Севастопольский государственный университет
Кафедра информационных систем

Курс лекций по дисциплине

«ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ"

(ООП)

Лектора: Пелипас Всеволод Олегович
Сметанина Татьяна Ивановна

Лекция 9

ИСКЛЮЧЕНИЯ

Исключения

Исключительная ситуация, или и**сключение** — это возникновение непредвиденного или аварийного события, которое может порождаться некорректным использованием аппаратуры. Например, это деление на ноль или обращение по несуществующему адресу памяти. Обычно эти события приводят к завершению программы с системным сообщением об ошибке. C++ дает программисту возможность восстанавливать программу и продолжать ее выполнение.

Исключения C++ не поддерживают обработку асинхронных событий, таких, как ошибки оборудования или обработку прерываний, например, нажатие клавиш Ctrl+C. Механизм исключений предназначен только для событий, которые происходят в результате работы самой программы и указываются явным образом.

Исключения возникают тогда, когда некоторая часть программы не смогла сделать то, что от нее требовалось. При этом другая часть программы может попытаться сделать что-нибудь иное.

Достоинства

Исключения позволяют логически разделить вычислительный процесс на две части — обнаружение аварийной ситуации и ее обработку.

Функция, обнаружившая ошибку, может не знать, что предпринимать для ее исправления, а использующий эту функцию код может знать, что делать, но не уметь определить место возникновения. Это особенно актуально при использовании библиотечных функций и программ, состоящих из многих модулей.

Другое достоинство исключений состоит в том, что для передачи информации об ошибке в вызывающую функцию не требуется применять возвращаемое значение, параметры или глобальные переменные, поэтому интерфейс функций не раздувается (это особенно важно, например, для конструкторов, которые по синтаксису не могут возвращать значение).

Пример

```
int main(){
int num1, num2;
        cout << "Введите num1: ";        cin >> num1;
        cout << "Введите num2: ";        cin >> num2;
           cout << "num1 / num2 = ";
         try //код, который может привести к ошибке
        {
           if (num2 == 0)
           {  throw 123; //генерировать целое число 123
           }
           cout << num1 / num2 << endl;
        }
     catch(int i)//сюда передастся число 123
         { cout << "Ошибка №" << i << " - на 0 делить нельзя!!!!";
         }
  /*…*/ return 0;
}
```

Пример

В **try**-блоке располагается код, который потенциально может вызвать ошибку в работе программы, а именно ошибку в случае деления на 0. Задаем условие **if** - если **num2** равно 0, то сгенерировать целое число 123 (например). В этом случае **try**-блок сразу прекращает выполнение дальнейших команд, а число 123 «падает» в **catch**. В нашем примере он выводит сообщение об ошибке. При этом программа продолжает работать и выполнять команды, размещенные ниже. Если же число **num2** не будет равно нулю, то в **try**-блоке выполнится команда **cout << num1 / num2 << endl;** , а **catch** не сработает.

Запустим программу и в первый раз введем значение переменной **num2** равное 0, а во второй — любое другое число:

```
Введите значение num1: 5
Введите значение num2: 0
num1 / num2 = Ошибка №123 - на 0 делить нельзя!!!!
```

```
Введите значение num1: 25
Введите значение num2: 5
num1 / num2 = 5
```

Исключения

try (пытаться) — блок исключений, в нем располагается код, который может привести к ошибке и аварийному закрытию программы;

throw (бросить) генерирует исключение, которое остановит работу **try**-блока и приведет к выполнению кода **catch-блока**. **Тип исключения** должен соответствовать, **типу принимаемого аргумента** **catch**-блока;

catch (поймать) - улавливающий блок, ловит то, что определил **throw** и выполняет свой код. Этот блок должен располагаться непосредственно под **try**-блоком, никакой код не должен их разделять.

Если в **try**-блоке исключение не генерируется, **catch**-блок не срабатывает и программа его обходит.

Синтаксис исключений

Ключевое слово **try** служит для обозначения контролируемого блока — кода, в котором может генерироваться исключение. Блок заключается в фигурные скобки:

try{ }

Все функции, прямо или косвенно вызываемые из **try**-блока, также считаются ему принадлежащими.

Генерация (порождение) исключения происходит по ключевому слову **throw**, которое употребляется либо с параметром, либо без него:

throw [выражение];

Тип выражения, стоящего после **throw**, определяет тип порождаемого исключения. При генерации исключения выполнение текущего блока прекращается, и происходит поиск соответствующего обработчика и передача ему управления. Как правило, исключение генерируется не непосредственно в **try**-блоке, а в функциях, прямо или косвенно в него вложенных.

Синтаксис исключений

Не всегда исключение, возникшее во внутреннем блоке, может быть сразу правильно обработано. В этом случае используются вложенные контролируемые блоки, и исключение передается на более высокий уровень с помощью ключевого слова **throw** без параметров.

Обработчики исключений начинаются с ключевого слова **catch**, за которым в скобках следует тип обрабатываемого исключения. Они должны располагаться непосредственно за **try**-блоком. Можно записать один или несколько обработчиков в соответствии с типами обрабатываемых исключений. Синтаксис обработчиков напоминает определение функции с одним параметром — типом исключения. Существует три формы записи:

catch(тип имя){ ... /* тело обработчика */ }
catch(тип){ ... /* тело обработчика */ }
catch(...){ .. /* тело обработчика */ }

Синтаксис исключений

catch(тип имя){ ... /* тело обработчика */ }

Первая форма применяется, когда имя параметра используется в теле обработчика для выполнения каких-либо действий — например, вывода информации об исключении.

catch(тип){ ... /* тело обработчика */ }

Вторая форма не предполагает использования информации об исключении, играет роль только его тип.

catch(...){ .. /* тело обработчика */ }

Многоточие вместо параметра обозначает, что обработчик перехватывает все исключения. Так как обработчики просматриваются в том порядке, в котором они записаны, обработчик третьего типа следует помещать после всех остальных.

После обработки исключения управление передается первому оператору, находящемуся непосредственно за обработчиками исключений. Туда же, минуя код всех обработчиков, передается управление, если исключение в **try**-блоке не было сгенерировано.

Пример y =sqrt(1/x)

```
int main()
{   double x, y;
      cout << "Введите x: ";    cin >> x;
      try
      {   if (!cin)  throw '1'; //генерируется char
          if (x == 0)  throw 1; // генерируется int
          if (1/x < 0)  throw 1.01; // генерируется double
           cout << "y =sqrt(1/x) ";
                y=sqrt(1/x);
                cout << "y=" << y << endl;
      }
  catch(char) {cout << "Ошибка ввода" << endl;}
  catch(int) {cout << "Делить на 0 нельзя" << endl;}
  catch(double d) {cout << "Извлекать корень из отр. числа
нельзя d=" <<d<< endl;}
   catch(...) {cout << "Ошибка" << endl;}
return 0;
}
```

```
Введите x: A
Ошибка ввода
```

```
Введите x: 0
Делить на 0 нельзя
```

```
Введите x: -9
Извлекать корень из отр. чисел нельзя d=1.01
```

```
Введите x: 0
Ошибка
```

Пример y =sqrt(1/x)

```
int main()
{   double x, y;
    cout << "Введите x: ";    cin >> x;
      try
      {   if (!cin)  throw 1;
          if (x == 0)  throw 2;
          if (1/x < 0)  throw 3;
       cout << "y =sqrt(1/x) ";
                y=sqrt(1/x);
           cout << "y=" << y << endl;
      }
      catch(int i) {cout << "Поймал исключение №" <<i<< endl;}
return 0;
}
```

```
Введите x: ttt
Поймал исключение № 1
```

```
Введите x: 0
Поймал исключение № 2
```

```
Введите x: -4
Поймал исключение № 3
```

```
Введите x: 25
y =sqrt(1/x) y=0.2
```

Пример с использованием функции

```
int f_del (int n1, int n2){
   if (n2 == 0)
   {  throw "на 0 делить нельзя"; }
   return n1 / n2;
}
```

```
Введите n1 , n2: 5
0
на 0 делить нельзя
```

```
int main()
{   int n1, n2;
      cout << "Введите n1 ,  n2: ";    cin >> n1>>n2;
      try
      {    cout << "n1/n2= "<< f_del(n1,n2) << endl;
      }
      catch(const char *s) {cout << s << endl;}
return 0;
}
```

```
Введите n1 , n2: 8
4
n1/n2= 2
```

Перехват исключений

Когда с помощью **throw** генерируется исключение, функции исполнительной библиотеки C++ выполняют следующие действия:

1) создают копию параметра **throw** в виде статического объекта, который существует до тех пор, пока исключение не будет обработано;

2) в поисках подходящего обработчика раскручивают стек, вызывая деструкторы локальных объектов, выходящих из области действия;

3) передают объект и управление обработчику, имеющему параметр, совместимый по типу с этим объектом.

При раскручивании стека все обработчики на каждом уровне просматриваются последовательно, от внутреннего блока к внешнему, пока не будет найден подходящий обработчик.

При вызове каждой функции в стеке создается область памяти для хранения локальных переменных и адреса возврата в вызывающую функцию. Термин **стек вызовов** обозначает последовательность вызванных, но еще не завершившихся функций. **Раскручиванием стека** называется процесс освобождения памяти из-под локальных переменных и возврата управления вызывающей функции. Когда функция завершается, происходит естественное раскручивание стека. Тот же самый механизм используется и при обработке исключений. Поэтому после того, как исключение было зафиксировано, исполнение не может быть продолжено с точки генерации исключения.

Перехват исключений

Обработчик считается найденным, если тип объекта, указанного после **throw**:

- тот же, что и указанный в параметре **catch** (параметр может быть записан в форме **T, const T, T&** или **const T&,** где **T**— тип исключения);
- является производным от указанного в параметре **catch** (если наследование производилось с ключом доступа **public**);
- является указателем, который может быть преобразован по стандартным правилам преобразования указателей к типу указателя в параметре **catch**.

Обработчики производных классов следует размещать до обработчиков базовых, поскольку в противном случае им никогда не будет передано управление. Обработчик указателя типа **void** автоматически скрывает указатель любого другого типа, поэтому его также следует размещать после обработчиков указателей конкретного типа.

Пример

```
class Hello{//класс, информ. о своем создании и уничтожении
public:  Hello(){cout << "Hello!" << endl;}
         ~Hello(){cout << "Bye!" << endl;}
};
void f1(){
ifstream ifs("\\INVALID\\FILE\\NAME"); // Открываем файл
if (!ifs){  cout << "Генерируем исключение" <<endl;
            throw "Ошибка при открытии файла";}
}
void f2(){ Hello H; f1(); }
int main(){
try{ cout << " Входим в try-блок" << endl;
     f2();  cout << " Выходим из try-блока" << endl;
}
catch(int i){
cout << " Вызван обработчик int. исключение - " << i << endl;
return -1;
}
```

Пример

```
catch(const char * p){
  cout << " Вызван обработчик const char*, исключение "<< p;
  return -1;
  }
catch(...){
   cout << " Вызван обработчик всех исключений ";
   return -1;
   }
return 0; // Все обошлось благополучно
}
```

```
 Входим в try-блок
Hello!
Генерируем исключение
Bye!
 Вызван обработчик const char*, исключение Ошибка при открытии файла
```

Исключения

Обратите внимание, что после порождения исключения был вызван деструктор локального объекта, хотя управление из функции **f1** было передано обработчику, находящемуся в функции **main**. Сообщение «Выходим из **try**-блока» не было выведено. Для работы с файлом в программе использовались потоки (о них позже).

Таким образом, механизм исключений позволяет корректно уничтожать объекты при возникновении ошибочных ситуаций. Поэтому выделение и освобождение ресурсов полезно оформлять в виде классов, конструктор которых выделяет ресурс, а деструктор освобождает. В качестве примера можно привести класс для работы с файлом. Конструктор класса открывает файл, а деструктор — закрывает. В этом случае есть гарантия, что при возникновении ошибки файл будет корректно закрыт, и информация не будет утеряна.

Исключения

Как уже упоминалось, исключение может быть как стандартного, так и определенного пользователем типа. При этом нет необходимости определять этот тип глобально — достаточно, чтобы он был известен в точке порождения исключения и в точке его обработки. Класс для представления исключения можно описать внутри класса, при работе с которым оно может возникать. Конструктор копирования этого класса должен быть объявлен как **public**, поскольку иначе будет не возможно создать копию объекта при генерации исключения (конструктор копирования, создаваемый по умолчанию, имеет спецификатор **public**).

```
class Hello{/*...*/};
int main(){Hello h;
try{ /*...*/    throw h;
  }
catch(Hello){ /*…*/}
}
```

Операторы throw без параметров

Блок **try-catch** может содержать вложенные блоки **try-catch** и если не будет определено соответствующего оператора **catch** на текущем уровне вложения, исключение будет поймано на более высоком уровне. Единственная вещь, о которой нужно помнить, - это то, что операторы, следующие за **throw**, никогда не выполнятся.

```
try{
  throw;
  // ни один оператор, следующий далее (до закрывающей скобки)
  // выполнен не будет
   }
catch(...) {
          cout << "Исключение!" << endl;}
```

Такой метод может применяться в случаях, когда не нужно передавать никаких данных в блок **catch**.

Исключения в конструкторах и деструкторах

Язык C++ не позволяет возвращать значение из конструктора и деструктора. Механизм исключений дает возможность сообщить об ошибке, возникшей в конструкторе или деструкторе объекта. Для иллюстрации создадим класс **Vector**, в котором ограничивается количество запрашиваемой памяти:

```
class Vector{
public:
class Size{}; //  Класс исключения
enum {max = 32000}; // Максимальная длина вектора
Vector(int n) // конструктор
  { if (n<0 || n>max) throw Size();   /*…*/ }
/*…*/};
```

При использовании класса **Vector** можно предусмотреть перехват исключений типа **Size**:

```
try{  Vector *p = new Vector(100);}
catch(Vector::Size){/*Обработка ошибки размера вектора*/}
```

Исключения в конструкторах и деструкторах

В обработчике может использоваться стандартный набор основных способов выдачи сообщений об ошибке и восстановления. Внутри класса, определяющего исключение, может храниться информация об исключении, которая передается обработчику.

Смысл заключается в том, чтобы обеспечить передачу информации об ошибке из точки ее обнаружения в место, где для обработки ошибки имеется достаточно возможностей.

Если в конструкторе объекта генерируется исключение, автоматически вызываются деструкторы для полностью созданных в этом блоке к текущему моменту объектов, а также для полей данных текущего объекта, являющихся объектами, и для его базовых классов. Например, если исключение возникло при создании массива объектов, деструкторы будут вызваны только для успешно созданных элементов. Если объект создается в динамической памяти с помощью операции **new** и в конструкторе возникнет исключение, память из-под объекта корректно освобождается.

Список исключений функции

В заголовке функции можно задать список исключений, которые она может прямо или косвенно порождать. Поскольку заголовок является интерфейсом функции, указание в нем списка исключений дает пользователям функции необходимую информацию для ее использования, а также гарантию, что при возникновении непредвиденного исключения эта ситуация будет обнаружена.

Типы исключений перечисляются в скобках через запятую после ключевого слова **throw**, расположенного за списком параметров функции, например:

void f1() throw (int, const char*){ /* Тело функции */ }

void f2() throw (Oops*){ /* Тело функции */ }

Функция **f1** должна генерировать исключения только типов **int** и **const char***. Функция **f2** должна генерировать только исключения типа указателя на класс **Oops** или производных от него классов. Если ключевое слово **throw** не указано, функция может генерировать любое исключение.

Список исключений функции

Пустой список означает, что функция не должна порождать исключений:

void f() throw (){
// Тело функции, не порождающей исключений
}

Исключения не входят в прототип функции. При переопределении в производном классе виртуальной функции можно задавать список исключений, такой же или более ограниченный, чем в соответствующей функции базового класса.

Указание списка исключений ни к чему не обязывает — функция может прямо или косвенно породить исключение, которое она обещала не использовать.

Список исключений функции

Эта ситуация обнаруживается во время исполнения программы и приводит к вызову стандартной функции **unexpected**, которая по умолчанию просто вызывает функцию **terminate**. С помощью функции **set_unexpected** можно установить собственную функцию, которая будет вызываться вместо **terminate** и определять действие программы при возникновении непредвиден-ной исключительной ситуации.

Функция **terminate** по умолчанию вызывает функцию **abort**, которая завершает выполнение программы. С помощью функции **set_terminate** можно установить собственную функцию, которая будет вызываться вместо **abort** и определять способ завершения программы. Функции **set_unexpected** и **set_terminate** описаны в заголовочном файле <**exception**>.