Программирование

Часть 8 Обработка исключений

6. Исключения

Обработка исключений - средства языка программирования, предназначенные для описания реакции программы на нестандартные ситуации (ошибки времени выполнения и другие возможные проблемы), возникающие при исполнении программы и приводящие к невозможности или нецелесообразности дальнейшей работы в соответствии с основным алгоритмом программы. Такие нестандартные ситуации называют исключениями.

Примеры исключений:

Нулевое значение знаменателя при выполнении операции деления. Результат орперации не определен, поэтому его дальнейшее использование в программе невозможно.

Ошибка при считывании данных с внешнего устройства. Если данные невозможно ввести, любые дальнейшие операции с ними бессмысленны.

Запрос на ввод данных (например, нажатие клавиши на клавиатуре). Программа должна перейти к чтению данных, чтобы не потерять поступающую информацию.

6. Исключения

В отсутствие собственного механизма обработки исключений для прикладных программ наиболее общей реакцией на любую исключительную ситуацию является немедленное прекращение выполнения с выдачей пользователю сообщения о характере исключения.

Альтернативной возможностью является игнорирование исключительной ситуации и продолжение работы, но такая тактика опасна, так как может привести к ошибочным результатам работы программы.

Обработка исключительных ситуаций самой программой заключается в том, что при возникновении исключительной ситуации, управление передаётся некоторому заранее определённому обработчику — блоку кода, процедуре, функции, которые выполняют необходимые действия.

.

6. Исключения

Исключительные ситуации, возникающие при работе программы, можно разделить на две группы: синхронные и асинхронные

Синхронные исключения могут возникнуть только в определённых, заранее известных точках программы. Ошибка деления на нуль, ошибка чтения файла — типичные синхронные исключения, так как возникают они только при выполнении соответствующих операций (соответственно, деления, чтения из файла и т.п.).

Асинхронные исключения могут возникать в любой момент времени и не зависят от того, какая конкретно инструкция программы выполняется Типичные примеры таких исключений: аварийный отказ питания или поступление новых данных.

6. Исключения

Существует два принципиально разных механизма функционирования обработчиков исключений:

Обработка с возвратом подразумевает, что обработчик исключения ликвидирует возникшую проблему и приводит программу в состояние, когда она может работать дальше по основному алгоритму. В этом случае после того, как выполнится код обработчика, управление передаётся обратно в ту точку программы, где возникла исключительная ситуация (либо на команду, вызвавшую исключение, либо на следующую за ней) и выполнение программы продолжается. Обработка с возвратом типична для обработчиков асинхронных исключений.

Обработка без возврата заключается в том, что после выполнения кода обработчика исключения управление передаётся в некоторое, заранее заданное место программы, и с него продолжается исполнение.

.

6. Исключения

Существует два варианта обработки исключений: структурная и неструктурная обработка.

Неструктурная обработка исключений реализуется путем регистрации функций или инструкций-обработчиков для каждого возможного типа исключения. Язык предоставляет программисту возможности регистрации обработчика и его разрегистрации. Регистрация «привязывает» обработчик к определённому исключению, разрегистрация — отменяет эту «привязку». Если исключение происходит, выполнение основного кода программы немедленно прерывается и начинается выполнение обработчика. По завершении обработчика управление передаётся либо в некоторую наперёд заданную точку программы, либо обратно в точку возникновения исключения (в зависимости от заданного способа обработки — с возвратом или без). Независимо от того, какая часть программы в данный момент выполняется, на определённое исключение всегда реагирует последний зарегистрированный для него обработчик. В некоторых языках зарегистрированный обработчик сохраняет силу только в пределах текущего блока кода (процедуры, функции), тогда процедура разрегистрации не требуется.

6. Исключения

Конструкция структурной обработки исключений содержит блок контролируемого кода и обработчик (обработчики) исключений. Такая конструкция имеет следующий вид:

```
НачалоБлока
  // Контролируемый код
  ...
  если (условие) то СоздатьИсключение Исключение2
  ...
  Обработчик Исключение1
  // Код обработчика для Исключения1
  ...
  Обработчик Исключение2
  // Код обработчика для Исключения2
  ...
  ОбработчикНеобработанныхИсключений
  // Код обработки ранее не обработанных исключений
  ...
КонецБлока
```

Блоки обработки исключений могут многократно входить друг в друга, как явно (текстуально), так и неявно (например, в блоке вызывается

6. Исключения

Блоки обработки исключений могут многократно входить друг в друга, как явно (текстуально), так и неявно (например, в блоке вызывается процедура, которая сама имеет блок обработки исключений).

Если ни один из обработчиков в текущем блоке не может обработать исключение, выполнение данного блока немедленно завершается, и управление передаётся на ближайший подходящий обработчик более высокого уровня иерархии. Это продолжается до тех пор, пока обработчик не найдётся и не обработает исключение.

6. Исключения

Блоки обработки исключений могут многократно входить друг в друга, как явно (текстуально), так и неявно (например, в блоке вызывается процедура, которая сама имеет блок обработки исключений).

Если ни один из обработчиков в текущем блоке не может обработать исключение, выполнение данного блока немедленно завершается, и управление передаётся на ближайший подходящий обработчик более высокого уровня иерархии. Это продолжается до тех пор, пока обработчик не найдётся и не обработает исключение.

6. Исключения

Язык C++ включает следующие возможности для работы с исключениями: создание защищенных блоков (try-блок) и перехват исключений (catch-блок); инициализация исключений (оператор throw).

Для сигнализации о возникновении исключительной ситуации программист может воспользоваться невнятным механизмом классификации исключений по типам, который предложен в стандарте языка, возбудить стандартное исключение (например, с помощью класса exception из стандартной библиотеки) или описать собственный тип исключения.

Язык C++ построен на доверии к программисту, и требует, чтобы ситуации, связанные, например, с отсутствием входного файла, обрабатывались самим программистом. Так, выполнение процедуры reset в Паскале для несуществующего файла приведет к ошибке выполнения, а в C++ открытие файлового потока с атрибутами ios::in | ios::nocreate будет выполнено без каких-либо дополнительных сообщений.

6. Исключения

Однако ситуации, связанные с делением на ноль или с обращением к неправильному адресу памяти, однозначно классифицируются как ошибки выполнения и приводят к аварийному завершению программы.

Для перехвата таких ошибок может быть использован простейший формат защищенного блока, который имеет вид

try {*операторызащищенногоблока*}
catch(…) {*обработчикошибочнойситуации*}
Рассмотрим пример перехвата такой ошибочной ситуации.

```
float k;
int i, j;
…
try {
 i=j/(int)k;
}
catch(…) {
 cout << “Ошибка (деление на ноль)” << endl;
}
```

6. Исключения

Однако ситуации, связанные с делением на ноль или с обращением к неправильному адресу памяти, однозначно классифицируются как ошибки выполнения и приводят к аварийному завершению программы.

Для перехвата таких ошибок может быть использован простейший формат защищенного блока, который имеет вид

try {*операторызащищенногоблока*}
catch(…) {*обработчикошибочнойситуации*}
Рассмотрим пример перехвата такой ошибочной ситуации.

```
float k;
int i, j;
…
try {
 i=j/(int)k;
}
catch(…) {
 cout << “Ошибка (деление на ноль)” << endl;
}
```

6. Исключения

Однако ситуации, связанные с делением на ноль или с обращением к неправильному адресу памяти, однозначно классифицируются как ошибки выполнения и приводят к аварийному завершению программы.

Для перехвата таких ошибок может быть использован простейший формат защищенного блока, который имеет вид

try {*операторызащищенногоблока*}
catch(…) {*обработчикошибочнойситуации*}
Рассмотрим пример перехвата такой ошибочной ситуации.

```
float k;
int i, j;
…
try {
 i=j/(int)k;
}
catch(…) {
 cout << “Ошибка (деление на ноль)” << endl;
}
```

6. Исключения

Для возбуждения **собственных исключений** используется оператор
throw ***[выражение]***

Тип выражения, указанного в операторе throw, определяет тип исключительной ситуации, а значение может быть передано в блок обработки исключительных ситуаций. Этот механизм, заявленный как стандартный, представляется весьма экзотическим без использования механизма классов. Лишь использование стандартных классов-исключений или разработка собственных классов позволяют в полной мере оценить все возможности такого подхода.

6. Исключения

Полный формат защищенного блока имеет вид
try {*операторызащищенногоблока}*
{catch-блоки}…

Catch-блок имеет один из следующих форматов:
catch (*тип) {обработчикошибочнойситуации}*
catch (*тип идентификатор) {обработчикошибочнойситуации}*
catch (…) {*обработчикошибочнойситуации}*

Первый формат используется, если нам надо указать тип перехватываемого исключения, но не нужно обрабатывать связанное с этим исключением значение (это достигается при использовании второго формата оператора catch).

Наконец, третий формат оператора catch позволяет обработать все исключения (в том числе и ошибки выполнения, что было описано в предыдущем пункте).

6. Исключения

Обработка исключений, возбужденных оператором throw, идет по следующей схеме:
1. Создается статическая переменная со значением, заданным в операторе throw. Она будет существовать до тех пор, пока исключение не будет обработано. Если переменная-исключение является объектом класса, при ее создании работает конструктор копирования.
2. Завершается выполнение защищенного try-блока: раскручивается стек подпрограмм, вызываются деструкторы для тех объектов, время жизни которых истекает и т.д.
3. Выполняется поиск первого из catch-блоков, который пригоден для обработки созданного исключения. Поиск ведется по следующим критериям:
если тип, указанный в catch-блоке, совпадает с типом созданного исключения, или является ссылкой на этот тип;
класс, заданный в catch-блоке, является предком класса, заданного в throw, и наследование выполнялось с ключом доступа public;
указатель, заданный в операторе throw, может быть преобразован по стандартным правилам к указателю, заданному в catch-блоке.
в операторе throw задано многоточие.

6. Исключения

Если нужный обработчик найден, то ему передается управление и, при необходимости, значение оператора throw. Оставшиеся catch-блоки, относящиеся к защищенному блоку, в котором было создано исключение, игнорируются.
Из указанных правил поиска следует, что очень важен порядок расположения catch-блоков. Так, блок catch(…) должен стоять последним в списке, а блок catch (void *) – после всех блоков с указательными типами.

Если ни один из catch-блоков, указанных после защищенного блока, не сработал, по таким же правилам проводится выход из внешнего блока try (если он, конечно, есть!).

В конце оператора catch может стоять оператор throw без параметров. В этом случае работа catch-блока считается незавершенной, и происходит поиск соответствующего обработчика на более высоких уровнях.

6. Исключения

Пример:

```
try {
 …
 try {
  …
  throw "Error!";
  …
 } //внутренний try
 catch (int) {…
 }
 catch (float) {…
 }
 …
} //внешний try
catch (char * c) {…
}
catch (...) { …
}
```

6. Исключения

Пример:

```
try {
 …
 try {
  …
  throw "Error!";
  …
 } //внутренний try
 catch (char *) {…
 }
 catch (float) {…
 }
 …
} //внешний try
catch (char * c) {…
}
catch (...) { …
}
```

6. Исключения

Пример:

```
try {
 …
 try {
  …
  throw "Error!";
  …
 } //внутренний try
 catch (char *) {…
 }
 catch (float) {…
 }
 …
} //внешний try
catch (char * c) {…
}
catch (...) { …
}
```

6. Исключения

```
class TMyException {
   char* msg;
   void operator = (const TMyException& e) {}
public:
      TMyException(const char* str) {
      assert(str != NULL);
      msg = new char[strlen(str) + 1];
      strcpy(msg, str);
   }

   TMyException(const TMyException& e) {
      msg = new char [strlen(e.msg) + 1];
      strcpy(msg, e.msg);
   }

   const char* getMessage() const { return msg; }

   ~TMyException() { delete [] msg; }
};
```

6. Исключения

Перегруженная операция присваивания в описании, защищенном спецификатором доступа private, блокирует выполнение этой операции.
Аналогично работает вызов функции assert (эта функция аварийно завершает работу программы, если выражение, заданное в параметре, ложно).
Написание конструктора копирования обязательно – ведь он может неявно вызываться в блоке catch!
Для генерации исключения из описанного класса достаточно записать инструкцию

throw TMyException("Extra data in input file");

Обработчик соответствующей ситуации может выглядеть так:

```
try {
// здесь может быть сгенерировано собственное исключение
}
catch (TMyException e) {
 cout << "ERROR! " << e.getMessage() << endl;
}
```