вызываемых в блоке try.

- Эффективность. Работа с исключениями увеличивает размер исполняемого файла, а также может замедлить его работу из-за дополнительных проверок. Однако основная потеря производительности для исключений происходит, когда исключение действительно выдается. В этом случае стек должен быть раскручен и найден соответствующий обработчик исключений, что является относительно дорогостоящей операцией. Поэтому механизм исключений используется, если нет хороших альтернативных способов.

- Преобразования. Неявные преобразования при сопоставлении исключений с блоками захвата не выполняются компилятором (например, исключение char не будет соответствовать блоку захвата int). Но приведение из производного класса к одному из его родительских классов будет выполнено (что используется при реализации классов исключений).

- Функции потенциально могут генерировать исключения любого типа данных.

- Существование catch-all handler. Этот обработчик позволяет обеспечить упорядоченное поведение при возникновении необработанного исключения.

- Возможность повторного создания исключений.

- Существование блоков проверки функций (function try blocks) предназначенных для того, чтобы обработчик исключений мог быть установлен вокруг тела всей функции, а не вокруг блока кода.

4 Для чего используются спецификатор и оператор noexcept?

В C++ все функции классифицируются как non-throwing (не генерируют исключения) или potentially throwing (могут генерировать исключение).

Спецификатор noexcept определяет функцию как non-throwing. Ключевое слово noexcept используется в объявлении функции. Спецификатор noexcept не запрещает функции генерировать исключения или вызывать другие функции, которые потенциально могут генерировать исключения. Скорее, когда возникает исключение, если оно выходит из noexcept функции, будет вызвана std::terminate. При этом раскручивание стека может произойти или не произойти, т.е. созданные объекты могут быть или не быть уничтожены должным образом до завершения программы. Функции, отличающиеся только своей спецификацией исключений, не могут быть перегружены.

Оператор noexcept принимает выражение в качестве аргумента и возвращает значение true либо false в зависимости от предположения компилятора, выдаст ли выражение исключение или нет. Оператор noexcept можно использовать внутри функций. Он проверяется статически во время компиляции и фактически не оценивает входное выражение. Оператор noexcept можно использовать для условного выполнения кода в зависимости от того, является ли он potentially throwing или нет. Это требуется для выполнения определенных гарантий безопасности исключений (см. вопрос [5]).

Итак, следует создавать noexcept конструкторы и перегруженные операторы присваивания (если они non-throwing) для использования оптимизаций компилятора. Кроме того, следует использовать noexcept для других функций, если требуется выразить гарантию их успешного выполения или отсутствия срабатывания механизма исключений (например, появится необходимость использовать функцию в деструкторе – она должна быть noexcept, поскольку деструктор должен быть non-throwing).

5 Как формулируются гарантии безопасности исключений?

Гарантия безопасности исключений – это договорное руководство о том, как функции или классы будут вести себя в случае возникновения исключения. Существует четыре уровня безопасности исключений:

- Нет гарантии. Нет никаких гарантий относительно того, что произойдёт, если возникнет исключение (например, класс может остаться в непригодном для использования состоянии).
- Базовая гарантия. Если возникнет исключение, утечка памяти не произойдет, и объект по-прежнему можно будет использовать (инварианты не будут нарушены), но программа может остаться в изменённом состоянии. Фактически базовая гарантия соответствует идиоме RAII.
- Строгая (надёжная) гарантия. Если возникнет исключение, утечка памяти не произойдёт, а состояние программы не изменится. Это означает, что функция должна либо полностью завершиться успешно, либо не иметь побочных эффектов в случае сбоя. Это легко сделать, если сбой произойдет до того, как что-либо будет изменено, но также может быть достигнуто путем отката любых изменений, чтобы программа вернулась в состояние, предшествующее сбою.
- Отсутствие исключений/без сбоя. Функция всегда будет выполнена успешно (без сбоев) или с ошибкой, но без создания исключения. Иными словами, исключения никогда не будут генерироваться.

Литература

[1] https://www.learncpp.com/