Объектно-ориентированное программирование. Лекция 5. Обработка исключений

Макаров П. А.

28 октября 2022 г.

Содержание

1 Ошибочные ситуации и их обработка

При выполнении программы могут возникать проблемы, препятствующие её нормальному функционированию. Это может быть обусловлено как непредусмотрительностью программиста либо действиями пользователя, так и состоянием рабочего окружения программы (отсутствие достаточного объёма оперативной памяти, места на жёстком диске и т. д.).

Рассмотрим, например, исходный текст, приведённый в Листинге 1.

```c
#include <stdio.h>

int main(void) {
    int x, y, z, status;

    printf("Input 3 space-separated integers x, y, z: ");
    status = scanf("%d %d %d", &x, &y, &z);

    printf("Number of correctly processed arguments: %d\n",
       status);
    printf("The variables x, y, z took the values: %d, %d, %d
       \n", x, y, z);

    return 0;
}
```

Листинг 1: Анализ ошибок, возможных при работе функции `scanf`

Для решения возможных проблем, связанных с ошибочными ситуациями, C-программисты обычно предпринимают одно из следующих действий:

- прерывают выполнение программы (здесь уместно изучить страницу мануала Linux-программиста, посвященную функции `exit`, прототип которой содержится в заголовочном файле `stdlib.h`);
- выводят значение, означающее ошибку (см. документацию по глобальной переменной `errno`, а также функциям `strerror`, `perror` и `error`);
- выводят сообщение об ошибке в стандартный поток ошибок `stderr` и так или иначе передают программе некоторое приемлемое значение, которое позволит ей продолжить работу.

В том случае, когда структура программы достаточно сложна, принято возлагать ответственность за принятие решений о возможных действиях при возникновении ошибочной ситуации на головную часть программы. Другими словами, завершать программу, а также выполнять тем или иным способом выдачу диагностических сообщений обычно может только функция `main`, либо вызываемые ей функции, специально предназначенные для этого. Остальная часть программы при возникновении ошибки обычно должна только оповестить об этом головную программу, не предпринимая никаких собственных действий. На практике реализовать данную схему не всегда просто, а в случае большого числа вложенных вызовов функций — очень затруднительно и сопряжено с большой громоздкостью кода.

2 Общая идея механизма обработки исключений

Для решения описанной проблемы в C++ появился ещё один механизм уведомления об ошибках и их обработки, который называется обработкой исключительных ситуаций (либо просто — исключений).

Определение 1. ***Исключение*** *в общем смысле слова — это возникновение аномальных условий, требующих специальной обработки во время выполнения программы. Исключение нарушает нормальный ход выполнения программы и приводит к выполнению предварительно сформулированного обработчика исключений.*

Определение 2. *В более специальном смысле слова,* ***исключение*** *— это особый способ завершения функции, т. е. в языках программирования, поддерживающих исключения, функция может либо вернуть значение, либо возбудить исключение.*

При возбуждении исключения программа переходит в особое состояние, в котором все активные на данный момент функции досрочно завершаются одна за другой, пока не найдётся такая функция, которая может справиться с данным типом исключительных ситуаций.

Замечание 1. *Механизм исключений не является неотъемлемой частью ООП, а скорее относится к функциональному программированию.*

Перечислим преимущества механизма обработки исключений:

- исключения невозможно проигнорировать, т. е. любая исключительная ситуация обязательно будет обработана;
- исключения выделяют обработку ошибок и восстановление после них из основного потока управления, т. е. алгоритм обработки данных описывается отдельно от обработки исключительных ситуаций, которые могут в нем возникнуть;
- при использовании в программах классов и объектов исключения иногда оказываются единственной возможностью обработки ошибок.

Замечание 2. *Никогда не применяйте механизм обработки исключений для штатного выхода из вложенных управляющих конструкций или глубоких цепочек вызовов функций.*

Для обработки исключений в язык C++ введены дополнительные операторы: `try`, `catch` и `throw`. Обработка исключений состоит из нескольких этапов:

1. выделение контролируемого блока, т. е. блока, в котором может возникнуть исключительная ситуация — блок `try`;

2. генерация одного или нескольких исключений с помощью оператора `throw` внутри блока `try` или внутри функций, которые вызываются из этого блока;

3. размещение сразу за блоком `try` одного или нескольких обработчиков исключений `catch`.

Обычно всю обработку ошибок с помощью механизма обработки исключений выполняют в функции `main`, т. к. в подавляющем большинстве случаев она остаётся активной всё время исполнения программы, а значит может обработать исключение, возбуждённое практически в любом месте программы.

3 Возбужение исключений

Оператор

```
1 throw <expression>;
```

Листинг 2: Синтаксис оператора `throw`

возбуждает исключение, тип и значение которого определяются выражением `<expression>`. В простейшем случае `<expression>` имеет один из стандартных типов, таких как целое число `(int)`, C-строка `(const char*)` и т. д., но чаще всего в роли исключения используется объект специального класса.

4 Обработка исключений

С помощью ключевого слова `try` в тексте программы выделяется блок операторов, исполнение которых, предположительно, может привести к исключениям. Для обработки исключений сразу после `try`-блока помещается один или несколько `catch`-блоков.

Рассмотрим пример кода для обработки исключительной ситуации деления на 0 (см. Листинг 3). Исключение будет генерироваться до тех пор, пока пользователем не будут введены корректные данные.

```
while(true) {
    int x, y;
    cout << "Input x and y: " << endl;
    cin >> x >> y;
    try {
        if(y == 0)
            throw 0;
        cout << " x/y = " << (x / y) << endl;
        break;
    }
    catch(int e) {
        cout << "Error! Dividing by 0." << endl;
    }
}
```

Листинг 3: Пример обработки исключения деления на 0

Один контролируемый блок `try` может быть предназначен для обработки нескольких видов исключительных ситуаций. Это может происходить, если в блоке `try` присутствуют вызовы функций, которые могут генерировать исключения разного вида. В этом случае следует предусмотреть обработку каждого из возможных исключений в вызывающей функции. При этом следует учитывать следующие правила:

1. Обработчики (`catch`-блоки) рассматриваются по порядку, один за другим, причём сработать может только один из них, или ни одного.

2. Обработчик может поймать только исключение, возникшее во время работы соответствующего `try`-блока.

3. Если исключение не поймано ни одним из обработчиков, оно "выбрасывается" дальше, как если бы фрагмент кода, в котором исключение возникло, вовсе не был обрамлён никаким `try`-блоком.

Приведем пример (см. Листинги 4 и 5) такой обработки при вычислении обратной матрицы. Как известно, обратная матрица существует только для квадратных невырожденных матриц. Тогда понятно, что функция вычисления обратной матрицы должна генерировать исключения в случаях, когда, во-первых, матрица не является квадратной и, во-вторых, когда определитель матрицы равен 0.

```
double** InverseMatrix(double** a, int m, int n) {
    if(m != n)
        throw "The matrix must be square!";
    double det = Determinant (a, n);
    if(det == 0.0)
        throw 0;
    // code for calculating the inverse matrix
```

```
8     // ...
9 }
```

Листинг 4: Возбуждение исключений в функции, вычисляющей обратную матрицу

Программа, которая вычисляет обратную матрицу, должна содержать блок `try` для выявления указанных выше исключений и соответствующие обработчики `catch`.

```
1 double** a;
2 int m, n;
3 // ...
4 try {
5     double** b = InverseMatrix(a, m, n);
6 }
7 catch(char* str) {
8     cout << str << endl;
9 }
10 catch(int e) {
11     cout << "The matrix is degenerate!" << endl;
12 }
```

Листинг 5: Обработка возможных исключений при вычислении обратной матрицы

Отметим, что в приведенном примере генерация исключительных ситуаций происходит в функции вычисления обратной матрицы, а их обработка — в вызывающей её функции. Таким образом, генерация исключений и их обработка могут быть разделены между различными функциями.

Теперь рассмотрим ситуацию (см. Листинг 6), когда в блоке `try` сначала производится умножение двух матриц, которое может сгенерировать исключение типа `int` со значением `1` (когда размеры перемножаемых матриц не корректны), а затем вычисление обратной для первой матрицы.

```
1 double** a, **b;
2 int m, n, k, l;
3 // ...
4 try {
5     double** c = MultiplyMatrix(a, m, n, b, k, l);
6     double** d = InverseMatrix(a, m, n);
7 }
8 catch(char* s) {
9     cout << s << endl;
10 }
11 catch(int i) {
12     if(i == 0)
13         cout << "The matrix is degenerate!" << endl;
```

```
14      if(i == 1)
15          cout << "Matrices of this size cannot be multiplied!"
                << endl;
16  }
```

Листинг 6: Первый вариант обработки связанных исключений

В этом случае возникновение ошибки при выполнении операции перемножения матриц приведет к её обработке и передаче управления операторам `catch`, которые следуют за `try`-блоком. Таким образом, вычисление обратной матрицы производиться не будет, даже если это можно сделать без каких-либо ошибок. Такой подход обычно используют, когда операторы блока `try` образуют взаимосвязанную последовательность действий и без корректного выполнения предшествующих действий невозможно правильно выполнить следующие.

Можно действовать иначе (см. Листинг 7) — разбить контролируемый блок на части, каждой из которых будет соответствовать свой набор исключений. Далее каждая часть заключается в свой собственный блок `try`. В этом случае ошибка, возникшая при перемножении матриц, никак не повлияет на получение обратной матрицы. Такой подход удобнее применять, когда решаемые в блоке `try` задачи не зависят друг от друга. Тогда все задачи, которые могут быть выполнены без ошибок, будут решены.

```
1  double** a, **b;
2  int m, n, k, l;
3  // ...
4  try {
5      double** c = MultiplyMatrix(a, m, n, b, k, l);
6  }
7  catch(int i) {
8      cout << "Matrices of this size cannot be multiplied!" <<
          endl;
9  }
10 try {
11     double** d = InverseMatrix (a, m, n);
12 }
13 catch(char* s) {
14     cout << s << endl;
15 }
16 catch(int i) {
17     cout << "The matrix is degenerate!" << endl;
18 }
```

Листинг 7: Второй вариант обработки связанных исключений

5 Обработчики с многоточием

В некоторых случаях полезно отследить произвольное исключение независимо от его типа с помощью обработчика с многоточием `catch(...)`. Обработчики с многоточием часто используются в связке с оператором `throw`, когда тот вызывается без параметров. Это позволяет поймать исключение произвольного типа, выполнить те или иные действия, после чего бросить исключение дальше. В качестве примера рассмотрим следующий код:

```
1 void f(int n) {
2     int *p = new int[n];
3     // ...
4     delete[] p;
5 }
```

Листинг 8: Проблема обработки исключений при работе с динамической памятью

Если во время выполнения кода, находящегося между `new` и `delete` возникнет исключение, то `delete` не будет выполнен и массив, на который указывал `p`, будет продолжать занимать память. Данную проблему можно решить так:

```
1  void f(int n) {
2      int *p = new int[n];
3      try {
4          // ...
5      }
6      catch(...) {
7          delete[] p;
8          throw;
9      }
10     delete[] p;
11 }
```

Листинг 9: Решение данной проблемы

6 Автоматическая очистка

В Листинге 9 мы были вынуждены перехватывать, а после этого заново генерировать исключения произвольного вида, чтобы обеспечить корректное освобождение локально выделенной динамической памяти. Так приходится делать не всегда, так как работа с исключениями может быть упрощена за счёт использования механизма автоматической очистки, который состоит в следующем.

Замечание 3. *Компилятор гарантирует вызов деструкторов для всех локальных объектов, у которых имеются деструкторы, прежде чем текущая функция завершится штатно либо по исключению.*

7 Преобразования типов исключений

Тип исключения, указанный в `catch`-блоке не всегда точно совпадает с типом выражения, использованного в операторе `throw`. Поэтому важно знать, что правила преобразования типов в данном случае отличаются от "обычных" правил, действующих, например, при вызове функций.

Замечание 4. *Целочисленные типы при обработке исключений не преобразуются друг к другу. Таким образом, обработчик* `catch(int x)` *не поймает исключение типа* `short` *или* `long`.

Допускаются только следующие преобразования:

- `throw any_type` $\longrightarrow$ `catch(any_type &);`
- `throw any_type *` $\longrightarrow$ `catch(const any_type *)`
  (но не `throw const any_type *` $\not\longrightarrow$ `catch(any_type *)`!);
- `throw any_type &` $\longrightarrow$ `catch(const any_type &)`
  (но не `throw const any_type &` $\not\longrightarrow$ `catch(any_type &)`!).

8 Обработка исключений с помощью специального класса

Обычно, при возникновении исключения желательно передать обработчику максимум информации о возникших проблемах. Обычные типы данных плохо пригодны для этого.

Пример 1. *При открытии файлов может быть полезным передавать обработчику не только строку типа «*`open file error`*», но и имя соответствующего файла, а также значение переменной* `errno` *сразу после выполнения* `fopen` *для анализа возможных проблем.*

Программа может использовать несколько библиотек и быть разделена на несколько подсистем, создаваемых независимо друг от друга. При такой организации желательно иметь универсальный способ разделения исключений по признаку их возникновения в той или иной подсистеме. Встроенные типы для этого принципиально не подходят.

В связи с этим чаще всего в качестве типа исключения выступает специально созданный для этого класс, а значением исключения — объект такого класса.

"Выбрасываемый" в качестве исключения объект, как правило, создаётся локально. Локальные объекты имеют ограниченный срок жизни и исчезают вместе со стековым фреймом создавшей их функции. Как следствие, объект "пойманный" обработчиком исключения не может быть тем же экземпляром объекта, который присутствовал в качестве аргумента оператора `thow`, а является его копией.

Замечание 5. *В классе, используемом для исключений, практически всегда обязан присутствовать конструктор копирования.*

```
1  class FileException {
2      int err_code;
3      char *filename;
4      char *comment;
5  public:
6      FileException(const char *fn, const char *cmt);
7      FileException(const FileException& other);
8      ~FileException();
9      const char *GetName() const { return filename; }
10     const char *GetComment() const { return comment; }
11     int GetErrno() const { return err_code; }
12 private:
13     static char *strdup(const char *str);
14 };
```

Листинг 10: Заголовок класса `FileException`

```
1  FileException::FileException(const char *fn, const char *cmt)
       {
2      err_code = errno;
3      filename = strdup(fn);
4      comment  = strdup(cmt);
5  }
6  FileException::FileException(const FileException& other) {
7      err_code = other.errno;
8      filename = strdup(other.filename);
9      comment  = strdup(other.comment);
10 }
11 FileException::~FileException() {
12     delete[] filename;
13     delete[] comment;
14 }
15 char* FileException::strdup(const char *str) {
16     char *res = new char[strlen(str) + 1];
17     strcpy(res, str);
18     return res;
```

```
19 }
```

Листинг 11: Реализация класса `FileException`

```
1  // ...
2  try {
3      // ...
4      FILE *f = fopen(file_name, "r");
5      if(!f)
6          throw FileException(file_name, "file open error");
7      // ...
8  }
9  // ...
10 catch(const FileException &ex) {
11     fprintf(stderr, "File exception: %s (%s): %s\n", ex.
           GetName(), ex.GetComment(), strerror(ex.GetErrno()));
12     return 1;
13 }
14 // ...
```

Листинг 12: Пример использования класса `FileException`

9 Контрольные вопросы и задания

1. Изучите самостоятельно возможные методы обнаружения ошибок ввода при использовании объекта `std::cin`.

2. Почему не следует применять механизм обработки исключений для штатного выхода из вложенных управляющих конструкций или глубоких цепочек вызовов функций в программах на C++?

3. Приведите пример ситуации, когда функция `main` не сможет обработать возбуждённое исключение.

4. Что такое динамическая идентификация типов (RTTI — Run-Time Type Identification)? Какое это имеет отношение к обработке исключений?

5. Изучите самостоятельно особенности обработки исключений в языке Python.