Урок 8

Механизм исключительных ситуаций

Возбуждение и обработка ситуаций. Свертка стека. Исполнение конструкторов и деструкторов. Поддержка иерархии классов. Стандартные классы исключительных ситуаций. Примеры программ с использованием исключительных ситуаций.

Введение в исключения

Встречаются ситуации, которые препятствуют дальнейшей нормальной работе программ. К примеру, при делении числа на ноль программа просто закроется — несмотря на то, сколько пользователь работал в ней и какой объем данных внес. И представьте: пользователь вносил в программу данные несколько часов и проводил расчеты, а при аварийном закрытии все они пропадут.

Или другой случай: программа пытается открыть недоступный в этот момент файл или запросить больше памяти, чем доступно. Такого рода ситуации программистам надо стараться предвидеть и строить программы так, чтобы они могли гибко реагировать, а не аварийно закрываться.

Исключение — это реакция на нештатную ситуацию, возникшую на время выполнения программы. Исключения позволяют передать управление из одной части программы в другую.

Исключения в C++ реализованы с помощью трех ключевых слов, которые работают в связке друг с другом: **throw**, **try** и **catch**.

Обработка исключений

В C++ **оператор throw** используется, чтобы сигнализировать о возникновении исключения или ошибки — это называется **генерацией исключения**.

Для применения оператора **throw** используется ключевое слово **throw**, за которым указывается значение любого типа данных, которое вы хотите использовать, чтобы сигнализировать об ошибке. Как правило, этим значением является код ошибки, описание проблемы или настраиваемый класс-исключение (класс **Exception**). Например:

```
throw -1;
// генерация исключения типа int

throw ENUM_INVALID_INDEX;
// генерация исключения типа enum

throw "Cannot take square root of negative number";
// генерация исключения типа const char* (строка C-style)

throw dX;
// генерация исключения типа double (переменная типа double, которая была
определена ранее)

throw MyException("Fatal Error");
// генерация исключения с использованием объекта класса MyException
```

В C++ мы используем ключевое слово **try** для определения блока операторов и команд — так называемый **блок try**. Он действует как наблюдатель, отслеживая возникновение исключений, которые были выброшены каким-либо оператором в этом же блоке **try**.

Например:

```
try
{
                // Здесь мы пишем стейтменты, которые будут генерировать
следующее исключение
    throw -1; // типичный стейтмент throw
}
```

Фактически, обработка исключений — это работа блоков **catch**. Ключевое слово **catch** используется для определения блока кода (**блока catch**), который обрабатывает исключения определенного типа данных.

Вот пример блока **catch**, который обрабатывает (ловит) исключения типа **int**:

```
catch (int a)
{
    // Обрабатываем исключение типа int
    cerr << "We caught an int exception with value" << a << '\n';
}
```

Блоки **try** и **catch** работают вместе. Блок **try** обнаруживает любые исключения, которые были выброшены в нем, и направляет их в соответствующий блок **catch** для обработки. Блок **try** должен иметь по крайней мере один блок **catch**, который находится сразу за ним, но также может иметь и несколько блоков **catch**, размещенных последовательно, друг за другом.

Как только исключение было поймано блоком **try** и направлено в блок **catch** для обработки, оно считается обработанным (после выполнения кода блока **catch**), и выполнение программы возобновляется.

Параметры **catch** работают так же, как и параметры функции, причем параметры одного блока **catch** могут быть доступны и в другом блоке **catch**, который находится за ним. Исключения фундаментальных типов данных могут быть пойманы по значению (параметром блока **catch** является значение). Но исключения не фундаментальных типов данных должны быть пойманы по константной ссылке (параметром блока **catch** является константная ссылка), чтобы избежать ненужного копирования.

Если исключение направлено в блок **catch**, оно считается обработанным, даже если блок **catch** пуст. Но хочется, чтобы ваши блоки **catch** делали что-то полезное. Есть три распространенные вещи, которые выполняют блоки **catch**, когда они поймали исключение:

1. Выводят сообщение об ошибке.
2. Возвращают значение или код ошибки обратно в **caller** (тот оператор, который вызвал исполняемый блок).
3. Генерируют другое исключения. Поскольку блок **catch** не находится внутри блока **try**, новое сгенерированное исключение будет обрабатываться следующим блоком **try**.

Генерация исключений за пределами блока try

Благодаря выполнению операции раскручивания стека операторы **throw** вовсе не обязаны находиться непосредственно в блоке **try**. Это дает нам необходимую гибкость в разделении общего потока выполнения кода программы и обработки исключений. Продемонстрируем это, написав программу вычисления квадратного корня:

```
#include <cmath> // для sqrt()
#include <iostream>
using namespace std;

// Отдельная функция вычисления квадратного корня
double mySqrt(double a)
{
    // Если пользователь ввёл отрицательное число, то выбрасываем исключение
    if (a < 0.0)
        throw "Can not take sqrt of negative number"; // выбрасывается
исключение типа const char*

    return sqrt(a);
}

int main()
{
    cout << "Enter a number: ";
    double a;
    cin >> a;

    try // ищем исключения, которые выбрасываются в блоке try, и отправляем их
для обработки в блок(и) catch
    {
        double d = mySqrt(a);
        cout << "The sqrt of " << a << " is " << d << '\n';
    }
    catch (const char* exception) // обработка исключений типа const char*
    {
        cerr << "Error: " << exception << endl;
    }

    return 0;
}
```

Результат работы программы:

Enter a number: -3
Error: Cannot take sqrt of negative number

При генерации исключения компилятор смотрит, можно ли сразу же обработать его. Поскольку точка выполнения не находится внутри блока **try**, то и обработать исключение немедленно не получится. Выполнение функции **mySqrt()** приостанавливается, и программа смотрит, может ли **caller** (который и вызывает **mySqrt()**) обработать это исключение. Если нет, то компилятор завершает выполнение **caller**'а и переходит на уровень выше: к **caller**'у, который вызывает текущего **caller**'а, чтобы проверить, сможет ли тот обработать исключение. И так последовательно до тех пор, пока не будет найден соответствующий обработчик исключения или **main()** не завершится без обработки исключения. Этот процесс называется **раскручиванием стека**.

Теперь рассмотрим детальнее. Сначала компилятор проверяет, генерируется ли исключение внутри блока **try**. В нашем случае — нет, поэтому стек начинает раскручиваться. При этом функция **mySqrt()** завершает свое выполнение, и точка выполнения перемещается обратно в **main()**. Теперь компилятор проверяет снова, находимся ли мы внутри блока **try**. Поскольку вызов **mySqrt()** был выполнен из блока **try**, то компилятор начинает искать соответствующий обработчик **catch**. Он находит обработчик типа **const char***, и исключение обрабатывается блоком **catch** внутри **main()**.

Передача обработки исключения в **caller** необходима потому, что программы обрабатывают ошибки/исключения по-разному. Консольная программа выводит сообщение об ошибке, а приложение Windows — диалоговое окно с ошибкой.

Рассмотрим пример посложнее:

```
#include <iostream>
using namespace std;

void last() // вызывается функцией three()
{
    cout << "Start last\n";
    cout << "last throwing int exception\n";
    throw -1;
    cout << "End last\n";

}

void three() // вызывается функцией two()
{
    cout << "Start three\n";
    last();
    cout << "End three\n";
}

void two() // вызывается функцией one()
{
    cout << "Start two\n";
    try
    {
        three();
    }
    catch(double)
    {
        cerr << "two caught double exception\n";
    }
    cout << "End two\n";
}

void one() // вызывается функцией main()
{
    cout << "Start one\n";
    try
    {
        two();
    }
    catch (int)
    {
        cerr << "one caught int exception\n";
    }
    catch (double)
    {
        cerr << "one caught double exception\n";
    }
```

```
    cout << "End one\n";
}

int main()
{
    cout << "Start main\n";
    try
    {
        one();
    }
    catch (int)
    {
        cerr << "main caught int exception\n";
    }
    cout << "End main\n";

    return 0;
}
int main()
{
    cout << "Start main\n";
    try
    {
        one();
    }
    catch (int)
    {
        cerr << "main caught int exception\n";
    }
    cout << "End main\n";

    return 0;
}
```

Результат работы программы:

Start main
Start one
Start two
Start three
Start last
last throwing int exception
one caught int exception
End one
End main

Поскольку **last()** не обрабатывает исключения самостоятельно, стек начинает раскручиваться. Функция **last()** немедленно завершает свое выполнение, и точка выполнения возвращается обратно в **caller** (в функцию **three()**).

Функция **three()** не обрабатывает исключения, поэтому стек раскручивается дальше, выполнение функции **three()** прекращается и точка выполнения возвращается в **two()**.

Функция **two()** имеет блок **try**, в котором находится вызов **three()**, поэтому компилятор пытается найти обработчик исключений типа **int**, но так как его не находит, точка выполнения возвращается в **one()**. Обратите внимание: компилятор не выполняет неявного преобразования, чтобы сопоставить исключение типа **int** с обработчиком типа **double**.

Функция **one()** также имеет блок **try** с вызовом **two()** внутри, поэтому компилятор смотрит, есть ли подходящий обработчик **catch**. Есть! Функция **one()** обрабатывает исключение и выводит **one caught int exception**.

Поскольку исключение было обработано, точка выполнения перемещается в конец блока **catch** внутри **one()**. Это означает, что **one()** выводит **End one**, а затем завершает свое выполнение как обычно.

Точка выполнения возвращается обратно в **main()**. Хотя **main()** имеет обработчик исключений типа **int**, наше исключение уже было обработано функцией **one()**, поэтому блок **catch** внутри **main()** не выполняется. **main()** выводит **End main**, а затем завершает свое выполнение.

Раскручивание стека — очень полезный механизм, так как позволяет функциям не обрабатывать исключения, если они этого не хотят. Раскручивание выполняется до тех пор, пока не будет обнаружен соответствующий блок **catch**! Таким образом мы можем сами решать, где следует обрабатывать исключения.

Обработчик catch-all

Функции могут генерировать исключения любого типа данных, и если исключение не поймано, это приведет к раскручиванию стека и потенциальному завершению выполнения целой программы. C++ предоставляет механизм обнаружения/обработки всех типов исключений — **обработчик catch-all**. Он работает так же, как и обычный блок **catch**, но вместо обработки исключений определенного типа данных использует эллипсис (…) в качестве типа данных.

Вот простой пример:

```
#include <iostream>
using namespace std;

int main()
{
    try
    {
        throw 7; // выбрасывается исключение типа int
    }
    catch (double a)
    {
        cout << "We caught an exception of type double: " << a << '\n';
    }
    catch (...) // обработчик catch-all
    {
        cout << "We caught an exception of an undetermined type!\n";
    }
}
```

Поскольку для типа **int** не существует специального обработчика **catch**, обработчик **catch-all** ловит это исключение. Следовательно, результат:

We caught an exception of an undetermined type!

Обработчик **catch-all** должен быть последним в цепочке блоков **catch**.

Класс-исключение

Класс-исключение — это обычный класс, который выбрасывается в качестве исключения. Создадим простой класс-исключение:

```
#include <string>
using namespace std;

class classException
{
private:
    string m_error;

public:
    classException(string error) : m_error(error)
    { }
    const char* getError() { return m_error.c_str(); }
};
```

Используя такой класс, мы можем генерировать исключение, возвращающее описание возникшей проблемы. Это даст точно понять, что именно пошло не так. И поскольку исключение **classException** — уникального типа, мы можем обрабатывать его соответствующим образом. В обработчиках исключений объекты класса-исключения нужно принимать по ссылке, а не по значению. Это предотвратит затратную операцию — создание копии исключения компилятором.

Исключения и наследования

Обработчики могут обрабатывать исключения не только одного определенного класса, но и исключения дочерних ему классов. Рассмотрим следующий пример:

```
#include <iostream>
using namespace std;
class Parent
{
public:
    Parent() {}
};

class Child: public Parent
{
public:
    Child() {}
};

int main()
{
    try
    {
        throw Child();
    }
    catch (Parent &parent)
    {
        cerr << "caught Parent";
    }
    catch (Child &child)
    {
        cerr << "caught Child";
    }

    return 0;
}
```

Здесь выбрасывается исключение типа **Child**. Но результат выполнения этой программы:

caught Parent

Чтобы обработчик **Child** ловил исключения класса **Child**, нужно просто поменять последовательность блоков **catch**.

Обработчики исключений дочерних классов должны находиться перед обработчиками исключений родительского класса.

std::exception

Многие классы и операторы из стандартной библиотеки C++ выбрасывают классы-исключения при сбое. Например, оператор **new** и **std::string** могут выбрасывать **std::bad_alloc** при нехватке памяти. Неудачное динамическое приведение типов с помощью **dynamic_cast** выбрасывает исключение **std::bad_cast** и так далее. Начиная с C++14, существует 21 класс-исключение, которые могут быть выброшены, в C++17 их еще больше.

Хорошая новость: все эти классы-исключения являются дочерними классу **std::exception**. Это небольшой интерфейсный класс, который используется в качестве родительского для любого исключения, которое выбрасывается в стандартной библиотеке C++.

В большинстве случаев, если исключение выбрасывается стандартной библиотекой C++, нам все равно, будет ли это неудачное выделение, конвертирование или что-либо другое. Достаточно знать, что случилось что-то катастрофическое, из-за чего в программе произошел сбой. Благодаря **std::exception** мы можем настроить обработчик исключений типа **std::exception**, который будет ловить и обрабатывать как **std::exception**, так и все (21+) дочерние ему классы-исключения!

В **std::exception** есть виртуальный метод **what()**, который возвращает строку **C-style** с описанием исключения. Большинство дочерних классов переопределяют функцию **what()**, изменяя это сообщение. Обратите внимание: эта строка **C-style** предназначена для использования только в качестве описания.

Иногда нужно обрабатывать определенный тип исключений соответствующим ему образом. В таком случае мы можем добавить обработчик исключений для этого конкретного типа, а все остальные исключения «перенаправлять» в родительский обработчик. Например:

```
try
{
    // Здесь код с использованием стандартной библиотеки C++
}
// Этот обработчик ловит bad_alloc и все дочерние ему классы-исключения
catch (bad_alloc &exception)
{
    cerr << "You ran out of memory!" << '\n';
    }
    // Этот обработчик ловит exception и все дочерние ему классы-исключения
    catch (exception &exception)
    {
        cerr << "Standard exception: " << exception.what() << '\n';
    }
}
```

В этом примере исключения типа **std::bad_alloc** перехватываются и обрабатываются первым обработчиком. Исключения типа **std::exception** и всех других дочерних ему классов-исключений обрабатываются вторым обработчиком.

Повторная генерация исключений

Бывает, что нужно поймать исключение, но обрабатывать его сразу не хочется или нет возможности. Тогда вы можете записать ошибку в лог-файл, а затем передать ее обратно в **caller** для выполнения фактической обработки. Есть и другой вариант — генерация нового исключения. Можно в блоке **catch** сгенерировать новое исключение в блоке **throw**. Помните, что только исключения, сгенерированные в блоке **try**, могут быть перехвачены блоком **catch**. Это означает, что исключение, сгенерированное в блоке **catch**, не будет перехвачено этим же блоком **catch**, в котором оно находится. Вместо этого стек начнет раскручиваться и исключение будет передано caller'у, который находится на уровне выше в стеке вызовов.

C++ предоставляет способ повторной генерации исключения. Для этого нужно просто использовать ключевое слово **throw** внутри блока **catch** без указания идентификатора. Например:

```
#include <iostream>
using namespace std;

class Parent
{
public:
    Parent() {}
    virtual void print() { cout << "Parent"; }
};

class Child: public Parent
{
public:
    Child() {}
    virtual void print() { cout << "Child"; }
};

int main()
{
    try
    {
        try
        {
            throw Child();
        }
        catch (Parent& p)
        {
            cout << "Caught Parent p, which is actually a ";
            p.print();
            cout << "\n";
            throw; // Мы здесь повторно выбрасываем исключение
        }
    }
    catch (Parent& p)
    {
        cout << "Caught Parent p, which is actually a ";
        p.print();
        cout << "\n";
    }

    return 0;
}
```

Результат выполнения программы:

Caught Parent p, which is actually a Child
Caught Parent p, which is actually a Child

Ключевое слово **throw** в блоке **catch** на первый взгляд не генерирует что-либо конкретное, а на самом деле генерирует точно такое же исключение, которое было только что обработано блоком **catch**. Не выполняется копирования исключения и, следовательно, обрезки объекта.

При повторной генерации исключения используйте ключевое слово **throw** без указания идентификатора.

Функциональный try-блок

Функциональности блоков **try** и **catch** достаточно в большинстве случаев, но есть одна ситуация, в которой это не так. Рассмотрим следующий код:

```
#include <iostream>
using namespace std;

class Parent
{
private:
    int m_age;
public:
    Parent(int age) : m_age(age)
    {
        if (age <= 0)
            throw 1;
    }
};

class Child : public Parent
{
public:
    Child(int age) : Parent(age)
    {
        // Что произойдет, если создать Parent не удастся, а исключение нужно
обрабатывать здесь?
    }
};

int main()
{
    try
    {
        Child child(0);
    }
    catch (int)
    {
        cout << "Oops!\n";
    }
}
```

В этой программе дочерний класс **Child** вызывает конструктор родительского класса **Parent**, который генерирует исключение при успешном выполнении условия. Поскольку объект **child** создается в блоке **try** функции **main()**, если **Parent** выбросит исключение, блок **try** функции **main()** поймает его и передаст обработчику **catch (int)**. Следовательно, результат выполнения этой программы будет таким:

Oops!

Но что, если нужно обрабатывать исключение внутри класса **Child**? Вызов конструктора родительского класса **Parent** происходит через список инициализации членов, перед выполнением тела конструктора класса **Child**. Поэтому использовать стандартный блок **try** здесь не получится.

В этой ситуации мы должны использовать слегка модифицированный блок **try — функциональный try-блок.**

Функциональный try-блок используется для установления обработчика исключений вокруг тела всей функции, а не ее части (блока кода). Рассмотрим на примере:

```
#include <iostream>
using namespace std;

class Parent
{
private:
    int m_age;
public:
    Parent(int age) : m_age(age)
    {
        if (age <= 0)
            throw 1;
    }
};

class Child : public Parent
{
public:
    Child(int age) try : Parent(age) // обратите внимание на ключевое слово try здесь
    {
    }
    catch (...) // Этот блок находится на том же уровне отступа, что и конструктор
    {
        // Исключения из списка инициализации членов класса Child или тела конструктора обрабатываются здесь

        cerr << "Construction of Parent failed\n";
        // Если мы здесь не будем явно выбрасывать исключение, то текущее (пойманное) исключение будет повторно сгенерировано и отправлено в стек вызовов
    }
};

int main()
{
    try
    {
        Child child(0);
    }
    catch (int)
    {
        cout << "Oops!\n";
    }
}
```

Результат выполнения программы:

Construction of Parent failed
Oops!

Рассмотрим эту программу более подробно.

1. Обратите внимание на добавление ключевого слова **try** перед списком инициализации членов класса класса **Child**. Это означает, что все, что находится после этого ключевого слова (вплоть до конца функции), рассматривается как часть блока **try**.

2. Блок **catch** находится на том же уровне отступа, что и вся функция. Любое исключение, выброшенное между ключевым словом **try** и концом тела конструктора, будет обработано этим же блоком **catch**.

3. Обычные блоки **catch** либо обрабатывают исключения, либо выбрасывают новое, либо повторно генерируют пойманное исключение. Но при использовании функциональных блоков **try** вы должны либо выбросить новое исключение, либо повторно сгенерировать пойманное. Если этого не сделать, пойманное исключение будет повторно сгенерировано и стек начнет раскручиваться.

Так как в программе выше мы явно не генерируем исключение внутри блока **catch**, исключение повторно генерируется и передается caller'у на уровень выше, то есть функции **main()**. Блок **catch** функции **main()** ловит и обрабатывает исключение.

Функциональные try-блоки также могут использоваться и с обычными функциями, которые не являются методами класса. Но это не распространенная практика, так как случаев, где они могут быть полезны, очень мало. Они почти всегда используются только с конструкторами!

Недостатки и опасности использования исключений

Начинающие программисты, используя исключения, сталкиваются с проблемой очистки выделенных ресурсов после генерации исключения. Рассмотрим следующий пример:

```
try
{
    openFile(filename);
    writeFile(filename, data);
    closeFile(filename);
}
catch (FileException &exception)
{
    cerr << "Failed to write to file: " << exception.what() << endl;
}
```

Что произойдет, если **writeFile()** не сработает и выбросит объект класса-ислючения **FileException**? К этому моменту мы уже открыли файл, и точка выполнения перейдет к обработчику **catch**, который выведет ошибку и завершит свое выполнение. Обратите внимание: операция закрытия файла **closeFile(filename)** никогда не выполнится! Чтобы этого избежать, нужно добавить перед выводом сообщения об ошибке команду на закрытие файла.

Такой же тип ошибок возникает и при выделении динамической памяти.

В отличие от конструкторов, где генерация исключений может быть полезным способом указать, что создать объект не удалось, исключения *никогда* не должны генерироваться в деструкторах.

Проблема возникает, когда исключение генерируется в деструкторе во время раскручивания стека. Если это происходит, компилятор оказывается в ситуации, когда он не знает, продолжать ли процесс

раскручивания стека или обработать новое исключение. Программа немедленно прекратит выполнение.

Лучше вообще воздержаться от использования исключений в деструкторах. Лучше вместо этого записать ошибку в лог-файл.

У производительности исключений есть своя небольшая цена. Они увеличивают размер исполняемого файла и могут заставить его выполняться медленнее из-за дополнительной проверки. Тем не менее основное снижение производительности происходит при выбрасывании исключения. В этот момент стек начинает раскручиваться и выполняется поиск соответствующего обработчика исключений, что само по себе затратная операция.

Исключения и их обработку лучше всего использовать, если выполняются все следующие условия:

1. Обрабатываемая ошибка возникает редко.
2. Ошибка является серьезной, и выполнение программы не может продолжаться без ее обработки.
3. Ошибка не может быть обработана в том месте, где она возникает.
4. Нет хорошего альтернативного способа вернуть код ошибки обратно в caller.

Практическое задание

1. Написать шаблонную функцию **div**, которая должна вычислять результат деления двух параметров и запускать исключение **DivisionByZero**, если второй параметр равен 0. В функции **main** выводить результат вызова функции **div** в консоль, а также ловить исключения.
2. Написать класс **Ex**, хранящий вещественное число **x** и имеющий конструктор по вещественному числу, инициализирующий **x** значением параметра. Написать класс **Bar**, хранящий вещественное число **y** (конструктор по умолчанию инициализирует его нулем) и имеющий метод **set** с единственным вещественным параметром **a**. Если **y + a > 100**, возбуждается исключение типа Ex с данными **a*y**, иначе в **y** заносится значение **a**. В функции **main** завести переменную класса **Bar** и в цикле в блоке **try** вводить с клавиатуры целое **n**. Использовать его в качестве параметра метода **set** до тех пор, пока не будет введено 0. В обработчике исключения выводить сообщение об ошибке, содержащее данные объекта исключения.
3. Написать класс «робот», моделирующий перемещения робота по сетке 10x10, у которого есть метод, означающий задание переместиться на соседнюю позицию. Эти методы должны запускать классы-исключения **OffTheField**, если робот должен уйти с сетки, и **IllegalCommand**, если подана неверная команда (направление не находится в нужном диапазоне). Объект исключения должен содержать всю необходимую информацию — текущую позицию и направление движения. Написать функцию **main**, пользующуюся этим классом и перехватывающую все исключения от его методов, а также выводящую подробную информацию о всех возникающих ошибках.

Дополнительные материалы

1. Бьерн Страуструп. Программирование. Принципы и практика использования C++.
2. Стивен Прата. Язык программирования C++. Лекции и упражнения.
3. Роберт Лафоре. Объектно-ориентированное программирование в C++.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Исключения в C++ (exception).
2. Бьерн Страуструп. Программирование. Принципы и практика использования C++.
3. Ральф Джонсон, Ричард Хелм, Эрих Гамма. Приемы объектно-ориентированного программирования. Паттерны проектирования.