# Лекция № 4.
# Обработка ошибок и механизм исключительных ситуаций в C++

# СОДЕРЖАНИЕ

# 1 ОБРАБОТКА ОШИБОК

Программа — это последовательность обработки ошибок.

*Неизвестный автор*

Обработка ошибок требуется в любой программе. Чем сложнее логика работы программы и структура исходных данных, тем больше различных ошибок может возникнуть, и соответственно увеличиться код их обработки. Однако, основная проблема заключается не в том, что обработка ошибок занимает много места, — в конце концов, это тоже часть логики работы приложения, — а в том, что это вынуждает менять структуру программы в ущерб упомянутой логике и с риском новых ошибок.

*Пример:* требуется получить информацию из удаленной базы данных (БД), но не работает сеть. Сначала сетевая подсистема должна сообщить подсистеме работы с БД, что соединение не может быть установлено. Подсистеме работы с БД требуется сообщить вызывающему коду, что получение данных невозможно (при этом без потери сведений, что невозможно подключение, так как нет сети). Вызывающий код должен отправить сведения об ошибке разработчику по почте, но сети нет, и это не удается. Эту проблему также нужно корректно обработать, записав сведения об ошибках (их уже пять) в файл журнала, который может оказаться недоступен для записи, и т. д. Наконец, нужно показать пользователю окно с сообщением только о невозможности доступа к данным.

Как видно из примера, логика обработки ошибок запутана, в ней участвуют разные подсистемы приложения, а кроме того, сам код обработки может порождать новые ошибки.

## 1.1 Классические подходы к обработке ошибок

Долгое время основных методов обработки ошибок, не считая заведомо непрактичных, существовало три [2, п. 13.1.2]:

1) *Проверка возвращаемых значений.* При этом функция, в которой могли возникнуть ошибки, возвращает код ошибки (или специальный код «Нет ошибок»), обычно это число. Вызывающий код может анализировать возникшие ошибки. Данный метод характерен для большинства небольших библиотек, особенно на C.

2) *Установка глобального признака ошибки.* При возникновении ошибок функция устанавливает некий глобальный (для приложения или подсистемы) признак возникшей ошибки со всей необходимой информацией. Внешний код может проверить этот глобальный признак после вызова. Данный метод применяется обычно в сочетании с предыдущим в сравнительно крупных библиотеках с интерфейсом в стиле C: Windows API, OpenGL (графическая библиотека), MySQL API (доступ к БД MySQL).

3) *Вызов пользовательской функции-обработчика ошибок.* Данный метод предполагает, что пользователь алгоритма предоставит указатель на специальную функцию, которая будет вызвана в случае ошибки. Этот способ наименее удобен для пользователя, но применяется иногда для асинхронных случаев (например, для обработки ошибок фоновой передачи данных по сети).

У классического подхода к обработки ошибок имеются существенные недостатки. Основной, неустранимой, проблемой является то, что ответственность за проверку ошибок возлагается на программиста. Если она не выполнена, программа продолжит работу с непредсказуемыми последствиями. Еще одной важной проблемой является управление ресурсами: при завершении алгоритма — штатном или аварийном — необходимо корректное освобождение всех выделенных в его коде ресурсов. Классический подход не дает иных методов это обеспечить, кроме как вручную, что иногда может быть непросто: ресурсы могут выделяться в разных местах, в зависимости от условий, быть задействованы асинхронно и т. п.

## 1.2 Механизм исключительных ситуаций

Механизм исключительных ситуаций и средства их обработки (говорят: *обработка исключений, exception handling)* призван решить проблемы классического подхода к обработке ошибок. Основная идея — обозначить блок, где возможны ошибки, и блок обработки ошибок, после чего при возникновении ошибки передать управление в блок её обработки вместе с любой необходимой информацией. При этом блок обработки ошибок не привязан к месту их возникновения, но вызывается автоматически. Таким образом решается принципиальная проблема классического подхода, связанная с ответственностью программиста-автора каждого блока за обработку всевозможных ошибок в нем.

Для C++ характерно, что если исключение не перехвачено в той же функции, где оно выброшено, и поиск обработчика продолжается во внешнем коде (происходит *разворот стека, stack unwinding),* — будут вызваны деструкторы всех созданных локальных переменных, как если бы произошел штатный выход из функции [1, подраздел 15.2]. Данная особенность делает особенно полезной идиому RAII (Resource Acquisition is Initialization), при которой объект завладевает ресурсом, например, областью памяти или открытым файлом, и освобождает этот ресурс в деструкторе [2, подразделы 13.3 и 5.2].

Как недостаток механизма исключительных ситуаций можно отметить то, что его поддержка требует генерации компилятором некоторого дополнительного кода, немного замедляющего работу программы. Обычно это не является проблемой, так как эти *накладные расходы времени (performance overhead)* сравнительно малы [2, п. п. 13.1.5

и 13.1.7]. Тем не менее, в некоторых специфичных случаях обработку исключений использовать не рекомендуется (например, в системах реального времени) или запрещается на уровне компилятора, как на некоторых встраиваемых или мобильных платформах (например, Symbian).

### 1.2.1 Пример использования исключений

Рассмотрим обработку исключений на примере демонстрационного класса строки `String`. Реализуем функцию, сокращающую имя, фамилию и отчество до «Ф. И. О.». Функция справедливо предполагает, что у человека всегда есть имя и фамилия, а отчество может отсутствовать, поэтому перед обращением к нулевому символу проверяется только длина отчества в рамках логики алгоритма, а не как обработка ошибок:

```
String getInitials(const String& name,
                   const String& surname,
                   const String& patronymic) {
     String result = String(toupper(surname[0])) + ". " +
          toupper(name[0]) + ".";
     if (patronymic.getLength() > 0) {
          result += " " + toupper(patronymic[0]) + ".";
     }
     return result;
}
```

Добавим обработку ошибок классическим способом, возвращая в случае любых проблем полное имя:

```
String getInitials(const String& name,
                   const String& surname,
                   const String& patronymic) {
     do {
          if (surname.getLength() == 0) break;
          String result = String(toupper(surname[0])) + ".";
          if (name.getLength() == 0) break;
          result += String(toupper(name[0])) + ".";
          if (patronymic.getLength() > 0) {
               result += String(" ") + toupper(patronymic[0]) + ".";
          }
          return result;
     } while (false);
     return surname + " " + name + " " + patronymic;
}
```

Цикл **do**…**while** (**false**) используется как замена **goto**: инструкция **break** передает управление в конец цикла, который выполняется единожды. Код обработки ошибок оказался смешан с кодом алгоритма, который стало сложнее читать.

Важно и то, что проверка правильности индексов, вообще говоря, — это задача самой строки, а не функций обработки строк. Можно было бы создать метод

```
bool String::tryGetSymbol(const size_t index, char& symbol) {
	if (index < length) {
		symbol = data[index];
		return true;
	}
	return false;
}
```

Однако, при таком подходе, интерфейс класса `String` становится менее удобен в угоду обработке ошибок (говорят: обработка ошибок не прозрачна). Решим возникшие проблемы при помощи механизма исключений.

Определим класс `StringException`, который будет представлять исключение — информацию об ошибке, — связанное со строками:

```
class StringException {
	String errorMessage;
public:
	StringException(const String& theErrorMessage)
		: errorMessage(theErrorMessage) { }
	const String& getErrorMessage() const {
		return errorMessage;
	}
};
```

Добавим в индексатор `String` код, оповещающий программу о возникновении ошибки при попытке получить символ по недопустимому индексу:

```
const char String::operator[](size_t index) {
	if (index >= length) {
		throw StringException(
			String("Индекс %u больше длины %u.").format(
				index, length)
			);
	}
	return data[index];
}
```

Использование инструкции **`throw`** называется *возбуждением, или выбрасыванием, исключения.* Метод `format()` пусть выполняет форматирование, подобно `printf()`, и возвращает результат.

Перепишем вновь функцию получения инициалов:

```
String getInitials(const String& name,
		   const String& surname,
		   const String& patronymic) {
```

```
        String result;
        try {
            result = String(toupper(surname[0])) + ". " +
                toupper(name[0]) + ".";
            if (patronymic.getLength() > 0) {
                result += " " + toupper(patronymic[0]) + ".";
            }
        } catch (const StringException& exception) {
            result = surname + " " + name + " " + patronymic;
            writeErrorToLog("Не удалось сократить имя " + result +
                ": ошибка при работе со строками: " +
                exception.getErrorMessage());
        }
        return result;
    }
```

В теле функции находится блок **try**…**catch**, аналогичный блоку **try**…**except** в Delphi: если внутри блока **try** возникает исключение, являющееся `StringException`, управление передается блоку **catch** (обработчику исключений). Туда же предается и объект исключения `exception`, из которого можно узнать дополнительную информацию — сообщение об ошибке. При классическом подходе программисту `getInitials()` пришлось бы формировать его самостоятельно (хотя это тоже сфера ответственности класса `String`). Заметим, что содержимое блока **try** практически соответствует изначальному варианту функции, то есть обработка ошибок прозрачна — изменение алгоритма из-за нее не требуется.

Блоков **catch** может быть несколько, и выбирается первый из них, тип которого соответствует типу выброшенного объекта исключения (такой же или его наследник). Если в текущем блоке подходящего обработчика найти не удается, поиск продолжается в обработчиках блоков выше по стеку. Так например, если бы функция `getInitials()` не могла самостоятельно обработать исключение, управление вызывающему коду. Вот как могла бы выглядеть функция печати инициалов списка людей, использующая *первый* вариант `getInitials()`:

```
    void printInitialList(const Person people[], size_t count) {
        for (size_t i = 0; i < count; i++) {
            const Person& person = people[i];
            try {
                const String& initials = getInitials(
                    person.getName(),
                    person.getSurname(),
                    person.getPatronymic());
                print(initials);
```

```
            } catch (const StringException& exception) {
                  print(String("Ошибка обработки данных ") +
                        + person.getID() + ": " +
                        exception.getErrorMessage());
            }
      }
}
```

**Листинг 1 — Функция печати списка людей с обработкой исключений**

Данная возможность позволяет обрабатывать исключения именно там, где это возможно, предоставляя компилятору передачу туда объекта исключения.

## 1.3 Гарантии в исключительных ситуациях

Если исключение можно перехватить и должным образом на него отреагировать, после обработки ошибки программа должна оказаться в некотором *«корректном» состоянии («valid» state)*. В объектно-ориентированных программах ошибки возникают в методах объектов, поэтому говорят о состоянии не программы, а объекта, так как при правильной инкапсуляции ошибка затронет только его. Понятие корректного состояния условное, но обычно это означает, что объект можно продолжить использовать извне. Для этого он должен быть полностью создан (конструктором), быть пригодным к уничтожению (деструктором), а также должны быть соблюдены его инварианты класса.

*Инварианты класса (class invariants)* — это некоторые свойства, присущие всем его объектам. Например, для класса `String` поле `data` должно всегда указывать на область памяти с массивом символов длиной, указанной в поле `length`, или быть равно `0`, если `length == 0`. Когда соблюдены все инварианты класса, объект находится в согласованном состоянии.

Однако, даже если состояние объекта «корректно», это не означает, что алгоритм может продолжить его использовать. Например, если функции `getInitials()` передано имя-пустая строка, в блоке **`try`** переменная `result` будет содержать сокращение от фамилии, что не имеет смысла с точки зрения алгоритма, использующего `getInitials()`. Поэтому обработчик ошибок дополнительно присваивает переменной-результату несокращенное полное имя, чтобы обеспечить логическую корректность состояния программы.

Обеспечение «корректного» состояния программы в случае возникновения ошибок называется *гарантией в исключительных ситуациях (exception guarantees)*. Существуют:

*Базовая гарантия (basic guarantee),* что объект будет в «корректном» состоянии, а также не произойдет утечек ресурсов, которые могли быть выделены к моменту ошибки.

*Сильная гарантия (strong guarantee)* — это базовая гарантия с дополнительным условием, что или блок выполняется успешно, или никаких изменений не происходит. Например, класс строки может гарантировать, что если операция `+=` не удастся (не хватит памяти и т. п.), исходная строка не изменится.

*Гарантия отсутствия исключений (nothrow guarantee)* — это базовая гарантия, подкрепленная условием, что исключений во внешний код выброшено не будет. Эта гарантия отличается тем, что может быть обеспечена в C++ не только логически, но и технически — спецификациями исключений (см. п. 2.2.2 ниже).

Объект или метод хотя бы с базовой гарантией называют *exception-safe,* а это его свойство — *exception-safety* (не переводится; дословно: «безопасность [использования] при исключительных ситуациях»). Вообще говоря, exception-safety не связана непосредственно с механизмом исключительных ситуаций, а может быть применена и к классическому подходу, но исторически это понятие появилось именно в связи с появлением нового, удобного способа обработки ошибок. Более того, современная концепция механизма исключительных ситуаций была заложена именно в рамках разработки C++, поэтому её описание в [2, подразделы 13.1—13.2] можно считать отправной точкой в данной теме.

Исключительные ситуации могут возникнуть за время жизни объекта в конструкторе, в методах и в деструкторе. При этом в последнем случае exception-safety возможна, только если обеспечить гарантию отсутствия исключений, так как после вызова деструктора объект нельзя переводить в «корректное» состояние — он должен быть уничтожен. Еще одна, техническая, проблема проявляется в том, что при раскрутке стека выполняются деструкторы объектов блок `try`; если такой деструктор выбросит новое исключение, программа безусловно аварийно завершится [1, подраздел 15.2, абзац 3]. Поэтому на практике непреложно правило: из деструктора никогда нельзя выбрасывать и пропускать исключения вовне.

# 2 МЕХАНИЗМ ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ В C++ И ЕГО ОСОБЕННОСТИ

## 2.1 Языковые средства обработки исключений

Как было показано в примерах, обработка исключительных ситуаций в C++ реализуется блоком **`try`**…**`catch`**, который отслеживает и перехватывает исключения, выброшенные оператором **`throw`**. Если исключение не было перехвачено, вызывается специальный обработчик `std::unhandled_exception()`, по умолчанию завершающий работу программы (см. п. 2.3.2 ниже).

### 2.1.1 Оператор `throw`

Вообще говоря, объектом исключения может быть значение любого типа, не обязательно объект, — даже целое число. Однако, такое использование **`throw`** не принято и порицается, так как технически нет возможности расширить такой «объект исключения», а логически это значение и вовсе не является объектом исключения, пока не выброшено оператором **`throw`**.

Можно оператором **`throw`** выбросить объект, сконструированный заранее. В частности, можно выбросить перехваченное исключение из блока **`catch`** — это называется *повторным выбрасыванием (rethrow)*. Данная возможность позволяет частично обработать ошибку на данном уровне и позволить продолжить её исправление.

Часто из блоков-обработчиков ошибок выбрасывают новые исключения. Например, перехватив `StringException`, функция `getInitials()` может выбросить исключение `NameException` с более предметной информацией об ошибке (не просто в обработке строк, а при обработке имени, причем конкретного).

Каков бы ни был объект исключения, оператор **`throw`** конструирует и передает в блок **`catch`** не сам объект, а его копию [1, подраздел 15.1].

### 2.1.2 Блок `catch`

Блоков-обработчиков ошибок **`catch`** может быть несколько. Из них выбирается первый, чей перехватываемый тип исключения соответствует выброшенному. С одной стороны, это означает, что наследник `StringException` класс `UnicodeStringException` будет перехвачен обработчиком `StringException`, а с другой стороны, если обработчик `UnicodeStringException` находится после обработчика `StringException`, обработчик класса-наследника никогда вызван не будет,

поэтому порядок блоков **catch** важен. В этом аспекте C++ подобен Delphi и большинству других объектно-ориентированных языков с обработкой исключений.

Можно создать блок **catch**, который перехватывал бы исключения любого типа [1, подраздел 15.3]. Очевидно, такой блок должен быть последним в списке. Этому блоку недоступен объект исключения, однако, можно повторно выбросить его оператором **throw** (как и любой другой блок **catch**):

```
catch (...) {
     throw;     // Выбрасывает обрабатываемое исключение.
}
```

### 2.1.3 Блок try уровня функции

Возможен следующий вариант использования блока **try**:

```
class Person {
     String name, surname, patronymic;
public:
     Person(const String& myName,
            const String& mySurname,
            const String& myPatronymic) :
     name(myName), surname(mySurname), patronymic(myPatronymic)
     try {
          // Тело конструктора.
     } catch (const StringException& exception) {
          throw PersonException(
               "Не удалось заполнить строковые поля!");
     }
};
```

Это и называется *блоком **try** уровня функции (function **try** block)*. В отличие от простого блока **try**, «реагирующего» не ошибки только в теле функции, данный вид блока **try** приводит и к перехвату ошибок, возникших в списке инициализации. Таким образом, блок **try** уровня функции имеет смысл только для конструкторов.

## 2.2 Спецификации исключений

### 2.2.1 Проверяемые исключения

В функции `printInitialList()` листинга 1 имеется неясность: как её разработчик узнаёт о возможности появления `StringException`, если в самой функции работы со строками не ведется? Может ли он быть уверен на то, что `Person::getName()` исключений не генерирует? Полагаться на документацию неразумно: даже если разработчик класса `String` укажет, что индексатор может выбрасывать исключение,

нет гарантии, что разработчик `getInitials()` не забудет этого сделать для своей функции. Представляется желательным, чтобы возможность появления и обработка тех или иных исключений автоматически проверялась компилятором. Это называется *механизмом проверяемых исключений (checked exceptions),* и в C++ он реализуется спецификациями исключений (exception specifications) [1, подраздел 15.4].

### 2.2.2 Спецификации исключений

Укажем, что индексатор `String` может выбросить исключение `StringException`, а методы `getLength()` и `getData()` — не могут:

```
class String {
public:
      const char operator[](const size_t index) const
            throw(StringException);
      const size_t getLength() const noexcept;
      const char* getData() const throw();
      …
};
```

Как видно из примера, чтобы указать возможные выбрасываемые исключения, нужно перечислить их в скобках в спецификации **`throw()`**, а пустой означает, что исключений не выбрасывается. Он эквивалентен спецификации **`noexcept`**, появившейся в C++11, а форма **`noexcept`** является сокращением **`noexcept(true)`**, см. 2.2.3 ниже. Отсутствие спецификации означает возможность выбрасывания любых исключений.

К сожалению, из-за необходимости обеспечивать обратную совместимость, в C++ нет жесткого требования обрабатывать все возможные (согласно спецификации) исключения в блоке **`try`** либо указывать необрабатываемые исключения в спецификации функции. Для кода примера: `getInitials()` по-прежнему не обязана иметь спецификацию исключений с указанием, что возможен выброс `StringException`. Более того, из соображений производительности использование **`throw()`** со списком типов считается устаревшим и не рекомендуется [1, подраздел D.4], [2, п. п. 13.5.1.3]. Для сравнения: в языке Java, если метод отмечен как выбрасывающий исключение типа `E`, при использовании данного метода вызывающий код должен либо обрабатывать исключение этого типа, либо указывать, что он также может выбрасывать исключение типа `E`.

Тем не менее, в классах-наследниках переопределения методов со спецификациями исключений должны будут указывать такие же ограничения на выбрасываемые

исключения или более жесткие (таким образом, если в базовом классе метод имеет спецификацию **noexcept**, он должен быть **noexcept** и в классе-наследнике).

Если функция выбрасывает исключение, не завяленное в спецификации, вызывается специальный обработчик `std::unexpected()`, рассмотренный в п. 2.3.3.

### 2.2.3 Спецификация noexcept и оператор noexcept

Ключевое слово **noexcept** используется и для спецификации исключений, и как оператор времени компиляции (подобно **sizeof**). Запишем следующую спецификацию для функции `getInitials()`:

```
1       String getInitials(/* аргументы опущены для краткости */)
2             noexcept(
3                   noexcept(surname[0]));
```

На строке 2 начинается *спецификация* **noexcept**, которая означает, что функция не бросает исключений, если аргумент спецификации истинен, иначе — что функция может выбрасывать любые исключения. На строке 3 расположен *оператор* **noexcept**, значение которого равно **false**, если аргумент *может* выбросить исключение, иначе — **true** [1, п. 5.3.7]. Таким образом, полностью приведенная спецификация означает, что `getInitials()` может выбрасывать исключения тогда и только тогда, когда может выбрасывать исключения индексатор строки. Выражение `surname[0]` означает вызов `String::operator[]()`, который может выбрасывать исключение `StringException`, следовательно, значение оператора **noexcept** — **false**, и спецификация получается **noexcept(false)**, то есть, функция может выбрасывать исключения.

Оператор **noexcept** действует во время компиляции, поэтому его аргумент в программе не вычисляется, то есть, нужно аргумент рассматривать как выражение (с неявными вызовами функций), а не как значение этого выражения. Например, можно было бы в примере вместо `surname[0]` записать `surname[10]`, — с точки зрения оператора **noexcept** это означает одно и то же — вызов `String::operator[]()`.

Использование **noexcept** поощряется, так как это освобождает разработчиков от написания лишних проверок, а компилятор — от генерации кода поддержки исключений [2, п. п. 13.5.1.1].

*Примечание.* По состоянию на август 2013 г. данная возможность не поддерживается Visual Studio 2012 и Visual Studio 2013 Preview, но поддерживается GCC 4.8.1.

## 2.3 Средства стандартной библиотеки для работы с исключениями и их обработкой

### 2.3.1 Класс `std::exception` и его наследники

В стандартной библиотеке имеется специальный класс, который удобно и рекомендуется использовать как базовый для объектов исключений — `std::exception` (стандартный заголовочный файл `<exception>`). Его определение приведено в листинге 2.

```
namespace std {
	class exception {
	public:
		exception() noexcept;
		exception(const exception&) noexcept;
		exception& operator=(const exception&) noexcept;
		virtual ~exception();
		virtual const char* what() const noexcept;
	};
}
```

**Листинг 2 — Определение класса `std::exception` [1, п. 18.8.1]**

Метод `what()` возвращает описание ошибки. Спецификация **`noexcept`** специальных функций требует реализовывать их такими же и в классах-наследниках.

От `std::exception` унаследован ряд важных классов стандартных исключений:

а) `std::bad_alloc` выбрасывается реализацией **`new`** по умолчанию при невозможности выделить память [1, п. п. 18.6.2.1, п. 5.3.4].

б) `std::bad_cast` выбрасывается **`dynamic_cast`** при невозможности приведения одного ссылочного типа к другому [1, п. 5.2.7].

в) `std::nested_exception` удобен как один из базовых классов для исключений, хранящих другие объекты исключений [1, п. 18.8.6]. Например, перехватив `StringException` функция `getInitials()` могла бы выбрасывать исключение `DataException` (ошибка в данных), сохранив его полем объект `StringException`, являющийся первопричиной ошибки.

### 2.3.2 Функции аварийного завершения работы программы

При выбрасывании исключения, которое нигде не перехватывается, происходит вызов функции `std::terminate()` и завершение работы программы. Допускается также явно вызывать эту функцию для выхода из программы.

Можно установить собственный обработчик завершения работы программы, который не принимает аргументов и не возвращает значения, функцией

`std::set_terminate()` и функцией `std::get_terminate()` получить текущий обработчик [1, п. 18.8.3].

### 2.3.3 Функции обработки непредвиденных исключений

Технически же, действие спецификации заключается в том, что при выбросе исключения незаявленного типа вызывается специальный обработчик — `std::unexpected()`. Он не принимает параметров, ничего не возвращает и может быть установлен `std::set_unexpected()`, а по умолчанию завершает работу программы. Важно, что если он выбросит новое исключение, уже заявленного типа, оно будет передано соответствующему обработчику [1, п. 15.5.2 и подраздел D.11].

# БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. ISO/IEC 14882:2011 Programming Languages — C ++. — Введен в 2011 г. — 1338 с.
2. Stroustrup, Bjarne. The C++ Programming Language (Fourth Edition) / Bjarne Stroustrup. — Addison-Wesley. — 2013 г. — 1347 с.