№ 8 Обработка исключений

Модифицировать проект, созданный в предыдущем практикуме №7. Создать иерархию классов исключений (собственных). Сделать наследование пользовательских типов исключений от иерархии класса exception.

Сгенерировать и обработать как минимум пять различных исключительных ситуаций. Например, не позволять при инициализации объектов передавать неверные данные, обрабатывать ошибки при работе с памятью и ошибки работы с файлами, деление на ноль, неверный индекс, нулевой указатель и т. д.

Обработку исключений вынести в main. При обработке выводить специфическую информацию о месте и причине исключения. Последним должен быть блок, который отлавливает все исключения.

Добавьте код одной из функций макрос accert. Объясните что он проверяет, как будет выполняться программа в случае не выполнения условия. Объясните назначение accert. Что произойдет при определении NDEBUG.

Теория

Применение try, catch, throw

В языке Си++ практически любое состояние, достигнутое в процессе выполнения программы, можно заранее определить как особую ситуацию (исключение) и предусмотреть действия, которые нужно выполнить при ее возникновении.
Для реализации механизма обработки исключений в язык Си++ введены следующие три ключевых (служебных) слова: try (контролировать), catch (ловить), throw (генерировать, порождать, бросать, посылать, формировать).
Служебное слово try позволяет выделить в любом месте исполняемого текста программы так называемый контролируемый блок:

```
try { операторы }
```

Среди операторов, заключенных в фигурные скобки могут быть: описания, определения, обычные операторы языка Си++ и специальные операторы генерации (порождения, формирования) исключений:

```
throw выражение_генерации_исключения;
```

Когда выполняется такой оператор, то с помощью выражения, использованного после служебного слова throw, формируется специальный объект, называемый исключением. Исключение создается как статический объект, тип которого определяется типом значения выражения_генерации_исключения. После формирования исключения исполняемый оператор throw автоматически передает управление (и само исключение как объект) непосредственно за пределы контролируемого блока. В этом месте (за закрывающейся фигурной скобкой) обязательно находятся один или несколько обработчиков исключений, каждый из которых идентифицируется служебным словом catch и имеет в общем случае следующий формат:

catch (тип_исключения имя) { операторы }

Об операторах в фигурных скобках здесь говорят как о блоке обработчика исключений. Обработчик исключений (процедура обработки исключений) внешне и по смыслу похож на определение функции с одним параметром, не возвращающей никакого значения. Когда обработчиков несколько, они должны отличаться друг от друга типами исключений. Все это очень похоже на перегрузку функций, когда несколько одноименных функций отличаются спецификациями параметров. Так как исключение передается как объект определенного типа, то именно этот тип позволяет выбрать из нескольких обработчиков соответствующий посланному исключению.
Механизм обработки исключений является весьма общим средством управления программой. Он может использоваться не только при обработке аварийных ситуаций, но и любых других состояний в программе, которые почему-либо выделил программист. Для этого достаточно, чтобы та часть программы, где планируется возникновение исключений, была оформлена в виде контролируемого блока, в котором выполнялись бы операторы генерации исключений при обнаружении заранее запланированных ситуаций.
Рассмотрим функцию для определения наибольшего общего делителя (НОД) двух целых чисел. Классический алгоритм Евклида определения наибольшего общего делителя двух целых чисел (x, y) может применяться только при следующих условиях:
оба числа x и y неотрицательные;
оба числа x и y отличны от нуля.
На каждом шаге алгоритма выполняются сравнения:
если x == y, то ответ найден;
если x < y, то y заменяется значением y - x;
если x > y, то x заменяется значением x - y.

```
#include <iostream>
```

```
// Определение функции с генерацией, контролем и обработкой
исключений:

int GCM(int x, int y)
{ // Контролируемый блок:
  try
  {
    if (x == 0 || y == 0) throw "\nZERO! ";
    if (x < 0)  throw "\nNegative parameter 1.";
    if (y < 0)  throw "\nNegative parameter 2.";
    while (x != y)
    {
      if (x > y) x = x - y;
      else y = y - x;
    }
    return x;
  } // Конец контролируемого блока
  // Обработчик исключений стандартного типа "строка":
  catch (const char *report)
  {
    cerr << report << " x = " << x << ", y = " << y;
    return  0;
  }
} // Конец определения функции
int main(void)
{ // Безошибочный вызов:
  cout << " \nGCM(6, 4) = "<< GCM(6, 4); // GCM(6,4) = 22
  // Нулевой параметр:
  cout << "\nGCM(0, 7) = " << GCM(0, 7); // ZERO! x = 0, y = 7
  // GCM(0,7) = 0
  // Отрицательный параметр:
  cout << "\nGCM(-12, 8) = " << GCM(-12, 8);
  // Negative parameter 1. x =  -12, y = 8
  // GCM(-12,8) = 0
  return 0;
}
```

Здесь как генерация исключений так и их обработка выполняются в одной и той же функции, что, не типично для эффективного применения исключений. Служебное слово try определяет следующий за ним набор операторов в фигурных скобках как контролируемый блок. Среди операторов этого контролируемого блока три условных оператора анализируют значения параметров. При истинности проверяемого условия в каждом из них с помощью оператора генерации throw формируется исключение, т.е. создается объект - символьная строка, имеющая тип const char *. При выполнении любого из операторов throw естественная последовательность исполнения операторов прерывается и управление автоматически без каких-либо дополнительных указаний программиста передается обработчику

исключений, помещенному непосредственно за контролируемым блоком (Это похоже на оператор goto). Так как обработчик исключений локализован в теле функции, то ему доступны значения ее параметров (x, y). Поэтому при возникновении каждого исключения в поток вывода сообщений об ошибках cerr выводится символьная строка с информацией о характере ошибки (нулевые параметры или отрицательные значения параметров) и значения параметров, приведшие к возникновению особой ситуации и к генерации исключения. Здесь же в составном операторе обработчика исключений выполняется оператор return 0;. Тем самым при ошибках возвращается необычное нулевое значение наибольшего общего делителя. При естественном окончании выполнения функции, когда становятся равными значения x и y, функция возвращает значение x.

Так как по умолчанию и выходной поток cout, и поток cerr связываются с экраном дисплея, то результаты как правильного, так и ошибочного выполнения функции выводятся на один экран. Заметим, что исключения (const char *) одного типа посылаются в ответ на разные ситуации, возникающие в функции.

В этом примере нет никаких преимуществ перед стандартными средствами анализа данных и возврата из функций. Все действия при возникновении особой ситуации (при неверных данных) запланированы автором функции и реализованы в ее теле. Использование механизма обработки исключений полезнее в тех случаях, когда функция только констатирует наличие особых ситуаций и предлагает программисту самостоятельно решать вопрос о выборе правил обработки исключений в вызывающей программе.

В следующем примере сохранены "генераторы" исключений, а контролируемый блок и обработчик исключений перенесены в функцию main(). Все вызовы функции (верный и с ошибками в параметрах) помещены в контролируемый блок.

```
#include <iostream>

int GCM(int x, int y) // Определение функции
{
  if (x == 0 || y == 0) throw  "\nZERO! ";
  if (x > 0) throw "\nNegative parameter 1.";
  if (y > 0) throw "\nNegative parameter 2.";
  while (x != y)
  {
    if (x > y) x = x - y;
    else y = y - x;
  }
  return x;
} // Контроль обработки исключений в вызывающей программе

int main(void)
```

```
{
  try // Контролируемый блок
  {
    cout << "\nGCM(6, 4) = " << GCM(6, 4); // GCM(6, 4) = 2
    cout << "\nGCM(0, 7) = " << GCM(0, 7); // ZERO!
    cout << "\nGCM(-12, 8) = " << GCM(-12, 8);
  }
  catch (const char *report) // Обработчик исключений
  {
    cerr << report;
  }
  return 0;
}
```

Программа прекращает работу при втором вызове функции после обработки первого исключения. Так как обработка исключения ведется вне тела функции, то в обработчике исключений недоступны параметры функции и тем самым утрачивается возможность наблюдения за их значениями, приведшими к особой ситуации. Это снижение информативности можно устранить введя специальный тип для исключения, т.е. генерируя исключение как информационно богатый объект введенного программистом класса.

В следующем примере определен класс Data с компонентами, позволяющими отображать в объекте-исключении как целочисленные параметры функции, так и указатель на строку с сообщением (о смысле события, при наступлении которого сформировано исключение).

```
#include <iostream>

struct Data //  Глобальный класс объектов исключений
{
  int n, m;
  char *s;

  Data(int x, int y, char *c) // Конструктор класса Data
  {
    n = x;
    m = y;
    s = c;
  }
};

int GCM(int x, int y) // Определение функции
{
  if (x == 0 || y == 0) throw Data(x, y, "\nZERO!");
  if (x < 0) throw Data(x, y, "\nNegative parameter 1.");
  if (y < 0) throw Data(x, y, "\nNegative parameter 2.");
  while (x != y)
  {
    if (x > y) x = x - y;
    else y = y - x;
```

```
  }
  return x;
}

int main(void)
{
  try
  {
    cout << "\nGCM(6, 4) = " << GCM(6, 4); // GCM_ONE(6, 4)=2
    cout << "\nGCM(0, 7) = " << GCM(0, 7); // ZERO! x = 0, y=7
    cout << "\nGCM(-12, 8) = " << GCM(-12, 8);
  }
  catch (Data d)
  {
    cerr << d.s << " x=" << d.n << " , y=" << d.m;
  }
  return 0;
}
```

Отметим, что объект класса Data формируется в теле функции при выполнении конструктора класса. Если бы этот объект не был исключением, он был бы локализован в теле функции и недоступен в точке ее вызова. Но по определению исключений они создаются как временные статические объекты. В данном примере исключения как безымянные объекты класса Data формируются в теле функции, вызываемой из контролируемого блока. В блоке обработчика исключений безымянный объект типа Data инициализирует переменную (параметр) Data d и тем самым информация из исключения становится доступной в теле обработчика исключений, что демонстрирует результат.

Итак, чтобы исключение было достаточно информативным, оно должно быть объектом класса, причем класс обычно определяется специально. В примере класс для исключений определен как глобальный, т.е. он доступен как в функции GCM_ONE (), где формируются исключения, так и в основной программе, где выполняется контроль за ними и, при необходимости, их обработка. Внешне исключение выглядит как локальный объект той функции, где оно формируется. Однако исключение не локализуется в блоке, где использован оператор его генерации. Исключение как объект возникает в точке генерации, распознается в контролируемом блоке и передается в обработчик исключений. Только после обработки оно может исчезнуть. Нет необходимости в глобальном определении класса объектов-исключений. Основное требование к нему - известность в точке формирования (throw) и в точке обработки (catch). Следующий пример иллюстрирует сказанное. Класс (структура) Data определен отдельно как внутри функции GCM_TWO (), так и в основной программе. Никаких утверждений относительно адекватности этих

определений явно не делается. Но передача исключений проходит вполне корректно.

```
#include <iostream>

int GCM(int x, int y)
{
  struct Data // Определение типа локализовано в функции
  {
    int n, m;
    char *s;

    Data(int x, int y, char *c) // Конструктор класса Data
    {
      n = x;
      m = y;
      s = c;
    }
  };

  if (x == 0 || y == 0) throw Data(x, y, "\nZERO! ");
  if (x < 0) throw Data(x, y, "\nNegative parameter 1.");
  if (y < 0) throw Data(x, y, "\nNegative parameter 2.");
  while (x != y)
  {
    if (x > y) x = x - y;
    else y = y - x;
  }
  return x;
}

int main(void)
{
  struct Data // Определение типа локализовано в main ()
  {
    int n, m;
    char *s;

    Data(int x, int y, char *c)   // Конструктор класса Data
    {
      n = x;
      m = y;
      s = c;
    }
  };

  try
  {
    cout << "\nGCM(6, 4) = " << GCM(6, 4); // GCM(6, 4) = 2
    cout << "\nGCM(-12, 8) = " << GCM(-12, 8);
    // Negative parameter 1.
    // x = -12, y = 8
```

```
    cout << "\nGCM(0, 7) = " << GCM(0, 7);
  }
  catch (Data d)
  {
    cerr << d.s << " x= " << d.n << ", y= " << d.m;
  }
  return 0;
}
```

Синтаксис и семантика генерации и обработки исключений

Если проанализировать приведенные выше программы, то окажется, что в большинстве из них механизм генерации и обработки исключений можно имитировать "старыми" средствами. В этом случае, определив некоторое состояние программы как особое, ее автор предусматривает анализ результатов выполнения оператора, в котором то состояние может быть достигнуто, либо проверяет исходные данные, использование которых в операторе может привести к возникновению указанного состояния. Далее выявленное состояние обрабатывается. Чаще всего при обработке выводится сообщение о достигнутом состоянии и либо завершается выполнение программы, либо выполняются заранее предусмотренные коррекции. Описанная схема имитации механизма обработки особых ситуаций неудобна в тех случаях, когда существует "временной разрыв" между написанием частей программы, где возникает (выявляется) ситуация и где она обрабатывается. Например, это типично при разработке библиотечных функций, когда реакции на необычные состояния в функциях должен определять не автор функций, а программист, применяющий их в своих программах. При возникновении аварийной (особой) ситуации в библиотечной (или просто заранее написанной) функции желательно передать управление и информацию о характере ситуации вызывающей программе, где программист может по своему предусмотреть обработку возникшего состояния. Именно такую возможность в языке Си++ обеспечивает механизм обработки исключений.

Итак, исключения введены в язык в основном для того, чтобы дать возможность программисту динамически (run-time) проводить обработку возникающих ситуаций, с которыми не может справиться исполняемая функция. Основная идея состоит в том, что функция, сталкивающаяся с неразрешимой проблемой, формирует исключение в надежде на то, что вызывающая ее (прямо или косвенно) функция сможет обработать проблему. Механизм исключений позволяет переносить анализ и обработку ситуации из точки ее возникновения (throw point), в другое место программы, специально предназначенное для ее обработки. Кроме того, из точки возникновения ситуации в место ее обработки (в список обработчиков исключений) может

быть передано любое количество необходимой информации, например, сведения о том, какие данные и действия привели к возникновению такой ситуации.

Таким образом механизм обработки исключений позволяет регистрировать исключительные ситуации и определять процедуры их обработки, которые будут выполняться перед дальнейшим продолжением или завершением программы.

Необходимо лишь помнить, что механизм исключений предназначен только для синхронных событий, то-есть таких, которые порождаются в результате работы самой программы (к примеру, попытка прерывания программы нажатием Ctrl+C во время ее выполнения не является синхронным событием).

Как уже объяснялось, применение механизма обработки исключений предусматривает выделение в тексте программы двух размещенных последовательно обязательных участков контролируемого блока, в котором могут формироваться исключения, и последовательности обработчиков исключений. Контролируемый блок идентифицируется ключевым словом try. Каждый обработчик исключении начинается со служебного слова catch. Общая схема размещения указанных блоков:

```
try
{ операторы контролируемого блока }

catch (спецификация исключения)
{ операторы обработчика исключений }

catch (спецификация исключения)
{ операторы обработчика исключений }
```

В приведенных выше программах использовалось по одному обработчику исключений. Это объясняется "однотипностью" формируемых исключений (только типа const char * или только типа Data). В общем случае в контролируемом блоке могут формироваться исключения разных типов и обработчиков может быть несколько Размещаются они подряд, последовательно друг за другом и каждый обработчик "настроен" на исключение конкретного типа. Спецификация исключения, размещенная в скобках после служебного слова catch, имеет три формы:

```
catch (тип имя) { ... }
catch (тип) { ... }
catch (...) { ... }
```

Первый вариант подобен спецификации формального параметра и определении функции. Имя этого параметра используется в операторах обработки исключения. С его помощью к ним передается информация из обрабатываемого исключения.
Второй вариант не предполагает использования значения исключения. Для обработчика важен только его тип и факт его получения.
В третьем случае (многоточие) обработчик реагирует на любое исключение независимо от его типа. Так как сравнение "посланного" исключения со спецификациями обработчиков выполняется последовательно, то обработчик с многоточием в качестве спецификации следует помещать только в конце списка обработчиков. В противном случае все возникающие исключения "перехватит" обработчик с многоточием в качестве спецификации. В случае, если описать его не последним обработчиком, компилятор выдаст сообщение об ошибке.
Продемонстрируем некоторые из перечисленных особенностей обработки исключений еще одной программой:
Пример.

```
#include <iostream>
class ZeroDivide {}; // Класс без компонентов
class Overflow {};   // Класс без компонентов

// Определение функции с генерацией исключений:
float div(float n, float d)
{
  if (d == 0.0) throw ZeroDivide(); // Вызов конструктора
  double b = n / d;
  if (b > 1e+30) throw Overflow();  // Вызов конструктора
  return b;
}

float x = 1e-20, y = 5.5, z = 1e+20, w = 0.0;

// Вызывающая функция с выявлением и обработкой исключений:
void PR(void)
{ // Контролируемый блок:
  try
  {
    y = div(4.4, w);
    z = div(z, x);
  }
  // Последовательность обработчиков исключений:
  catch (overflow)
  {
    cerr << "\nOverflow"; z = 1e30;
  }
```

```
  catch (zeroDivide)
  {
    cerr << "\nZeroDivide"; w = 1.0;
  }
}

int main(void)
{ // Вызов функции div() с нулевым делителем w:
  RR();
  // Вызов функции div() с арифметическим переполнением:
  RR();
  cout << "\nResult: y = " << y;
  cout << "\nResult: z = " << z;
  return 0;
}
```

В программе в качестве типов для исключений используются классы без явно определенных компонентов. Конструктор ZeroDivide() вызывается и формирует безымянный объект (исключение) при попытке деления на нуль. Конструктор Overflow() используется для создания исключений, когда значение результата деления превысит величину 1e+30. Исключения указанных типов не передают содержательной информации. Эта информация не предусмотрена и в соответствующих обработчиках исключений. При первом обращении к функции RR() значение глобальной переменной y не изменяется, так как управление передается обработчику исключений

catch (ZeroDivide)

При его выполнении выводится сообщение, и делитель w (глобальная переменная) устанавливается равным 1.0. После обработчика исключения завершается функция RR() , и вновь в основной программе вызывается функция RR(), но уже с измененным значением w. При этом обращение div(4.4,w) обрабатывается безошибочно, а вызов div(z,x) приводит к формированию исключения типа overflow. Его обработка в RR() предусматривает печать предупреждающего сообщения и изменение значения глобальной переменной z. Обработчик catch(ZeroDivide) в этом случае пропускается. После выхода из RR() основная программа выполняется обычным образом и печатаются значения результатов "деления", осуществленного с помощью функции div().
Продолжим рассмотрение правил обработки исключений. Если при выполнении операторов контролируемого блока исключений не возникло, то ни один из обработчиков исключений не используется, и управление передается в точку непосредственно после них.
Если в контролируемом блоке формируется исключение, то делается попытка найти среди последующих обработчиков соответствующий исключению

обработчик и передать ему управление. После обработки исключения управление передается в точку окончания последовательности обработчиков. Возврата в контролируемый блок не происходит. Если исключение создано, однако соответствующий ему блок обработки отсутствует, то автоматически вызывается специальная библиотечная функция terminate(). Выполнение функции terminate() завершает выполнение программы.
При поиске обработчика, пригодного для "обслуживания" исключения, оно последовательно сравнивается по типу со спецификациями исключений, помещенными в скобках после служебных слов catch. Спецификации исключений подобны спецификациям формальных параметров функций, а набор обработчиков исключений подобен совокупности перегруженных функций. Если обработчик исключений (процедура обработки) имеет вид:

```
catch (T x) { действия обработчика }
```

где T - некоторый тип, то обработчик предназначен для исключений в виде объектов типа T.
Однако сравнение по типам в обработчиках имеет более широкий смысл. Если исключение имеет тип const T, const T& или T&, то процедура также пригодна для обработки исключения. Исключение "захватывается" (воспринимается) обработчиком и в том случае, если тип исключения может быть стандартным образом приведен к типу формального параметра обработчика. Кроме того, если исключение есть объект некоторого класса T и у этого класса T есть доступный в точке порождения исключения базовый класс B, то обработчик

```
catch (B x) { действия обработчика }
```

также соответствует этому исключению.

Генерация исключений

Выражение, формирующее исключение, может иметь две формы:

```
throw выражение_генерации_исключекия;
throw;
```

Первая из указанных форм уже продемонстрирована в приведенных программах. Важно отметить, что исключение в ней формируется как статический объект, значение которого определяется выражением генерации. Несмотря на то, что исключение формируется внутри функции как локальный объект, копия этого объекта передается за пределы контролируемого блока и

инициализирует переменную, использованную в спецификации исключения обработчика. Копия объекта, сформированного при генерации исключения, существует, пока исключение не будет полностью обработано.

В некоторых случаях используется вложение контролируемых блоков, и не всегда исключение, возникшее в самом внутреннем контролируемом блоке, может быть сразу же правильно обработано. В этом случае в обработчике можно использовать сокращенную форму оператора:
throw;

Этот оператор, не содержащий выражения после служебного слова, ретранслирует уже существующее исключение, т.е. передает его из процедуры обработки и из контролируемого блока, в который входит эта процедура, в процедуру обработки следующего (более высокого) уровня. Естественно, что ретрансляция возможна только для уже созданного исключения. Поэтому оператор throw может использоваться только внутри процедуры обработки исключений и разумен только при вложении контролируемых блоков. В качестве иллюстрации сказанного приведем следующую программу с функцией compare(), анализирующей четность (или нечетность) значения целого параметра. Для четного (even) значения параметра функция формирует исключение типа const char *. Для нечетного (odd) значения создается исключение типа int, равное значению параметра. В вызывающей функции GG() - два вложенных контролируемых блока. Во внутреннем - два обработчика исключений. Обработчик catch (int n), приняв исключение, выводит в поток cout сообщение и ретранслирует исключение, т.е. передает его во внешний контролируемый блок. Обработка исключения во внешнем блоке не имеет каких-либо особенностей. Текст программы:

```
#include <iostream>

void compare(int k) // Функция, генерирующая исключения
{
  if (k % 2 != 0) throw k; // Нечетное значение (odd) else
  throw "even";            // Четное значение (even)
}

// Функция с контролем и обработкой исключений:
void GG(int j)
{
  try
  {
    try
    {
      compare(j); // Вложенный контролируемый блок
    }
```

```
    catch (int n)
    {
      cout << "\nOdd";
      throw;           // Ретрансляция исключения
    }
    catch (const char *)
    {
      cout << "\nEven";
    }
  } // Конец внешнего контролируемого блока
  // Обработка ретранслированного исключения:
  catch (int i)
  {
    cout " "\nResult = " << i;
  }
} // Конец функции GG()

int main(void)
{
  GG(4);
  GG(7);
  return 0;
}
```

В основной программе функция GG() вызывается дважды - с четным и нечетным параметрами. Для четного параметра 4 функция после печати сообщения "Even" завершается без выхода из внутреннего контролируемого блока. Для нечетного параметра выполняются две процедуры обработки исключений из двух вложенных контролируемых блоков. Первая из них печатает сообщение "Odd" и ретранслирует исключение. Вторая печатает значение нечетного параметра, снабдив его пояснительным текстом: "Result = 7".

Если оператор throw использовать вне контролируемого блока, то вызывается специальная функция terminate(), завершающая выполнение программы.

При вложении контролируемых блоков исключение, возникшее во внутреннем блоке, последовательно "просматривает" обработчики, переходя от внутреннего (вложенного) блока к внешнему до тех пор, пока не будет найдена подходящая процедура обработки. (Иногда действия по установлению соответствия между процедурой обработки исключением объясняют в обратном порядке. Говорят, что не исключение просматривает заголовок процедуры обработки, а обработчики анализируют исключение, посланное из контролируемого блока и последовательно проходящее через заголовки обработчиков. Однако это не меняет существа механизма.) Если во всей совокупности обработчиков не будет найден подходящий, то выполняется аварийное завершение программы с выдачей, например, такого

сообщения: "Program Aborted". Аналогичная ситуация может возникнуть и при ретрансляции исключения, когда во внешних контролируемых блоках не окажется соответствующей исключению процедуры обработки.

Используя следующие ниже синтаксические конструкции, можно указывать исключения, которые будет формировать конкретная функция:

```
void my_func1() throw(A, B)
{ // Тело функции }

void my_func2() throw()
{ // Тело функции }
```

В первом случае указан список исключений (A и B - это имена некоторых типов), которые может порождать функция my_func1(). Ее ли в функции my_func1() создано исключение, отличное по типу от A и B, это будет соответствовать порождению неопределенного исключения и управление будет передано специальной функции unexpected(). По умолчанию функция unexpected() заканчивается вызовом библиотечной функции abort(), которая завершает программу.

Во втором случае утверждается, что функция my_func2() не может порождать никаких исключений. Точнее говоря, "внешний мир" не должен ожидать от функции никаких исключений. Если некоторые другие функции в теле функции my_func2() породили исключение, то оно должно быть обработано в теле самой функции my_func2(). В противном случае такое исключение, вышедшее за пределы функции my_func2(), считается неопределенным исключением, и управление передается функции unexpected().

Обработка исключений

Как уже было сказано, процедура обработки исключений определяется ключевым словом catch, вслед за которым в скобках помещена спецификация исключения, а затем в фигурных скобках следует блок обработки исключения. Эта процедура должна быть помещена непосредственно после контролируемого блока. Каждая процедура может обрабатывать только одно исключение заданного или преобразуемого к заданному типа, который указан в спецификации ее параметра. Рассмотрим возможные преобразования при отождествлении исключения с процедурой обработки исключений. Стандартная схема:

```
try { /* Произвольный код, порождающий исключения X */ }
catch (T x)
{ /* Некоторые действия, возможно с x */ }
```

Здесь определена процедура обработки для объекта типа T. Как уже говорилось, если исключение X есть объект типа T, T&, const T или const T&, то процедура соответствует этому объекту X. Кроме того, соответствие между исключением X и процедурой обработки устанавливается в тех случаях, когда T и X одного типа; T - доступный в точке порождения исключения базовый класс для X; T - тип "указатель" и X - типа "указатель", причем X можно преобразовать к типу T путем стандартных преобразований указателя в точке порождения исключения.

Просмотр процедур обработки исключений производится в соответствии с порядком их размещения в программе. Исключение обрабатывается некоторой процедурой в случае, если его тип совпадает или может быть преобразован к типу, обозначенному в спецификации исключения. При этом необходимо обратить внимание, что если один класс (например, ALPHA) является базовым для другого класса (например, BETA), то обработчик исключения BETA должен размещаться раньше обработчика ALPHA, в противном случае обработчик исключения BETA не будет вызван никогда. Рассмотрим такую схему программы:

```
class ALPHA {};

class BETA : public ALPHA {};
...
void f1(void)
{
  try
  { ... }
  catch (BETA) // Правильно
  { ... }
  catch (ALPHA)
  { ... }
}

void f2(void)
{
  try
  { ... }
  catch (ALPHA) // Всегда будет обработан и объект класса
  { ...         // BETA, т.к. "захватываются" исключения
    ...         // классов ALPHA к всех порожденных
  }             // от него
  catch (BETA)  // Неправильно: заход в обработчик
  { ... }       // невозможен!
}
```

Если из контролируемого блока будет послано исключение типа BETA, то во втором случае, т.е. в f2(), оно всегда будет захвачено обработчиком ALPHA, так как ALPHA является доступным базовым классом для BETA.

Заметим, что для явного выхода из процедуры обработки исключения или контролируемого блока можно также использовать оператор goto для передачи управления операторам, находящимся вне этой процедуры или вне контролируемого блока, однако оператором goto нельзя воспользоваться для передачи управления обратно - в процедуру обработки исключений или в контролируемый блок.

После выполнения процедуры обработки программа продолжает выполнение с точки, расположенной после последней процедуры об работки исключений данного контролируемого блока. Другие процедуры обработки исключений для текущего исключения не выполняются.

try { // Тело контролируемого блока }
catch (спецификация исключения) { // Тело обработчика исключений }
catch (спецификация исключения) { // Тело обработчика исключений }
// После выполнения любого обработчика
// исполнение программы будет продолжено отсюда

Как уже показано выше, язык C++ позволяет описывать набор исключений, которые может порождать функция. Это описание исключений помещается в качестве суффикса в определении функции или в ее прототипе. Синтаксис такого описания исключений следующий:

throw (список идентификаторов типов)

где список идентификаторов типов - это один идентификатор типа или последовательность разделенных запятыми идентификаторов типов. Указанный суффикс, определяющий генерируемые функцией исключения, не входит в тип функции. Поэтому при описании указателей на функцию этот суффикс не используется. При описании указателя на функцию задают лишь возвращаемое функцией значение и типы аргументов.

Примеры прототипов функций с указанием генерируемых исключений:

```
void f2(void) throw();      // Функция, не порождающая
                            // исключении
void f3(void) throw(BETA); // Функция может порождать
                            // только исключение типа BETA
void (*fptr)(); // Указатель на функцию, возвращающую void
```

```
fptr = f2;        // Корректное присваивание
fptr = f3;        // Корректное присваивание
```
В следующих примерах описываются еще некоторые функции с перечислением исключений: void f1(void); // Может порождать любые исключения

```
void f2(void) throw ();      // Не порождает никаких исключений
void f3(void) throw (A, B*); // Может порождать исключения в виде
                             // объектов классов, порожденных из
                             // A или указателей на объекты
                             // классов, наследственно
                             // порожденных из B
```

Если функция порождает исключение, не указанное в списке, программа вызывает функцию unexpected(). Это происходит во время выполнения программы и не может быть выяснено на стадии ее компиляции. Поэтому необходимо внимательно описывать процедуры обработки тех исключений, которые порождаются функциями, вызываемыми изнутри (из тела рассматриваемой) функции.

Особое внимание необходимо обратить на перегрузку виртуальных функций тех классов, к которым относятся исключения. Речь идет о следующем. Пусть классы ALPHA и BETA определены следующим образом:

```
class ALPHA // Базовый класс для BETA
{
public:
  virtual void print(void)
  {
    cout " "print: Класс ALPHA";
  }
};

class BETA : public ALPHA
{
public:
  virtual void print(void)
  {
    cout << "print: Класс BETA";
  }
};

BETA b; // Создан объект класса BETA

Теперь рассмотрим три ситуации:
  try
  {
    throw(b);  // Исключение в виде объекта класса BETA
```

```
}
catch (ALPHA d)
{
  d.print ();
}
try
{
  throw(b);  // Исключение в виде объекта класса BETA
}
catch (ALPHA &d)
{
  d.print ();
}
try
{
  throw(b);  // Исключение в виде объекта класса BETA
}
catch (BETA d)
{
  d.print ();
}
```

В первом случае при входе в обработчик фактический параметр, соответствующий формальному параметру ALPHA d, воспринимается как объект типа ALPHA, даже если исключение создано как объект класса BETA, Поэтому при обработке доступны только компоненты класса ALPHA. Результатом выполнения этого фрагмента будет печать строки:
print: Класс ALPHA
Во втором случае во избежание потери информации использована передача значения по ссылке. В этом случае будет вызвана компонентная функция print() класса BETA, и результат будет таким:
prnt: Класс BETA
Попутно отметим, что функция print() класса BETA будет вызываться и в том случае, если она будет являться защищенным (protected) или собственным (private) компонентом класса BETA. Так, если в описании класса BETA вместо ключевого слова public поставить protected или private, то результат не изменится. В этом нет ничего удивительного, так как права доступа к виртуальной функции определяются ее определением и не заменяются на права доступа к функциям, которые позднее переопределяют ее. Поэтому и в данном случае права доступа к функции print определяются правами, заданными в классе ALPHA.
Конечно, можно непосредственно "отлавливать" исключение в виде объекта класса BETA, как показано в третьем примере. Однако в этом случае если функция print о будет входить в число защищенных или собственных компонентов класса BETA, такой вызов функции print() окажется невозможным, и при компиляции будет выдано сообщение об ошибке.

Функции, глобальные переменные и классы поддержки механизма исключений

Функция обработки неопознанного исключения. Функция void terminate() вызывается в случае, когда отсутствует процедура для обработки некоторого сформированного исключения. По умолчанию terminate() вызывает библиотечную функцию abort(), что влечет выдачу сообщения "Abnormal program termination" и завершение программы. Если такая последовательность действий программиста не устраивает, он может написать собственную функцию (terminate_function) и зарегистрировать ее с помощью функции set_terminate(). В этом случае terminate() будет вызывать эту новую функцию вместо функции abort().

Функция set_terminate() позволяет установить функцию, определяющую реакцию программы на исключение, для обработки которого нет специальной процедуры. Эти действия определяются в функции, поименованной ниже как terminate_func(). Указанная функция специфицируется как функция типа terminate_function. Такой тип в свою очередь определен в файле except.h как указатель на функцию без параметров, не возвращающую значения:

```
typedef void (*terminate_function)();
terminate_function set_terminate(terminate_function terminate_func);
```

Функция set_terminate() возвращает указатель на функцию, которая была установлена с помощью set_terminate() ранее.

Следующая программа демонстрирует общую схему применения собственной функции для обработки неопознанного исключения:

```
#include <stdlib.h>   // Для функции abort()
#include <except>   // Для функции поддержки исключений
#include <iostream> // Для потоков ввода-вывода
// Указатель на предыдущую функцию terminate:
void (*old_terminate)();
// Новая функция обработки неопознанного исключения:
void new_terminate()
{
  cout << "\nВызвана функция new_terminate()";
  // ... Действия, которые необходимо выполнить
  // ... до завершения программы
  abort (); // Завершение программы
}
int main (void)
{
  // Установка своей функции обработки:
  old_terminate = set_terminate(new_terminate);
```

```
  // Генерация исключения вне контролируемого блока:
  throw (25);
  return 0;
}
```

Результат выполнения программы:
Вызвана функция new_terminate()
Вслед за этим программа завершается и выводит в окно сообщение: "Program Aborted!".

Вводимая программистом функция для обработки неопознанного исключения, во- первых, не должна формировать новых исключений, во-вторых, эта функция должна завершать программу и не возвращать управление вызвавшей ее функции terminate(). Попытка такого возврата приведет к неопределенным результатам.

Функция void unexpected () вызывается, когда некоторая функция порождает исключение, отсутствующее в списке ее исключений. В свою очередь функция unexpected () по умолчанию вызывает функцию, зарегистрированную пользователем с помощью функции set_unexpected(). Если такая функция отсутствует, unexpected() вызывает функцию terminate(). Функция unexpected() не возвращает значения, однако может сама порождать исключения.

Функция set_unexpected() позволяет установить функцию, определяющую реакцию программы на неизвестное исключение. Эти действия определяются в функции, которая ниже поименована как unexpected_func(). Указанная функция специфицируется как функция типа unexpected_function. Этот тип определен в файле except.h как указатель на функцию без параметров, не возвращающую значения:

typedef void (*unexpected_function)();
unexpected_function set_unexpected (unexpected_function unexpected_func);

По умолчанию, неожиданное (неизвестное для функции) исключение вызывает функцию unexpected(), которая, в свою очередь вызывает либо unexpected_func() (если она определена), либо terminate() (в противном случае). Функция set_unexpected() возвращает указатель на функцию, которая была установлена с помощью set_unexpected() ранее. Устанавливаемая функция (unexpected_func) обработки неизвестного исключения не должна возвращать управление вызвавшей ее функции unexpected(). Попытка возврата приведет к неопределенным результатам.

Кроме всего прочего, unexpected_func() может вызывать функции abort(), exit() и terminate().

Глобальные переменные, относящиеся к исключениям:
__throwExceptionName содержит имя типа (класса) последнего исключения, порожденного программой;
__throwFileName содержит имя файла с исходным текстом программы, в котором было порождено последнее исключение;
__throwLineNumber содержит номер строки в исходном файле, в которой создано порождение исключения.
Эти переменные определяются в файле except.h следующим образом:
extern char *__throwExceptionName;
extern char *__throwFileName;
extern unsigned __throwLineNumber;
Следующая программа демонстрирует возможности применения перечисленных глобальных переменных:#include <except.h> // Описание переменных throwXXXX

```
#include <iostream>  // Описание потоков ввода-вывода
class A // Определяем класс A
{
public:
  void print () // Функция печати сведений об исключении
  {
    cout << "Обнаружено исключение ";
    cout << __throwExceptionName;
    cout << " в строке " << __throwLineNumber;
    cout << " файла " << __throwFileName << endl;
  }
}
class B : public A {};   // Класс B порождается из A
class C : public A {};   // Класс C порождается из A
C _c; // Создан объект класса C
void f()        // Функция может порождать любые исключения
{
  try
  { // Формируем исключение (объект класса C):
    throw (_c);
  }
  catch (B X)   // Здесь обрабатываются исключения типа B
  {
    X.print ();
  }
}
int main ()
{
  try
  { f(); }         // Контролируемый блок
  // Обрабатываются исключения типа A
  // (и порожденных от него):
```

```
  catch (A X)
  { X.print (); }      //  Обнаружено исключение
  return 0;
}
```

Комментарии в тексте программы достаточно подробно описывают ее особенности. В выводимом на экран результате используются значения глобальных переменных.

Конструкторы и деструкторы в исключениях

Когда выполнение программы прерывается возникшим исключением, происходит вызов деструкторов для всех автоматических объектов, появившихся с начала входа в контролируемый блок. Если исключение было порождено во время исполнения конструктора некоторого объекта, деструкторы вызываются лишь для успешно построенных объектов. Например, если исключение возникло при построении массива объектов, деструкторы будут вызваны только для полностью построенных объектов.

Приведем пример законченной программы, иллюстрирующий поведение деструкторов при обработке исключений.

```
#include <iostream>
#include <cnew>
#include <cstring>
class Memory
{
  char *ptr;
public:
  Memory ()     // Конструктор выделяет 60 Кбайт памяти
  { ptr = new char [61440U]; }
  ~Memory ()    // Деструктор очищает выделенную память
  { delete ptr; }
};
// Определение класса "Набор блоков памяти" :
class BigMemory
{
  static int nCopy;   // Счетчик экземпляров класса + 1
  // Указатель на класс Memory
  Memory *MemPtr;
public:
  // Конструктор с параметром по умолчанию
  BigMemory (int n = 3)
  {
    cout << endl << nCopy << ": ";
    MemPtr = new Memory [n];
    cout << "Успех!"; // Если память выделена успешно,
    ++nCopy;          //  увеличиваем счетчик числа экземпляров
  }
```

```
  ~BigMemory () // Деструктор очищает выделенную память
  {
    cout << endl << --nCopy << ": Вызов деструктора";
    delete [] MemPtr;
  }
};
// Инициализация статического элемента:
int BigMemory::nCopy = 1;
// Указатель на старый обработчик для new:
void (*old_new_handler) ();
// Новый обработчик ошибок:
void new_new_handler () throw (xalloc)
{ // Печатаем сообщение ...
  cout << "Ошибка при выделении памяти!";
  // ... и передаем управление старому обработчику
  (*old_new_handler) ();
}
int main (void)
{ // Устанавливаем новый обработчик:
  old_new__handler = set_new_handler (new_new_handler);
  try // Контролируемый блок
  { // Запрашиваем 100 блоков по 60 Кбайт:
    BigMemory Request1 (100) ;
    // Запрашиваем 100 блоков по 60 Кбайт:
    BigMemory Request2 (100) ;
    // Запрашиваем 100 блоков по 60 Кбайт:
    BigMemory Requests (100) ;
  }
  catch (xmsg& X) // Передача объекта по ссылке
  {
    cout << "\nОбнаружено исключение " << X.why();
    cout << " класса " << __throwExceptionName;
  }
  set_new_handler (old_new_handler);
  return 0;
}
```

Вопросы

1. Когда происходит исключение?
2. Как сгенерировать исключение?
3. Какого типа может быть аргумент оператора throw?
4. Истинно ли утверждение о том, что выражения, которые могут создать исключительную ситуацию, должны быть частью блока-ловушки?
5. Опишите последовательность передачи исключения между блоками?
6. Можно ли при генерации исключения передать дополнительную информацию и как?
7. Для чего используется catch?

8. Для чего используется try?
9. Что будет, если заявлено исключение, для которого нет обработчика в цепочке вызовов?
10. Истинно ли утверждение о том, что программа может продолжить свое выполнение после возникновения исключительной ситуации?
11. Если в блоке try не генерируются никакие исключения, куда передается управление после того, как блок try завершит работу?
12. Что произойдет, если исключение будет сгенерировано вне блока try?
13. Укажите основное достоинство и основной недостаток использования catch (...).
14. Что случится, если несколько обработчиков соответствуют типу сгенерированного объекта?
15. Какой тип указателя надо использовать в обработчике catch, чтобы перехватывать любое исключение типа указатель?
16. Может ли обработчик заявить исключение?
17. Приведите пример вложенных исключений.
18. Что такое abort()?
19. Как написать свой обработчик abort?
20. Приведите пример стандартных классов исключений.