Обработка исключений

М. А. Ложников

3 октября 2019 г.
Ревизия: 1

Это предварительная невыверенная версия. Читайте на свой страх и риск.

1 Обработка ошибок в языке C

Для обработки ошибок в языке C как правило используются возвращаемые значения функций или внешние переменные. Таким образом, программист каждый раз должен проверять возвращаемое значение каждой функции, которая может отработать с ошибкой.

```
1  int ReadArray(FILE* fin, double* a, int n) {
2    int i;
3
4    for (i = 0; i < n; i++) {
5      if (fscanf(fin, "%lf", a + i) != 1)
6        return -1;
7    }
8
9    return 0;
10 }
```

Предположим, что функция `ReadArray()` вызывается в функции `foo()`, которая в свою очередь вызывается в функции `bar()`. В таком случае одну и ту же ошибку, связанную с чтением данных приходится проверять во всём стеке вызываемых функций.

```
1  int foo() {
2    if (ReadArray(...) < 0)
3      return -1;
4
5    return 0;
6  }
7
8  int bar() {
9    if (foo() < 0)
10     return -1;
11
12   return 0;
13 }
```

Для того, чтобы избежать лишних проверок вручную, в языке C++ существует механизм исключений.

2 Обработка ошибок при помощи исключений в C++

Исключение является объектом некоторого типа данных (тип данных может быть любым), который содержит в себе информацию об ошибке. Само исключение может быть сгенерировано (ещё говорят выброшено) функцией в результате какой-либо ошибки. Затем это исключение можно поймать в специальном обработчике и обработать ошибку должным образом. При генерации исключения выполнение программы сразу перемещается в соответствующий обработчик. При этом компилятор вызывает деструктор для всех объектов, которые пропали из области видимости при перемещении в обработчик. Если для исключения не назначен обработчик, то в случае его срабатывания программа вылетает.

Для генерации исключения используется ключевое слово `throw`, за которым следует сам объект (переменная), содержащий информацию об ошибке. При этом для того, чтобы исключение можно было перехватить, код, который потенциально может сгенерировать исключение должен находиться в специальном блоке `try`. Обработчик исключения пишется отдельно для каждого типа данных, объектом которого может являться исключение в специальном блоке `catch`, следующим сразу за соответствующим блоком `try`.

Таким образом, при генерации исключения программа сразу переходит из блока `try` в соответствующий обработчик (блок `catch`), если он есть. После обработки исключения в обработчике программа продолжает своё выполнение с места сразу после блоков `try/catch`.

```
try {
  /* В блоке try пишется код программы, который может потенциально сгенерировать
     исключение. */

  int i = 0;

  std::cin >> i;

  if (i == 0) {
    // Генерируем исключение типа int.
    throw -1;
  } else if (i == 1) {
    // Генерируем исключение типа const char*.
    throw "Pointer to const char exception.";
  } else if (i == 2) {
    // Генерируем исключение типа std::string.
    throw std::string("std::string exception.");
  }
} catch (const ExceptionType1& exception1) {
  /* Обрабатываем исключение типа ExceptionType1. */
} catch (const ExceptionType2& exception2) {
  /* Обрабатываем исключение типа ExceptionType2. */
} catch (const ExceptionTypeN& exceptionN) {
  /* Обрабатываем исключение типа ExceptionTypeN. */
} catch (...) {
  /* Если в круглых скобках после catch указано многоточие, то такой обработчик
     будет обрабатывать любой тип исключения кроме перечисленных ранее. */
```

```
28 }
29
30 /* В случае если исключение не было сгенерировано, то этот код выполняется сразу
31    после блока try. В противном случае этот код выполняется сразу после соответствующего
32    обработчика исключения. */
```

Отметим, что в конструкции try/catch может быть выполнен только один обработчик. Сами обработчики должны располагаться в порядке от частного к общему.

2.1 Пример

```
1  #include <iostream>
2  #include <cmath>
3
4  double Divide(double left, double right) {
5    if (std::fabs(right) < 1e-14) {
6      /* Попытка деления на нуль. Генерируем исключение. */
7      throw "Division by zero.";
8    }
9
10   /* Выполняем операцию деления. */
11   return left / right;
12 }
13
14 int main() {
15   double a, b, c = 0;
16
17   std::cin >> a >> b;
18
19   try {
20     /* Исключение не обязательно нужно генерировать явно в блоке try. Оно может быть
21        сгенерировано какой-либо функцией, вызывающейся в блоке try, и обработано
22        в соответствующем ему блоке catch. */
23     c = Divide(a, b);
24   } catch(const char* e) {
25     /* Словили исключение. Выводим информацию о нём. */
26     std::cout << "Got an exception. Reason: " << e << std::endl;
27   }
28
29   std::cout << "Result: " << c << std::endl;
30   return 0;
31 }
```

3 Создание собственных типов исключений

```
1 #include <iostream>
2 #include <cstdio>
3 #include <stdexcept>
```

```
4
5  /* Специальный тип исключения для обработки ошибок ввода/вывода. */
6  class IOError {
7   private:
8    std::string reason;
9   public:
10   IOError(const std::string& reason) :
11     reason(reason)
12   { }
13
14   const std::string& Reason() const { return reason; }
15 };
16
17 /* Специальный тип исключения для обработки ошибок, связанных с некорректными
18    значениями. */
19 class ValueError {
20  private:
21   std::string reason;
22  public:
23   ValueError(const std::string& reason) :
24     reason(reason)
25   { }
26
27   const std::string& Reason() const { return reason; }
28 };
29
30 /* А это класс для чтения данных из файла. Класс в конструкторе открывает файл,
31    а в деструкторе его самостоятельно закрывает. */
32 class FileReader {
33  private:
34   std::FILE* file;
35  public:
36   FileReader(const char* filename) :
37     file(std::fopen(filename, "rt"))
38   {
39     if (!file) {
40       /* Не смогли открыть файл. Выбрасываем исключение. */
41       throw IOError("Can't open file");
42     }
43   }
44
45   ~FileReader() {
46     if (file)
47       std::fclose(file);
48   }
49
50   double ReadDouble() const {
51     double value;
52
53     if (std::fscanf(file, "%lf", &value) != 1) {
54       /* Не получилось считать значение. Генерируем исключение. */
```

```
55        throw IOError("Can't read the next value.");
56      }
57
58      return value;
59    }
60
61    double ReadInt() const {
62      int value;
63
64      if (std::fscanf(file, "%d", &value) != 1) {
65        /* Не получилось считать значение. Генерируем исключение. */
66        throw IOError("Can't read the next value.");
67      }
68
69      return value;
70    }
71  };
72
73  void ReadMatrix(const FileReader& reader, double* matrix, int numRows, int numCols) {
74    for (int row = 0; row < numRows; row++) {
75      for (int col = 0; col < numCols; col++) {
76        /* Метод ReadDouble() может сгенерировать исключение IOError. */
77        matrix[row * numCols + col] = reader.ReadDouble();
78      }
79    }
80  }
81
82  int main() {
83    double* matrix = nullptr;
84
85    try {
86      /* Конструктор может сгенерировать исключение IOError. */
87      FileReader reader("input.txt");
88
89      /* Метод ReadInt() может сгенерировать исключение IOError. */
90      int numRows = reader.ReadInt();
91      int numCols = reader.ReadInt();
92
93      if (numRows <= 0 || numCols <= 0) {
94        throw ValueError("Incorrect matrix dimensions.");
95      }
96
97      /* Оператор new может сгенерировать исключение std::bad_alloc. */
98      matrix = new double[numRows * numCols];
99
100     /* Функция ReadMatrix() может сгенерировать исключение IOError. */
101     ReadMatrix(reader, matrix, numRows, numCols);
102   } catch (const IOError& e) {
103     std::cout << "Input/output error. Reason: " << e.Reason() << std::endl;
104
105   } catch (const ValueError& e) {
```

```
106     std::cout << "Value error. Reason: " << e.Reason() << std::endl;
107   } catch (const std::bad_alloc& e) {
108     /* Исключение типа std::bad_alloc из заголовочного файла <stdexcept> генерирует
109        оператор new в случае ошибки выделения памяти. */
110     std::cout << "Can't allocate memory! Reason: " << e.what() << std::endl;
111   }
112
113   if (matrix)
114     delete[] matrix;
115   return 0;
116 }
```

Замечание 3.1 *В примере выше возвращаемое значение оператора* `new` *не проверяется, поскольку в случае ошибки выделения памяти оператор* `new` *генерирует исключение* `std::bad_alloc` *из заголовочного файла* `<stdexcept>`*. Таким образом, оператор* `new` *в C++ никогда не возвращает значение* `nullptr`*.*

3.1 Проброс исключения вверх по стеку

В некоторых случаях возникает необходимость поймать исключение в обработчике, а затем передать это исключение дальше в следующий обработчик (если он есть). Речь идёт о ситуации, когда в некотором внешнем блоке `try` написана группа внутренних блоков `try/catch`. Для проброса исключения из внутреннего обработчика во внешний нужно написать конструкцию `throw;` (без аргумента) в соответствующем внутреннем обработчике.

```
1  try {
2    try {
3      int i = 0;
4
5      std::cin >> i;
6
7      if (i == 0) {
8        throw -101;
9      }
10   } catch (int code) {
11     std::cout << "Got an exception for the first time. Code = " << code << std::endl;
12
13     /* Пробрасываем исключение во внешний обработчик. */
14     throw;
15   }
16 } catch(int code) {
17   /* Ловим исключение второй раз. */
18   std::cout << "The same exception. Again! Code = " << code << std::endl;
19 }
```

3.1.1 Пример проброса исключения

Рассмотрим пример, в котором перехват исключения внутри функции необходим для корректного закрытия файла.

```
#include <iostream>
#include <cstdio>

/* Функция читает целое число из заранее открытого файла. В случае ошибки генерируется
   исключение. */
int DoRead(std::FILE* file) {
  int value;

  if (std::fscanf(file, "%d", &value) != 1) {
    /* Не получилось считать число. Генерируем исключение. */
    throw std::string("Can't read value.");
  }

  return value;
}

int ReadIntFromFile(const char* filename) {
  std::FILE* file = std::fopen(filename, "rt");
  int value = 0;

  if (!file) {
    /* Не открылся файл. Выбрасываем исключение. */
    throw std::string("Can't open file!");
  }

  try {
    /* Функция DoRead() может сгенерировать исключение. Если его не обработать
       внутри текущей функции, то файл останется открыт. */
    value = DoRead(file);
  } catch (const std::string& e) {
    /* Не получилось считать данные из файла. Нужно закрыть файл,
       а затем пробросить исключение вверх по стеку. */
    std::fclose(file);

    /* Пробрасываем исключение вверх по стеку. */
    throw;
  }

  /* Если функция DoRead() выбрасывает исключение, то этот код не выполняется.
     Таким образом, файл мог бы остаться открытым, если бы не был написан правильный
     обработчик. */
  std::fclose(file);

  return value;
}

int main() {
  try {
    int value = ReadIntFromFile("input.txt");

```

```
51      std::cout << "Value: " << value << std::endl;
52    } catch (const std::string& e) {
53      std::cout << "Got an exception. Reason: " << e << std::endl;
54    }
55    return 0;
56  }
```

Замечание 3.2 *Отметим, что в предыдущем примере данная проблема решается при помощи специального класса, который закрывает файл в деструкторе. Напомним, что при генерации исключения, деструктор вызывается у всех объектов, которые пропадают из области видимости при переходе в обработчик. По этой причине (и не только) в современном C++ не рекомендуется пользоваться сырыми указателями, обычными файлами из языка C, а так же всем тем, что требует ручного освобождения памяти. Вместо этого в стандартной библиотеке C++ есть специальные классы-оболочки над динамическим массивом (шаблон* `std::vector`*), над файлом (*`std::ifstream` *и* `std::ofstream`*), а также над обычным указателем (шаблоны* `std::unique_ptr`*,* `std::shared_ptr` *и не только).*

Список литературы

[1] *Бьерн Страуструп* Язык программирования C++. М:Бином. 2011.

[2] `https://en.cppreference.com`

[3] `https://en.cppreference.com/w/cpp/language/throw`

[4] `https://en.cppreference.com/w/cpp/language/try_catch`