### Лекция 13. Исключения

При разработке программы иногда бывает известно, что при её выполнении могут возникать ошибки и стопроцентного способа предотвратить их появление не существует. В таких ситуациях может быть предпочтительнее допускать вероятность возникновения проблем и писать код, являющийся достаточно надежным для того, чтобы справляться с этими ошибками аккуратным образом, без прерывания процесса выполнения.  
На проверку готового кода уходит почти столько же времени, сколько на его со­здание. Качество кода является (или должно являться) главной целью каждого программиста. Научившись устранять возможные проблемы еще до их появления, вы продви­нетесь далеко на пути к этой цели.   
Все существующие для этого приемы вместе называются *обработкой ошибок.* Если вы точно знаете, что нужно делать, и в контексте функции у вас имеется вся необходимая ин­формация, обработка ошибки достаточно просто выполняется в теле функции. Проблемы начинаются тогда, когда вы не обладаете всей информацией в текущем контексте, и вам необходимо передать информацию об ошибке в другой контекст, где эта ин­формация существует. Традиционно для это цели используется код возврата.  
Однако программисты часто иг­норируют информацию об ошибках, потому что проверять все возможные коды возврата после каждого вызова функции было бы слишком утомительно и неудобно. Кроме того, может оказаться, что результат анализа кода возврата из функции, в которой возникла исключительная ситуация, часто не имеет смысла, а увеличение размера программ и ухудшение их читаемости будет всегда.

Усовершенствование обработки ошибок является одним из самых эффективных путей повышения надежности программ.

В C++ для достижения этой цели имеется механизм *обработки исклю­чений.* О нем можно сказать следующее.

* Обработка ошибок гораздо проще программируется, а ее код не смешивает­ся с «обычным» кодом. Сначала вы программируете *нормальное* течение со­бытий, а позднее в отдельной секции пишется код для решения проблем. При многократном вызове функции обработка ошибок этой функции про­изводится только один раз и в одном месте.
* Ошибки могут игнорироваться. Если функция должна отправить сообще­ние об ошибке вызывающей стороне, она «запускает» объект, представляю­щий эту ошибку. Если вызывающая сторона не «перехватит» ошибку и не обработает ее, то ошибка переходит в следующую внешнюю динамическую область видимости, и так далее. В итоге либо ошибка будет перехвачена, либо про­грамма завершится из-за отсутствия обработчика для данного типа исклю­чения.

*Исключением* (exception) называется такая ошибка, которая генерируется в основном коде или в какой-то вызываемой им функции во время выполнения. Определение ошибки здесь немного более размыто, чем было до этого, поскольку исключения могут генерироваться при вводе данных с клавиатуры, при попытке открыть несуществующий файл, внутри функций и т.д. Например, исключение внутри функции может генерироваться в случае, если один из ее строковых параметров не начинается с буквы "а". Строго говоря, оно в таком случае не является ошибкой за рамками контекста функции, но будет восприниматься как таковое вызывающим эту функцию кодом.  
Пожалуй, простейшим примером является попытка обратиться к элементу массива за его пределами:

int myArray[] = {1, 2, 3, 4};  
int myElem = myArray[4];  
Этот код приведет к отображению сообщения «*out\_of\_rang»* о возникшем исключении и завершению приложения.

# Обработка исключений

Механизм обработки исключений переводит обработку ошибок на уровень языка программирования. Исключение представляет собой объект, который «генерируется» в месте ошибки и «перехватывается» соответствующим обработчиком исключения, предназначенным специально для обработки этого конкретного типа ошибки. Со стороны все выглядит так, словно при возникновении ошибки выполнение программы идет по другой, параллельной ветви. А это означает, что код обработки ошибок может быть отделен от кода нормального выполнения. Подобная изоляция упрощает программу, поскольку программисту не приходится постоянно проверять все возможные ошибки. Кроме того, сгенерированное исключение принципиально отличается от возвращаемого численного кода или флага, устанавливаемого функцией для обозначения ошибки, — эти признаки могут просто игнорироваться. Исключения не игнорируются, поэтому рано или поздно они заведомо будут обработаны. Наконец, исключения могут обеспечивать восстановление после аварийных ситуаций. Вместо завершения программы иногда удается исправить ошибку и продолжить работу, что значительно повышает устойчивость системы.  
Запуск исключений

Если в программе возникла исключительная ситуация (то есть в текущем контек­сте не хватает информации для принятия решения о том, как действовать дальше), информацию об ошибке можно передать во внешний, более общий контекст. Для этого в программе создается объект с информацией об исключении, который за­тем «запускается» из текущего контекста (говорят, что в программе *запускается исключение).* Вот как это выглядит:

//:L13 МуЕггог.срр

class MyError {//класс для объекта исключения

const char\* const data;

public:

MyError(const char\* const msg = 0) : data (msg) {}

};

void f() {

// "Запускаем" объект исключения:

throw MyError("something bad happened");//вызывается конструктор MyError

int main() {

// Как вскоре будет показано,

// здесь должен находиться "блок try":

f();

} //т.к. для MyError нет обработчика исключений,

//то программа завершится не нормально

MyError — обычный класс, конструктор которого в нашем примере получает тип char\*. При запуске исключения можно использовать произвольный тип (в том чис­ле и встроенные типы), но обычно для этой цели создаются специальные классы.

Ключевое слово throw производит целый ряд полузагадочных манипуляций. Сначала оно создает копию запускаемого объекта и фактически «возвращает» ее из функции, содержащей выражение throw, даже если тип этого объекта не соот­ветствует типу, который положено возвращать этой функции. Обработку исклю­чений можно упрощенно представить себе как альтернативный механизм возвра­та (но если зайти с этой аналогией слишком далеко, ничего хорошего не выйдет). Генерируя исключения, вы также можете выходить из обычных областей видимо­сти. В любом случае возвращается значение, а управление из функции или облас­ти видимости передается наружу.

На этом все сходство с командой return завершается — точка, в которую проис­ходит возврат, не имеет ничего общего с точкой возврата при обычном вызове функ­ции. Управление передается в специальную часть программы, называемую об­работчиком исключения; она может находиться далеко от того места, где было запущено исключение. Также уничтожаются все локальные объекты, созданные к моменту запуска исключения. Автоматическое уничтожение локальных объек­тов часто называется «раскруткой стека».

В программе могут запускаться объекты исключений разнообразных типов. Как правило, для каждой категории ошибок используется свой тип. Предполагается, что информация о возникшем исключении может содержаться как внутри объек­та, так и в *имени* его класса; благодаря этому в контексте вызова можно будет ре­шить, как поступить с исключением.

## Перехват исключений

Как упоминалось ранее, одно из преимуществ механизма обработки исключений C++ состоит в том, что он позволяет программисту сосредоточиться на решаемой задаче, а затем организовать обработку ошибок в другом месте.

#### Блок try

Если внутри функции происходит исключение, выполнение этой функции завер­шается. Если вы не хотите, чтобы команда throw приводила к выходу из функции, создайте внутри функции специальный блок, который должен решать проблемы (а возможно — запускать новые исключения). Этот блок, называемый *блоком try,* представляет собой обычную область видимости, перед которой ставится ключе­вое слово try:

try {

// Программный код. который может генерировать исключения

}

Если проверять ошибки по возвращаемому значению функций, вам придется «обрамлять» каждый вызов функции кодом подготовки и кодом проверки, даже при многократном вызове одной функции. При обработке исключений выполняе­мый код помещается в блок try, а обработка исключений производится после бло­ка try. Это существенно упрощает написание и чтение программы, поскольку ос­новной код не смешивается с кодом обработки ошибок.

#### Обработчики исключений

Конечно, программа должна где-то среагировать на запущенное исключение. Это место называется *обработчиком исключения,* в программу необходимо включить обработчик исключения для каждого типа перехватываемого исключения. Тем не менее полиморфизм распространяется и на исключения: один обработчик может перехватывать как определенный тип исключения, так и исключения классов, про­изводных от этого типа.

Обработчики исключений следуют сразу же за блоком try и обозначаются клю­чевым словом catch:

try {

// Программный код. который может генерировать исключения

}

catch(type1 id1) {

// Обработка исключений типа type1

}

catch(type2 id2) {

// Обработка исключений типа type2

}

catch(type3 id3) {

// И т. д.

}

catch(typeN idN)

// Обработка исключений типа typeN

}

// Здесь продолжается нормальное выполнение программы...

По своему синтаксису секции catch напоминают функции, вызываемые с од­ним аргументом. Идентификатор (id1, id2 и т. д.) может использоваться внутри обработчика по аналогии с аргументом функции, но если он не нужен — не ис­пользуйте его. Тип исключения обычно дает достаточно информации для его об­работки.

Обработчики должны находиться сразу же после блока try. Если в программе запускается исключение, механизм обработки исключений начинает искать пер­вый обработчик с аргументом, соответствующим типу исключения. Управление передается в найденную секцию catch, и исключение считается обработанным (то есть дальнейший поиск обработчиков прекращается). Выполняется только нуж­ная секция catch, а выполнение программы продолжается, начиная с позиции, сле­дующей за последним обработчиком для данного блока try.

Обратите внимание: в блоке try один тип исключения может генерироваться разными вызовами функций, но обработчик нужен только один.

//:L13 exeptions.cpp

// Демонстрация обработки исключений

#include <iostream>

using namespace std;

class Rainbow {

public:

Rainbow() { cout << "Rainbow()" << endl; }

~Rainbow() { cout << "~Rainbow()" << endl; }

};

void oz() {

Rainbow rb;

for(int i =0; i <3; i++)

cout << "there’s no place like home\n";

throw 47;

}

int main() {

try {

cout << "tornado, witch...\n";

oz();

}

catch (int) {

cout << "Auntie Em! "<<

"I had the strangest dream..."<< endl;

}

}

При выполнении команды throw в функции oz() начинается перебор секций catch до тех пор, пока не будет обнаружена секция catch с параметром типа int. Тело этой секции catch продолжает выполнение программы. Очень важно, что при выходе из функции oz() по команде throw вызывается деструктор объекта rb, хотя программа не вышла из операторного блока.

#### Завершение и продолжение

В теории обработки исключений существуют две основных модели: обработка с за­вершением программы и обработка с продолжением программы. В *модели с завер­шением программы* (поддерживаемой в C++) предполагается, что ошибка настолько серьезна, что автоматически продолжить программу с точки возникновения ис­ключения нельзя. Другими словами, при запуске исключения предполагается, что исправить ситуацию уже невозможно, и возвращаться *нежелательно.*

Альтернативная модель обработки ошибок предполагает, что обработчик исключе­ния каким-то образом исправит ситуацию, после чего сбойный фрагмент кода бу­дет автоматически выполнен заново; причем считается, что вторая попытка может оказаться успешной. Если вы захотите организовать подобную модель поведения в C++, вам придется явно передать управление к точке возникновения ошибки (обычно посредством вызова функции). Нередко блок try помещается в цикл while и выполняется до тех пор, пока результат не окажется удовлетворительным.

Опыт программирования показывает, что программисты, которые работали в операционных системах с поддержкой модели с продолжением, в конечном счете переходили на имитацию модели с завершением.

## Поиск подходящего обработчика

Когда в программе генерируется исключение, система обработки исключений на­чинает просматривать «ближайшие» обработчики в порядке их следования в ис­ходном коде. Если она обнаруживает совпадение, исключение считается обрабо­танным, и поиск на этом прекращается.

Поиск подходящего обработчика не требует идеального соответствия между исключением и его обработчиком. Объект (или ссылка на объект) исключения производного класса считается подходящим для обработчика, работающего с базо­вым классом. Однако если обработчик предназначен для объекта, а не для ссылки, объект исключения «усекается» до базового типа при передаче обработчику. Усе­чение безвредно, но оно приводит к потере всей информации, специфической для производного типа.

#### Перехват любых исключений

Иногда требуется написать обработчик для перехвата *любых* типов исключений. Для этой цели используется специальный список аргументов в виде многоточия

catch(...) {

cout << "an exception was thrown" << endl;

}

Поскольку такой обработчик перехватывает все исключения, он размещается *в конце* списка обработчиков (иначе следующие за ним обработчики никогда не выполняются).

Универсальный обработчик не может иметь аргументов, поэтому в нем невоз­можно получить какую-либо информацию об исключении или его типе. Такие сек­ции catch часто освобождают некие ресурсы и перезапускают исключение.

#### Перезапуск исключения

Перезапуск исключений обычно применяют для освобождения тех или иных ресурсов скажем, закрытия сетевых подключений или освобождения памяти в куче. При возникновении исключения иногда бывает не важно, какая ошибка породила его — просто нужно закрыть подключение, открытое ранее, после чего передать обработку исключения в другой контекст, ближе к пользо­вателю (то есть находящийся выше в цепочке вызовов). Конструкция catch(...) идеально подходит для таких случаев. Вы хотите перехватить *любые* исключе­ния, освободить ресурс, а затем перезапустить исключение для последующей обработки. Исключения перезапускаются командой throw без аргумента внут­ри обработчика:

catch(...) {

cout << "an exception was thrown" << endl;

// Освобождение ресурсов

throw;

}

Остальные секции catch того же блока try игнорируются — команда throw пере­дает исключение обработчикам следующего контекста. Вся информация объекта исключения сохраняется, поэтому обработчики внешнего контекста, перехваты­вающие конкретные типы исключений, смогут извлечь любую информацию, со­держащуюся в объекте.

#### Неперехваченные исключения

Как объяснялось в начале лекции, обработка исключений лучше традиционной мето­дики с возвратом кода ошибки, поскольку исключения не могут игнорироваться, а обработка исключения отделяется от непосредственно решаемой задачи. Если ни один из обработчиков, следующих за блоком try, не соответствует типу исключения, то исключение передается в контекст следующего уровня, то есть в функцию (или в блок try), в которой находится блок try, не перехвативший исключение, причем местонахождение этого внешнего блока try не всегда очевидно, поскольку он нахо­дится на более высоком уровне иерархии. Процесс продолжается до тех пор, пока в какой-то момент для исключения не будет найден подходящий обработчик. В этот момент исключение считается «перехваченным», и дальнейший поиск обработчи­ков для него не выполняется.

Зачистка

Одно из преимуществ обработки исключений состоит в том, что нормальный ход программы прерывается, и управление сразу передается в соответствующий обра­ботчик исключений. Но эта передача принесет пользу только в том случае, если в момент запуска исключения будет проведена необходимая деинициализация. Ме­ханизм обработки исключений C++ гарантирует, что при выходе из области види­мости для всех объектов этой области, *конструкторы которых завершены,* будут вызваны деструкторы.

Следующий пример убеждает в том, что для объектов с незавершенными кон­структорами деструкторы не вызываются. Кроме того, он показывает, что проис­ходит при запуске исключения в процессе создания массива объектов:

//L13:Сlеаnup.срр

// При запуске исключения уничтожаются только готовые объекты

#include <iostream>

using namespace std;

class Trace {

static int counter;

int objID;

public:

Trace() {

objID = counter++;

cout << "constructing Trace #" << objID << endl;

if(objID == 3) throw 3;

}

~Trace() {

cout << "destructing Trace #" << objID << endl;

}

};

int Trace::counter = 0;

int main() {

try {

Trace n1;

// Запуск исключения:

Trace array[5];

Trace n2; // Сюда не попадаем

} catch(int i) {

cout << "caught " << i << endl;

}

}

Класс Trace выводит информацию о создании и уничтожении своих объектов, что позволяет проследить за ходом выполнения программы. Класс подсчитывает созданные объекты в статической переменной counter, а идентификатор конкрет­ного объекта хранится в переменной objID.

Функция main() создает один объект n1(objID 0), а затем пытается создать мас­сив из пяти объектов Trace, но создание четвертого объекта (идентификатор 3) прерывается. Таким образом, создание объекта n2 так и не завершается. Из выход­ных данных программы видно, как происходит вызов деструкторов:

constructing Trace #0  
constructing Trace #1   
constructing Trace #2   
constructing Trace #3   
destructing Trace #2   
destructing Trace #1   
destructing Trace #0   
caught 3

Три элемента массива создаются успешно, но в процессе вызова конструктора четвертого элемента запускается исключение. Поскольку в main() конструирова­ние четвертого объекта (array[2]) не было завершено, в программе вызываются де­структоры только для объектов array[1] и аггау[0]. В конце уничтожается объект n1, но не объект n2, который так и не был создан.

Управление ресурсами

Программируя обработку исключений, всегда следует задавать себе вопрос: «Если произойдет исключение, будут ли освобождены задействованные ресурсы?» Как правило, механизм освобождения ресурсов работает достаточно надежно, но сущест­вует специфическая проблема, связанная с конструкторами: если до завершения конструктора будет сгенерировано исключение, то для данного объекта деструктор не вызывается. Это означает, что при написании конструкторов необходимо быть особенно внимательным.

Эта проблема может быть решена, если при создании конструкторов использовать методику « *Получение ресурсов при инициализации»* (Resource Acquisition Is Initialization, RAII), в которой управление ресурсом (выделение и освобождение) совмещается с основными точками жизненного цикла объекта ([1] (Глава 1. Обработка исключений, Управление ресурсами на уровне объектов)).

auto\_ptr

В типичной программе C++ динамическая память является наиболее часто исполь­зуемым ресурсом. По этой причине в стандарте предусмотрена RAII -оболочка для указателей на память в *куче* (шаблонный класс **auto\_ptr)***,* обеспечивающая автоматическое освобождение памяти. У шаблонного класса **auto\_ptr,** определяемого в заголовочном файле **<memory>,** имеется конструктор, получающий указатель на тип параметра (то есть тип, не­посредственно используемый в программе). Шаблон **auto\_ptr** также перегружает операторы \* и -> и выполняет соответствующие операции с исходным указателем, инкапсулированным в объекте **auto\_ptr.** Таким образом, с объектом **auto\_ptr** можно работать так, как если бы он был обычным указателем.   
 Шаблон **auto\_ptr** также удобен при работе с указателями на переменные клас­сов. Поскольку объекты классов, хранимые по значению, всегда уничтожаются, переменные типа **auto\_ptr** при уничтожении внешнего объекта всегда удаляют объект, связанный с хранящимся в них низкоуровневым указателем.

Стандартные исключения

Исключения стандартной библиотеки C++ также могут использоваться в ваших программах. Как правило, проще и удобнее начать со стандартного класса исключе­ния, чем пытаться определять собственный класс. Даже если стандартный класс де­лает не совсем то, что нужно, на его основе всегда можно создать производный класс.

Все стандартные классы исключений в конечном счете являются производны­ми от общего предка exception, определенного в заголовке <exception>. Два основных производных класса logic\_error и runtime\_error определяются в заголовке <stdexcept> (который включает <exception>). Класс logic\_error представляет ошибки в логике программирования — например, передачу недопустимых аргументов. К категории runtime\_error относятся те ошибки, которые возникают в результате непредвиден­ных обстоятельств (скажем, сбоев оборудования или нехватки памяти). В обоих классах, runtime\_error и logic\_error, определен конструктор с аргументом std::string. Он позволяет сохранить сообщение в объекте исключения и извлечь его позднее с помощью функции exception::what(), как это сделано в следующем примере:

//L13:StdExcept.cpp

// Создание класса исключения, производного от std::runtime\_error

#include <stdexcept>

#include <iostream>

using namespace std;

class MyError : public runtime\_error {

public:

MyError(const string& msg = "") : runtime\_error(msg) {}

};

int main() {

try {

throw MyError("my message");

} catch (MyError& х) {

cout << х.what() << endl;

}

}

Хотя конструктор runtime\_error сохраняет сообщение в подобъекте std::exception, класс std::exception не содержит конструктора с аргументом std::string. Обычно клас­сы исключений объявляются производными не от std::exception, а от runtime\_error или logic\_error (или одного из их потомков).

Ниже кратко описаны основные классы исключений.

exception  
Базовый класс для всех исключений, генерируемых стандартной библиотекой  
C++. Функция what() возвращает необязательную строку, указанную при ини­циализации исключения.

logic\_error  
Класс, производный от exception. Передает информацию о логических ошибках в программе. Предполагается, что такие ошибки обнаруживаются путем анали­за некоторых условий.

runtime\_erгог  
Класс, производный от exception. Передает информацию об ошибках времени выполнения. Предполагается, что такие ошибки обнаруживаются только во время выполнения программы.

Класс исключений потоков ввода-вывода ios::failure тоже является производ­ным от exception, но других подклассов не имеет.

Следующие классы (производные от logic\_error и от runtime\_error) можно ис­пользовать «как есть» или же создать на их основе производные классы для более специализированных типов исключений.

Далее перечислены классы исключений, производные от logic\_error.

domain\_error  
Сообщает о нарушении предусловия.

invalid\_argument  
Указывает, что функции, запустившей исключение, был передан недопусти­мый аргумент.

length\_error  
Сообщает о попытке создания объекта, размер которого больше или равен npos (размер наибольшего представимого значения контекстного типа, обычно std::size\_t).

out\_of\_range  
Сообщает о выходе аргумента из интервала допустимых значений.

bad\_cast  
Выдается при выполнении недопустимых выражений dynamic\_cast в подсисте­ме RTTI.

bad\_typeid  
Сообщает об использовании null-указателя р в выражении typeid(\*p). Также используется подсистемой RTTI.

А здесь перечислены классы исключений, производные от runtime\_error.

runtime\_error  
Сообщает о нарушении постусловия.

overflow\_error  
Сообщает о возникновении математического переполнения.

bad\_alloc  
Сообщает о неудачной попытке выделения памяти.

Спецификации исключений

Вообще говоря, вы не обязаны сообщать пользователям вашей функции, какие исключения она может запускать. Однако такое поведение считается нецивилизо­ванным — оно означает, что пользователи не будут знать, как написать код пере­хвата потенциальных исключений. При наличии исходных текстов они смогут просмотреть их и поискать команды throw, однако библиотеки не всегда поставля­ются с исходными текстами. Хорошая документация поможет решить проблемы, но много ли найдется хорошо документированных программных проектов? Спе­циальный синтаксис C++ позволяет сообщить пользователю, какие исключения запускаются данной функцией, чтобы он мог обработать их. Речь идет о необяза­тельной *спецификации исключений*, указываемой в объявлении функции после списка аргументов.

Спецификация исключений состоит из ключевого слова throw, за которым в круглых скобках перечисляются типы всех потенциальных исключений, которые могут запускаться данной функцией. Объявление функции может выглядеть при­мерно так:

void f() throw(toobig, toosmall. divzero);

В отличие от этого объявления традиционное объявление функции означает, что функция может запускать исключения *любых* типов:

void f();

Однако следующая конструкция говорит о том, что функция не может запускать *никаких* исключений (проследите за тем, чтобы функции, находя­щиеся на очередном уровне в цепочке вызовов, не передавали исключения на­верх!):

void f() throw();

Если хороший стиль программирования, полнота документации и удобства работы с функцией вам не безразличны, обязательно включайте спецификации исключений в те функции, которые их запускают.

Программирование с учетом исключений

Для большинства программистов исключения яв­ляются новшеством, к которому придется привыкать. Далее приводятся рекомен­дации по программированию с учетом исключений.

Когда лучше обойтись без исключений

Исключения — не панацея; не злоупотребляйте ими. В этом разделе рассматрива­ются ситуации, в которых применять исключения *не рекомендуется.*

Принимая решение об использовании исключений, лучше всего руководство­ваться правилом: *исключения запускаются только тогда, когда поведение функ­ции не соответствует ее спецификации.*

Асинхронные события

Асинхронные события - это события, которые происходят вне нормальной последовательности выполнения программы и появление которых невозможно предугадать. Исключения C++ не могут использоваться для обработки асинхрон­ных событий, потому что исключение и его обработчик принадлежат к одному стеку вызова, тогда как асинхронные события должны обрабатываться совершенно отдельным кодом (как правило — процедурами обработки прерываний), который не входит в нормальную последовательность выполнения программы. Не запускайте исключения из обработчиков прерываний!

Впрочем, это не значит, что асинхронные события не могут *ассоциироваться* с исключениями. Просто обработчик прерывания должен отработать как можно быстрее и вернуть управление. Обычно для этого в обработчике прерывания уста­навливается флаг, синхронно проверяемый в основном коде программы.

Устранимые ошибки

Если вы располагаете достаточной информацией для обработки ошибки, запус­кать исключение не требуется. Решите проблему в текущем контексте, не переда­вая ее в контекст более высокого уровня.

Кроме того, исключения C++ не запускаются для событий машинного уровня (таких, как деление на ноль). Предполагается, что такие события обрабатываются другими средствами, например операционной системой или оборудованием. При таком подходе исключения C++ работают с разумной эффективностью, а их при­менение ограничивается условиями программного уровня.

Управление последовательностью выполнения

На первый взгляд кажется, что исключения представляют собой альтернативный механизм возврата управления или отдаленное подобие команды switch. У некото­рых программистов возникает искушение использовать исключения вместо стан­дартных языковых механизмов, но делать этого не стоит. Прежде всего, обработка исключений значительно уступает по эффективности нормальному выполнению программы. Не стоит идти на затраты, связанные с исключениями, в обычной про­грамме. Кроме того, исключения, причиной которых не являются ошибки, собьют с толку пользователей вашего класса или функции.

Исключения не обязательны

Некоторые программы (например, несложные утилиты, ограничивающиеся полу­чением ввода и его элементарной обработкой) достаточно просты. В таких про­граммах тоже могут происходить сбои: неудачные попытки выделения памяти, открытия файлов и т. д. В таких программах можно вывести сообщение и пору­чить системе «прибрать за программой», вместо того чтобы самостоятельно пере­хватывать все исключения и освобождать все ресурсы. Короче говоря, если ваша программа может обойтись без исключений — не используйте их.

Новые исключения, старый код

Еще одна характерная ситуация возникает при модификации существующих про­грамм, не поддерживающих исключения. Допустим, к системе подключается биб­лиотека, в которой *используются* исключения; нужно ли изменять весь существу­ющий код? Если в системе уже имеется нормальная схема обработки ошибок, самое тривиальное решение — заключить максимально большой блок, в котором исполь­зуется новая библиотека (возможно, это будет весь код main()), в блок try с catch(...) с выводом простейших сообщений об ошибках. Представленная схема может уточ­няться до произвольной степени за счет добавления специализированных обра­ботчиков, но в любом случае объем нового кода должен быть минимальным. Еще лучше изолировать код, генерирующий исключения, в блоке try, и написать обра­ботчики для перевода исключений в существующую схему обработки ошибок.

Очень важно помнить об исключениях при создании библиотеки, которой бу­дут пользоваться другие — особенно если вы не знаете, как им потребуется реаги­ровать на критические ошибки (вспомните, что уже говорилось о безопасности исключений и о том, почему в стандартной библиотеке C++ отсутствуют специ­фикации исключений).

Типичные применения исключений

Используйте исключения для решения следующих задач:

* проблемы с повторным вызовом функции, в которой произошло исключе­ние;
* восстановление и продолжение программы без повторного вызова функ­ции;
* выполнение всех возможных действий в текущем контексте и перезапуск *того же* исключения в контексте следующего уровня;
* выполнение всех возможных действий в текущем контексте и запуск *друго­го* исключения в контекст следующего уровня;
* завершение программы;
* упрощение программы. Громоздкие и неудобные системы обработки оши­бок нередко усложняют работу программ. Возможно, исключения упростят обработку ошибок и повысят ее эффективность;
* повышение надежности библиотеки и программы. Выигрыш достигается как краткосрочный (отладка), так и долгосрочный (повышение устойчивости приложения).

Когда использовать спецификации исключений

*Спецификация исключений* похожа на прототип функции: она указывает пользо­вателю, что он должен написать код обработки ошибок, и сообщает, какие исключе­ния нужно обрабатывать. По спецификации компилятор узнает, какие исключения могут быть сгенерированы функцией, чтобы обнаружить возможные нарушения на стадии выполнения.

Простой просмотр кода не всегда позволяет определить, какие исключения могут быть запущены той или иной функцией. Иногда вызываемая функция по­рождает непредвиденные исключения, а иногда старая функция, не запускавшая исключений, заменяется новой функцией с исключениями.  
Не стоит использовать спецификации исключений в шаблонных классах, поскольку у вас нет информации о том, какие типы исклю­чений могут запускаться классами-параметрами.

Начинайте со стандартных исключений

Прежде чем создавать собственные исключения, ознакомьтесь с исключениями стандартной библиотеки C++. Если стандартное исключение делает то, что вам нужно, или примерно то, скорее всего, будет проще и удобнее работать с ним.

Если нужный тип исключения отсутствует в стандартной библиотеке, попро­буйте определить его наследованием от одного из существующих исключений.

Вложение специализированных исключений

Если вы создаете исключения для своего конкретного класса, желательно вложить классы исключений либо внутрь класса, либо внутрь пространства имен, содержа­щего класс. Тем самым вы четко сообщаете читателю программы, что исключение используется только вашим классом. Кроме того, вложенные исключения не за­громождают глобальное пространство имен.

Вложенными могут быть даже исключения, производные от стандартных ис­ключений C++.

Итоги

Восстановление работы после возникновения ошибок является одной из основ­ных проблем практически в любой программе. Это восстановление особенно важ­но в C++ при написании программных компонентов, которые будут использовать­ся другими программистами. Чтобы система в целом была устойчивой к ошибкам, устойчивым должен быть каждый из ее компонентов.

Механизм обработки исключений в C++ позволяет упростить разработку боль­ших надежных программ, содержащих минимально возможный объем вспомо­гательного кода и снижающих вероятность возникновения необработанных оши­бок. Поставленная цель достигается с незначительными потерями эффективности и с незначительным влиянием на работу существующих программ.

Освоить элементарную обработку исключений совсем несложно; после освое­ния начинайте использовать исключения в своих программах как можно раньше.

Исключения принадлежат к числу тех возможностей, польза от которых весьма существенна.

Упражнение

Создайте класс, функции которого запускают исключения. Определите внут­ри него вложенный класс для использования в качестве объекта исключе­ния. При конструировании класс исключения получает один аргумент const char\*, представляющий строку описания. Создайте функцию класса, запус­кающую это исключение (укажите это в спецификации исключений функ­ции). Напишите блок try с вызовом этой функции и секцию catch, которая выводит строку описания.

#### Книги по теме

1. Брюс Эккель, Чак Эллисон – Философия С++. Практическое программирование.,   
   (Глава 1. Обработка исключений.)