


АКАДЕМИЯ АЙТИ

Обработка ошибок. Традиционные техники, исключения.

Дубров Денис Владимирович
Ведущий инженер-программист, к.ф.-м.н.

denis.dubrov@harman.com

Структурный подход к обработке ошибок

Пример

```
bool saveData()
{
  bool bSuccess = false;
  FILE *pf = fopen("data.bin", "w");
  if (pf)
  {
    char achData[SOME_SIZE];
    getData(achData);
    size_t uWritten = fwrite(
      achData, sizeof (char),
      SOME_SIZE, pf);
```

Пример (окончание)

```
    if (SOME_SIZE == uWritten)
    {
      // и т. д.
      bSuccess = true;
    }
    fclose(pf);
  }    // if (pf)
  return bSuccess;
}    // saveData()
```

Обработка ошибок средствами библиотеки C

Пример

```
#include <setjmp.h>

jmp_buf env_loadData;

FILE *openData()
{
  FILE *pf = fopen("data.bin", "r");
  if (!pf)
    longjmp(env_loadData, 1);
  return pf;
}
```

Обработка ошибок средствами C (окончание)

Пример

```
void loadData()
{
  int nError = setjmp(
    env_loadData);
  switch (nError)
  {
    case 0:
    {
      FILE *pf = openData();
      // ...
      fclose(pf);
      break;
```

Пример (окончание)

```
    case 1:
      cout
        << "Ошибка открытия файла"
        << endl;
      break;
    // ...
  }    // switch (nError)
}    // loadData()
```

Обработка ошибок средствами C++

Пример

```
FILE *openData()
{
  FILE *pf = fopen("data.bin", "r");
  if (!pf)
    throw FileError(errorCode, __FILE__, __LINE__);
  return pf;
}
```

Обработка ошибок средствами C++ (окончание)

Пример

```
void loadData()
{
  FILE *fp = NULL;
  try
  {
    fp = openData();
    // ...
  }
```

Пример (окончание)

```
  catch (const FileError &rcError)
  {
    cout <<
      "Ошибка " << rcError.m_nCode <<
      ", файл " << rcError.m_szFile <<
      ", строка " << rcError.m_nLine << endl;
  }
  // catch (const OtherError &rcError) {} и т. д.
  if (fp)
    fclose(fp);
}    // loadData()
```

Обработка сторонних исключений

Пример

```
void loadData()
{
  FILE *fp = NULL;
  try
  {
    fp = openData();
    // ...
  }
  catch (const FileError &rcError)
  {
    // ...
  }
  // ...
  catch (...)
  {
    if (fp)
      fclose(fp);
    throw;
  }
  if (fp)
    fclose(fp);
}    // loadData()
```

Пример (окончание)

Правила генерирования исключений

Определение

- При генерировании исключения создаётся **временный объект** — копия информации об исключении.
- На начало обработки исключения исходный объект может уже **не существовать**.
- Временный объект создаётся при помощи **инициализации** исходным.

Правила генерирования исключений

Пример (копирование объекта)

```
class BaseError { /* ... */ };
class NewError : public BaseError
   { /* ... */ };
void process()
{
  try
  {
    // ...
    NewError err;
    BaseError &rBase = err;
    throw rBase;    // BaseError
```

Пример (копирование, окончание)

```
    // ...
  }    // try
  catch (const NewError &)
  {
    // ...
  }
  catch (const BaseError &)
  {
    // ...
  }
}    // process()
```

Правила генерирования исключений (продолжение)

Определение

» Как обычно в случае временных объектов, компилятор имеет право исключить лишнее копирование в целях оптимизации.

Пример

```
std::string f()
{
  std::string result;
  // ...
  return result;
}
```

Пример (окончание)

```
int main()
{
  // ...
  std::string result_f = f();
  // ...
}
```

Правила генерирования исключений (окончание)

Определение

» Конструкция «**`throw;`**» повторно генерирует исключение с тем же самым временным объектом.

Виды блоков обработки исключений

Блоки обработки исключений

- Оператор **try**;
- Функциональный блок **try**.

Особенность

Функциональный блок **try** может либо повторно сгенерировать то же исключение, либо другое.

Пример (функциональный блок)

```
X::X()
try
  : m_Data1("Init"),
    m_Data2(func(0))
{
  // Тело конструктора
}
catch (const Error &rcError)
{
  // throw;
}
```

Соответствие типа обработчика типу исключения

Определение (соответствие)

Тип выражения в **throw**:

» [**const**][**volatile**]Error

- **catch** ([**const**][**volatile**]Error [&])
- **catch** ([**const**][**volatile**]BaseError [&]), и BaseError — **доступный однозначный** базовый класс для Error.

» [**const**][**volatile**]Error *[**const**][**volatile**], обработчик: **catch** ([**const**][**volatile**]Type *), к которому выражение в **throw** может быть приведено любым(и) из:

- стандартных преобразований указателей, кроме как к указателям на недоступные, неоднозначные базовые классы.
- квалификационных преобразований.

Обработка исключений любого типа

Замечание

catch (...) соответствует любому исключению.

Правила обработки исключений

Правила

- При генерировании исключения управление передаётся соответствующему обработчику ближайшего (последнего по времени) активного блока **try**.
- Если в ближайшем блоке **try** нет обработчика, соответствующего исключению, управление передаётся соответствующему обработчику следующего после него ближайшего блока **try** (динамически охватывающего).
- Из всех обработчиков блока **try**, соответствующих исключению, выбирается первый по порядку.
- Если для исключения нет ни одного соответствующего ему обработчика ни в одном из активных блоков **try**, вызывается функция `std::terminate()`, аварийно завершающая программу.

Обработка исключений в охватывающих блоках

Пример

```
void f()
{
  try
  {
    throw Error();
  }
  catch (const AnotherError &)
  {
    // ...
  }
}
```

Пример (окончание)

```
int main()
{
  try
  {
    f();
  }
  catch (const Error &)
  {
    // ...
  }
}
```

Раскрутка стека

Определение

Раскрутка стека: (Stack Unwinding) — процесс вызова деструкторов для всех автоматических объектов с момента входа в блок **`try`**, в чей обработчик было передано управление, до момента вызова **`throw`**.

Правила раскрутки стека

- Объекты уничтожаются в порядке, обратном их созданию.
- Для частично созданных или уничтоженных объектов уничтожаются все их **полностью созданные** подобъекты (завершился конструктор, но не начал выполняться деструктор).
- Раскрутка завершается к началу выполнения обработчика исключения.

Использование раскрутки стека

Пример

```
#include <fstream>

void process()
{
  ifstream ifs("Data.bin");
  ifs.exceptions(ios_base::failbit | ios_base::badbit);
  int n = do_something(ifs);
  if (n > 10)
    return;
  do_something_more(ifs);
}
```

Спецификация исключений

Определение

Спецификация исключений: список типов исключений для функции (или указателя на неё), которые ей разрешается генерировать (напрямую или через вызовы других функций). Тип генерируемого исключения должен **соответствовать** одному из типов списка.

Пример

```
long f(long) throw (Error, AnotherError);
void g(char) throw (Error);
char h(void) throw ();    // noexcept;    noexcept(true);
void v(void);             // noexcept(false);
```

Правила использования спецификаций

Правила

- Спецификация исключений не является частью типа функции (**noexcept** является, начиная с C++17).
- Спецификация исключений должна быть одинаковой для всех объявлений функции.
- Для виртуальных функций производных классов спецификация исключений должна разрешать только исключения, разрешённые для перекрываемой функции базового класса. Аналогично — при присваивании указателей на функции.

Правила проверки спецификаций

Правила

- Если исключение, не соответствующее **динамической** спецификации, выходит за пределы функции, вызывается функция `std::unexpected()`, которая вызывает пользовательскую функцию, определённую при помощи `std::set_unexpected()`.
- Версия **noexcept** не выбрасывает `std::unexpected()` и может не раскручивать стек при исключении.

Правила проверки спецификаций (продолжение)

Пример

```
#include <exception>

void my_unexpected()
{
  // ...
}
```

Пример

```
int main()
{
  std::set_unexpected(my_unexpected);
  // ...
}
```

Правила проверки спецификаций (продолжение)

Правила (продолжение)

- Реализация обработчика `std::unexpected()` по умолчанию вызывает `std::terminate()`
- Если пользовательская реализация генерирует исключение, **разрешённое** спецификацией функции, оно замещает исходное исключение.

Правила проверки спецификаций (продолжение)

Пример

```
void my_unexpected()
{
  throw GoodError();
}
```

Пример

```
void process() throw (GoodError)
{
  throw BadError();
}
```

Правила проверки спецификаций (продолжение)

Правила (продолжение)

» Если пользовательская реализация генерирует исключение, **не разрешённое** спецификацией функции, но спецификация **разрешает** `std::bad_exception`, некоторый производный от него объект замещает исходное исключение.

Правила проверки спецификаций (продолжение)

Пример

```
void my_unexpected()
{
  throw AnotherBadError();
}

void process() throw (GoodError, std::bad_exception)
{
  throw BadError();
}
```

Правила проверки спецификаций (окончание)

Правила (окончание)

» Иначе вызывается функция `std::terminate()`.

Пример

```
void my_unexpected()
{
  throw AnotherBadError();
}
```

Пример

```
void process() throw (GoodError)
{
  throw BadError();
}
```

Повторная генерация исключения

Определение

Исключение считается **непойманным** с момента завершения вычисления выражения исключения (либо с момента повторного генерирования) до окончания инициализации параметра исключения в обработчике (либо входа в `std::terminate()` или `std::unexpected()`). В это время входит раскрутка стека.

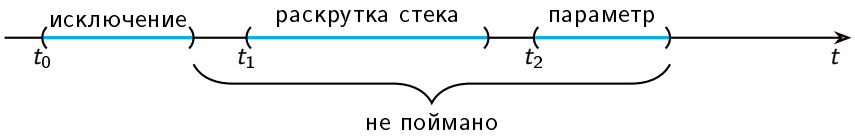

Рис.: процесс генерации и обработки исключения

Повторная генерация исключения (продолжение)

Исключения, генерируемые, пока другое исключение не поймано

- в деструкторах, вызываемых при раскрутке;
- в функциях, вызываемых после создания объекта исключения.

Повторная генерация исключения (продолжение)

Исключения, генерируемые, пока другое исключение не поймано

- в деструкторах, вызываемых при раскрутке;
- в функциях, вызываемых после создания объекта исключения.

Предупреждение

Если такие исключения не обрабатываются другим обработчиком, их генерирование приводит к вызову `std::terminate()`.

Повторная генерация исключения (продолжение)

Пример

```
struct SomeError
{
  SomeError() {}
  SomeError(const SomeError &)
    { throw UncaughtError(); }
};
struct Data
{
  ~Data()
    { throw UncaughtError(); }
};
```

Пример (окончание)

```
void process()
{
  try
  {
    Data d;
    throw SomeError();
  }
  catch (SomeError) // по значению
    { /* ... */ }
}
```

Повторная генерация исключения (окончание)

Функция `std::uncaught_exception()`

» `bool std::uncaught_exception() throw ();`
возвращает **`true`**, если в данный момент существует непойманное исключение.

Исключения стандартной библиотеки C++

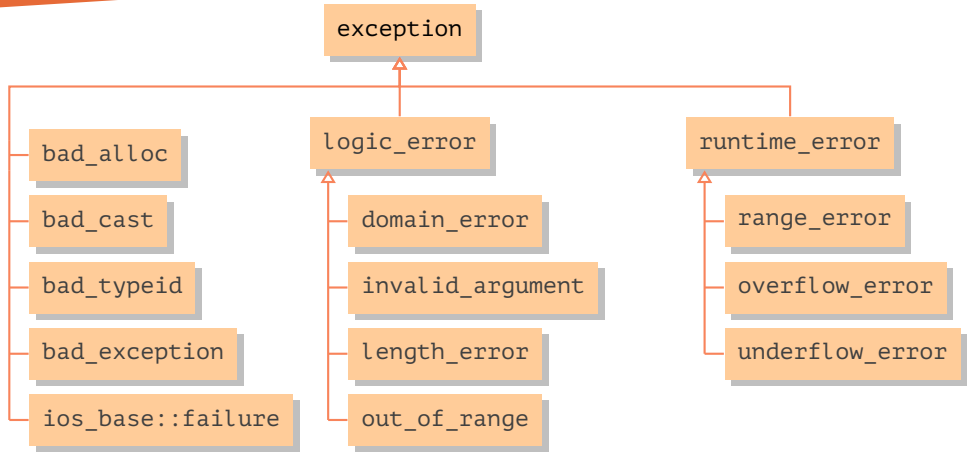

Рис.: иерархия классов исключений стандартной библиотеки C++

Объявления функций, использующих исключения

Функции, использующие исключения

В каждом транслируемом модуле неявно объявлены в глобальном пространстве имён следующие функции:

Функции, использующие исключения (C++98)

```
void *operator new (std::size_t) throw (std::bad_alloc);
void *operator new [] (std::size_t) throw (std::bad_alloc);
void operator delete (void *) throw ();
void operator delete [] (void *) throw ();
void operator delete (void *, std::size_t) throw ();
void operator delete [] (void *, std::size_t) throw ();
```

Объявления функций, использующих исключения

Функции, использующие исключения

В каждом транслируемом модуле неявно объявлены в глобальном пространстве имён следующие функции:

Функции, использующие исключения (C++11)

```
void *operator new (std::size_t);
void *operator new [] (std::size_t);
void operator delete (void *) noexcept;
void operator delete [] (void *) noexcept;
void operator delete (void *, std::size_t) noexcept;
void operator delete [] (void *, std::size_t) noexcept;
```

Правила объявления функций выделения (освобождения) памяти

Правила

- Должны объявляться в глобальном пространстве имён или внутри класса.
- Должны возвращать **void** * (**void**).
- Первый параметр должен иметь тип `std::size_t` (**void** *).
- (Функции-члены классов могут иметь второй параметр типа `std::size_t`.)

Соответствие функций выделения/освобождения памяти

Определения

Обычная функция: не имеет дополнительных параметров.

Функция с размещением: (placement) — имеет дополнительные параметры.

Пример (обычные функции выделения/освобождения памяти)

```
void *operator new (std::size_t);
void operator delete (void *);
struct X
{
  static void *operator new (std::size_t);
  static void operator delete (void *, size_t);
};
```

Соответствие функции освобождения памяти

Правила

- Обычной функции выделения памяти соответствует обычная функция освобождения.
- Функции выделения памяти с размещением соответствует функция освобождения с размещением с тем же количеством параметров, если типы всех параметров, кроме первых, совпадают (после возможных преобразований).
- Функции удаления с размещением передаются те же параметры, что и в функцию выделения (кроме первого).

Поведение операции new при исключении

Правила

- Если инициализация объекта прерывается исключением, операция **new** вызывает соответствующую ей функцию освобождения памяти и продолжает распространение исключения.
- Если соответствующая функция освобождения памяти отсутствует (или не может быть выбрана однозначно из перегруженных кандидатов), память не освобождается.

Соответствие функций (окончание)

Пример (пользовательские функции)

```
void *operator new (
  std::size_t nSize, char ch);     // (1)

void operator delete (
  void *pvData, char ch);          // (2)

void operator delete (
  void *pvData);                   // (3)
```

Пример (окончание)

```
void f()
{
  // ...
  X *pX = new ('!') X;
  delete pX;
  // (1, 3) либо (1, 2)
  // ...
}
```

Привила использования исключений

Правила

- В качестве информации об исключении предпочтительнее использовать сам объект, а не указатель на объект.
- В качестве информации об исключении рекомендуется использовать объекты классов, описанных в стандартной библиотеке (`std::exception` и т. д.), либо производных от них.
- В обработчике исключений предпочтительно параметр-исключение объявлять как ссылку, а не значение.

Привила использования исключений (окончание)

Правила (окончание)

- Необходимо обеспечивать при возникновении исключений корректное освобождение ресурсов и согласованное состояние данных.
- Необходимо не допускать выход исключений за пределы деструкторов, а также функций, выполняемых во время передачи исключения обработчику.
- Вместо динамических спецификаций исключений предпочтительнее использовать спецификации **noexcept**.
- Механизм исключений является ресурсоёмким, поэтому им следует пользоваться только для обработки ошибок.




АКАДЕМИЯ АЙТИ

Спасибо за внимание!

Центральный офис:
Москва, Варшавское шоссе 47,
корп.4, 10 этаж
Тел: +7 (495) 662-7894, 662-7895
Факс: +7(495) 974-7990
e-mail: academy@it.ru

www.academy.it.ru