Обработка ошибок

Типы ошибок

Существуют три типа ошибок в программе:

- *синтаксические* — это ошибки в имени оператора или функции, отсутствие закрывающей или открывающей кавычек и т. д., то есть ошибки в синтаксисе языка. Как правило, компилятор предупредит о наличии ошибки, а программа не будет выполняться совсем. Пример синтаксической ошибки:

```
std::cout << "Нет завершающей кавычки!;
std::cin.get();
```

При компиляции будут выведены следующие сообщения об ошибках:

```
main.cpp(6): error C2001: newline в константе
main.cpp(7): error C2143: синтаксическая ошибка: отсутствие ";"
перед "std::cin"
```

Типы ошибок

- *логические* — это ошибки в логике работы программы, которые можно выявить только по результатам работы программы. Как правило, компилятор не предупреждает о наличии ошибки, а программа будет выполняться, так как не содержит синтаксических ошибок. Такие ошибки достаточно трудно выявить. Основные ошибки в языке C++ связаны с указателями и массивами, так как компилятор не производит никакой проверки корректности указателя и не контролирует выход за границы массива. Весь контроль полностью лежит на плечах программиста;
- *ошибки времени выполнения* — это ошибки, которые возникают во время работы программы. В одних случаях ошибки времени выполнения являются следствием логических ошибок, а в других случаях причиной являются внешние события, например, нехватка оперативной памяти, отсутствие прав для записи в файл и др.

Операторы try…catch и throw

Некоторые операторы стандарной библиотеки в случае ошибки генерируют *исключения*. Например, исключение может генерироваться оператором new, при невозможности выделении динамической памяти. Если в исходном коде не предусмотрена обработка исключения, то программа аварийно прерывается. Обработать исключение в программе позволяет оператор try...catch. Формат оператора:

Операторы try…catch и throw

```
try {
    <Блок, в котором перехватывается исключение>
}
catch (<Тип исключения1> <Переменная>) {
    <Блок, выполняемый при возникновении исключения>
}
...
catch (<Тип исключенияN> <Переменная>) {
    <Блок, выполняемый при возникновении исключения>
}
catch (...) {
    <Блок, который выполняется, если предыдущие блоки catch
     не соответствуют типу исключения>
}
```

Операторы try…catch и throw

Инструкции, в которых перехватываются исключения, должны быть расположены внутри блока try. Если при выполнении этих инструкций возникнет исключение, то управление будет передано в блок catch, который соответствует типу исключения. Типом исключения может выступать встроенный тип или пользовательский класс. Обратите внимание на то, что если исключение не возникло, то инструкции внутри блока catch не выполняются. В качестве примера сгенерируем исключение типа int и обработаем его:

```
try {
    throw 10;     // Генерируем исключение типа int
    std::cout << "Эта инструкция не будет выполнена!!!";
}
catch (int x) { // Обработка исключения типа int
    std::cout << "x = " << x << std::endl; // x = 10
}
```

Операторы try…catch и throw

В этом примере исключение искусственно генерируется с помощью оператора `throw`. После оператора указывается объект исключения. Тип этого объекта становится типом исключения, а само значение доступно через переменную, заданную в операторе `catch`. При возникновении исключения внутри блока `try` управление сразу передается в соответствующий блок `catch`. Таким образом, код, расположенный после инструкции, сгенерировавшей исключение, выполнен не будет. После выполнения инструкций в блоке `catch` управление передается инструкции, расположенной сразу после оператора `try...catch`. Иными словами, считается, что исключение обработано и можно продолжить выполнение программы.

Операторы try…catch и throw

В некоторых случаях необходимо не продолжить выполнение программы, а прервать ее выполнение. Например, при нехватке памяти не имеет смысла продолжать работу. В этом случае можно внутри блока `catch` произвести завершающие действия (закрыть файл, освободить динамическую память и т. д.), а затем прервать работу программы с помощью функции `exit()` или `abort()`. Однако, в больших программах внутри блока `catch` не всегда имеется доступ ко всем указателям, хранящим адреса динамически выделенной памяти, так как область видимости переменных ограничена блоком. Если прервать выполнение программы, то эта динамическая память освобождена не будет. Чтобы сообщить остальным частям программы об исключении, следует повторно сгенерировать исключение внутри блока `catch`, указав оператор `throw` без параметра.

Операторы try…catch и throw

Пример:

```
try {
   try {
      throw 10;     // Генерируем исключение типа int
   }
   catch (int x) { // Обработка исключения типа int
      std::cout << "x = " << x << std::endl; // x = 10
      throw;        // Повторно генерируем исключение
   }
   std::cout << "Эта инструкция не будет выполнена!!!";
}
catch (int x) {
   std::cout << "x = " << x << std::endl; // x = 10
}
```

Операторы try…catch и throw

Как видно из примера, один обработчик исключения можно вложить в другой. При генерации исключения внутри вложенного блока `try` вначале управление передается во вложенный блок `catch`. Внутри этого блока выводится сообщение, а затем исключение генерируется повторно с тем же самым типом и значением. В этом случае исключение "всплывает" к обработчику более высокого уровня и управление передается внешнему блоку `catch`, минуя остальную часть программы. Если в коде нет других обработчиков, то программа аварийно прерывается.

Операторы try…catch и throw

Внутри оператора try...catch можно указывать несколько блоков catch с разными типами исключений. При возникновении исключения производится проверка типа в первом блоке catch. Если тип исключения соответствует, то управление передается в этот блок, а остальные блоки catch игнорируются. Если тип не соответствует, то производится проверка типа в следующем блоке catch и т. д. Если ни один блок catch не соответствует исключению, то исключение "всплывает" к обработчику более высокого уровня. При отсутствии обработчика программа аварийно завершается.

Операторы try…catch и throw

Пример использования нескольких блоков catch:

```
try {
    throw 10.5;     // Генерируем исключение типа double
}
catch (int x) {
    // Этот блок пропускается
    std::cout << "int x = " << x << std::endl;
}
catch (double x) {
    // Управление передается этому блоку
    std::cout << "double x = " << x << std::endl; // double x = 10.5
}
catch (char ch) {
    // Этот блок игнорируется
    std::cout << "char ch = " << ch << std::endl;
}
```

Операторы try…catch и throw

Если в блоке catch внутри круглых скобок указаны три точки вместо типа, то такой блок перехватывает все исключения. Обычно блок по умолчанию размещают после всех остальных блоков catch. Пример:

```
try {
   throw 'A';     // Генерируем исключение типа char
}
catch (int x) {
   // Этот блок пропускается
   std::cout << "int x = " << x << std::endl;
}
catch (...) {
   // Управление передается этому блоку
   std::cout << "catch (...)" << std::endl; // catch (...)
}
catch (double x) {
   // Этот блок никогда не будет выполнен!!!
}
```

Класс exception

В заголовочном файле exception объявлен класс exception, который наследуют некоторые стандартные классы исключений, например, bad_exception, bad_alloc и др. Класс имеет четыре конструктора:

```
exception();
explicit exception(const char * const &);
exception(const char * const &, int);
exception(const exception&);
```

Класс exception

Если параметр в конструкторе не задан, то текст сообщения выглядит так: "Unknown exception". С помощью второго и третьего конструктора можно изменить текст сообщения. Получить сообщение об ошибке позволяет метод what(). Прототип метода:

```
virtual const char *what() const;
```

В качестве примера сгенерируем исключение класса exception и обработаем его с помощью оператора try...catch

Класс exception

```
#include <iostream>
#include <exception>

int main() {
   try {
      std::exception err("my_error");  // Создаем объект
      throw err;                       // Генерируем исключение
   }
   catch (std::exception &err) {
      std::cout << err.what() << std::endl; // my_error
   }
   std::exception err1;
   std::cout << err1.what() << std::endl;    // Unknown exception
   std::exception err2("my_error");
   std::cout << err2.what() << std::endl;    // my_error
   std::cin.get();
   return 0;
}
```

Пользовательские классы исключений

Как вы уже знаете, типом исключения может выступать встроенный тип или пользовательский класс. Основное преимущество использования классов для обработки исключений заключается в возможности указания базового класса для перехвата всех исключений соответствующих классов-потомков. Например, если пользовательский класс наследует стандартный класс exception, то, указав в блоке catch объект класса exception, можно перехватить исключение пользовательского класса. Обратите внимание на то, что блок catch, в котором указан объект производного класса, должен быть расположен перед блоком catch, в котором указан объект базового класса.

Пользовательские классы исключений

При разработке библиотек обычно создается базовый класс, являющийся наследником класса exception, а все остальные классы исключений внутри библиотеки наследуют этот базовый класс. Таким образом, разработчик библиотеки получает возможность создания новых классов исключений, создавая иерархию классов. Пользователю в этом случае достаточно указать объект базового класса в блоке catch, чтобы перехватить все исключения, генерируемые внутри библиотеки.

ЛИСТИНГ. Создание иерархии пользовательских классов исключений

```
#include <iostream>
#include <exception>

// Базовый класс
class MyExceptionBase : public std::exception {
public:
   MyExceptionBase() : exception("Error") { }
   MyExceptionBase(const char *message) : exception(message)
   { }
   virtual ~MyExceptionBase() { }
};
```

ЛИСТИНГ. Создание иерархии пользовательских классов исключений

```
// Производные классы от MyExceptionBase
class MyException1 : public MyExceptionBase {
public:
   MyException1() : MyExceptionBase("Error") { }
   MyException1(const char *message) : MyExceptionBase(message)
   { }
   virtual ~MyException1() { }
};
class MyException2 : public MyExceptionBase {
public:
   MyException2() : MyExceptionBase("Error") { }
   MyException2(const char *message) : MyExceptionBase(message)
   { }
   virtual ~MyException2() { }
};
```

ЛИСТИНГ. Создание иерархии пользовательских классов исключений

```
int main() {
   try {
      MyException2 err("my_error");      // Создаем объект
      throw err;                         // Генерируем исключение
   }
   catch (MyException1 &err) {
      // Производный класс исключения должен обрабатываться
      // раньше, чем базовый класс!!!
      std::cout << "MyException2: " << err.what() << std::endl;
   }
   catch (MyExceptionBase &err) {
      std::cout << "MyExceptionBase: " << err.what() << std::endl;
   } // Результат: MyExceptionBase: my_error
   std::cin.get();
   return 0;
}
```

Иерархия наследования классов исключений

СОВЕТ

Если имеется иерархия наследования классов исключений, необходимо расположить блоки catch в таком порядке чтобы исключение самого последнего производного класса перехватывалось первым, а исключение базового класса — последним.

Исключения, классы и наследование

Исключения, классы и наследование взаимодействуют несколькими способами.
Во-первых, можно породить один класс исключения от другого класса, как это сделано в стандартной библиотеке C++.
Во-вторых, можно добавить исключения в классы, вставив объявление класса исключения в определение класса.
В-третьих, такое вложенное объявление может быть унаследовано и само служить базовым классом.

Ограничение типа исключений, генерируемых внутри функции

В объявлении функции после параметров можно указать спецификацию, ограничивающую типы исключений, которые допускается генерировать внутри функции. Спецификация имеет следующий синтаксис:

```
<Тип> <Название функции>([<Параметры>]) throw([<Типы исключений>]) {
   // Тело функции
}
```

Ограничение типа исключений, генерируемых внутри функции

Внутри инструкции throw() могут быть указаны допустимые типы исключений через запятую. Если типы исключений не заданы, то функция не должна генерировать исключения вообще. Если вместо типов указаны три точки, то функция может сгенерировать любое исключение. Пример:

```
void func() throw();     // Функция не должна генерировать исключение
void func() throw(...); // Функция может генерировать любое исключение
void func() throw(int, double); // Допустимы только типы int и double
```

Ограничение типа исключений, генерируемых внутри функции

Обратите внимание на то, что в VC++ спецификации исключений являются допустимыми, но не реализованы. Попытка указать внутри круглых скобок конкретные типы приводит к выводу предупреждающего сообщения с номером C4290. Чтобы предотвратить вывод предупреждающего сообщения следует вначале программы вставить следующую инструкцию:

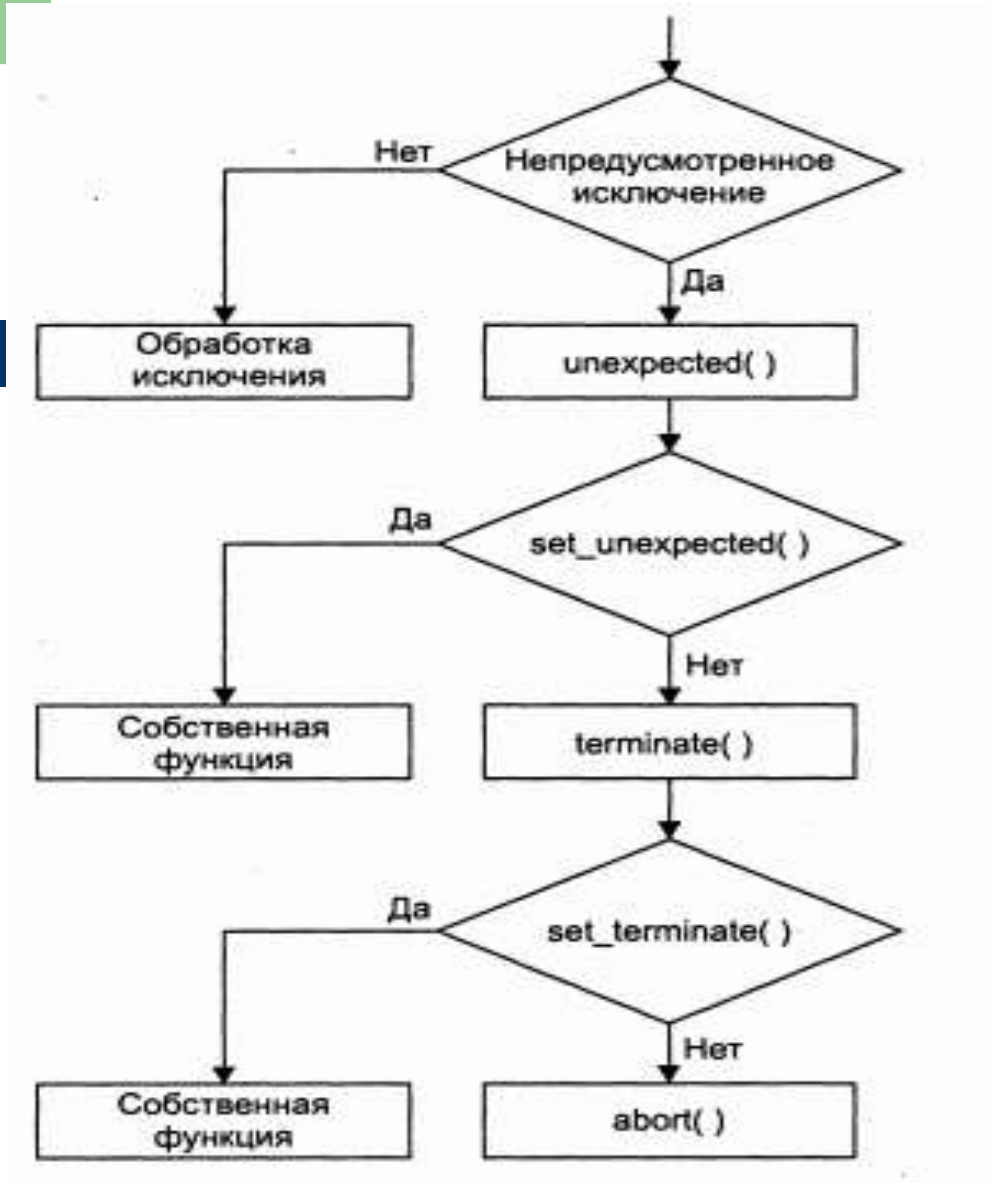
```
#pragma warning( disable: 4290 )
```

Потеря исключений

После того как исключение сгенерировано, у него есть две возможности вызвать проблемы. Если исключение сгенерировано в функции, имеющей спецификацию исключения, оно должно соответствовать одному из типов в списке спецификации. Если исключение не соответствует спецификации, оно называется *непредвиденным исключением* **и по умолчанию приводит к останову программы. (Хотя в C++11 спецификации исключений объявлены устаревшими, они по-прежнему остались в языке и кое-где в существующем коде.)**

Потеря исключений

Если исключение преодолевает этот первый барьер (или избегает его, в силу отсутствия спецификации исключения), то оно должно быть перехвачено. Если исключение не перехвачено, что может произойти при отсутствии блока try или соответствующего блока catch , оно называется *неперехваченным исключением.* **По умолчанию такое исключение приводит к останову программы. Однако можно изменить реакцию программы на непредвиденные и неперехваченные исключения.**



Алгоритм обработки исключения