Перечисления в С++
Исключения

Никитин Михаил Евгеньевич, 2015

Перечисления в C++

- *перечисления* - это тип который может содержать значения указанные программистом.
- Целочисленные именованные константы могут быть определены как члены перечисления.

```
enum { RED, GREEN, BLUE };
```

- определяет три целочисленные константы и присваивает им значения. По умолчанию, значения присваиваются по порядку начиная с нуля, т.е. RED == 0, GREEN == 1 и BLUE == 2.

Перечисления в C++

- Перечисление также может быть именованным:

```
enum color { RED, GREEN, BLUE };
```

- Например RED имеет тип color. Объявление типа переменной как color, вместо обычного unsigned, может подсказать и программисту и компилятору о том как эта переменная должна быть использована.

Перечисления в C++


```
void f(color c)
{
    switch (c) {
    case RED:
    // do something
    break;
    case BLUE:
    // do something
    break;
    }
}
```

Перечисления в C++


```
void f(color c)
{
    switch (c) {
    case RED:
    // do something
    break;
    case BLUE:
    // do something
    break;
    }
}
```

Перечисления в C++


```
enum color { RED=10, GREEN=20, BLUE =30};
```

```
enum Token_value {
    NAME, NUMBER, END,
    PLUS = '+', MINUS = '-', MUL = '*', DIV = '/',
    PRINT = ';', ASSIGN = '=', LP = '(', RP = 'Y'
};
```

Обработка ошибок

- При работе программ возникаю ошибки времени выполнения (runtime error), когда дальнейшее нормальное выполнение приложения становится невозможным.
- Причиной ошибок времени выполнения могут быть как ошибки в программе, так и неправильные действия пользователя, неверные данные и т.д.

Обработка ошибок

- Автор библиотеки может обнаружить ошибки времени выполнения, но, в общем случае, не имеет ни малейшего представления, что с ними делать.
- Пользователь библиотеки может знать, как бороться с такими ошибками, но не может их обнаружить — в противном случае, они бы обрабатывались в коде пользователя, и их обнаружение не было бы возложено на библиотеку.

Обработка ошибок

- Обработка ошибок должна быть разделена на две части:
  1. генерация информации о возникновении ошибочной ситуации, которая не может быть разрешена локально;
  2. обработка ошибок, обнаруженных в других местах.

Обработка ошибок

- При обнаружении проблемы, которая не может быть решена локально, функция может:
  1. прекратить выполнение;
  2. возвратить значение, означающее «ошибка»;
  3. возвратить допустимое значение и оставить программу в ненормальном состоянии;
  4. вызвать функцию, предназначенную для обработки «ошибочных» ситуаций.

Обработка ошибок

- Вариант 1 — «прекратить выполнение» — это то, что происходит по умолчанию, когда не перехватывается исключение.
- Вариант 2 — «возвратить значение, сигнализирующее об ошибке» — не всегда выполним, потому что часто нет приемлемого соответствующего значения.

Обработка ошибок

- Вариант 3 — «возвратить допустимое значение и оставить программу в ненормальном состоянии» — имеет тот недостаток, что вызывающая функция может не заметить, что программа находится в ненормальном состоянии.
- Обработка исключений не предназначена для решения проблем, для которых подходит вариант 4 «вызвать функцию обработки ошибок».

Обработка ошибок


```
namespace Error {
    int no_of_errors;
    double error(const char* s)
    {
        std::cerr << "ошибка " << s << "
std::endl;
        no_of_errors++;
        return 1;
    }
}
```

Обработка ошибок

- Для помощи в решении проблем возникновения и обработки ошибок введено понятие ***исключения.***
- Фундаментальная идея состоит в том, что функция, обнаружившая проблему, но не знающая как ее решить, ***генерирует*** (throw) исключение в надежде, что вызвавшая ее (непосредственно или косвенно) функция сможет решить возникшую проблему.
- Функция, которая хочет решать проблемы данного типа, может указать, что она ***перехватывает*** (catch) такие исключения.

Исключения

- Механизм обработки исключений предоставляет
  - альтернативу традиционным методам в тех случаях, когда они не достаточны, не элегантны и подвержены ошибкам.
  - способ явного отделения кода обработки ошибок от «обычного» кода.
  - более регулярный способ обработки ошибок, упрощая в результате взаимодействие между отдельно написанными фрагментами кода.

Исключения. Блок try-catch

- Простейший формат защищенного блока имеет вид:

Полный формат защищенного блока имеет вид:
try {операторы защищенного блока}
catch-блоки

Catch-блок имеет один из следующих форматов:
catch (тип) {обработчик ошибочной ситуации}
catch (тип идентификатор) {обработчик ошибочной ситуации}
catch (...) {обработчик ошибочной ситуации}

Исключения. Блок try-catch

- Простейший формат защищенного блока :

```
try
    {операторы защищенного блока}
catch(...)
    {обработчик ошибочной ситуации}
```

```
int main()
{
   int x = 0;
   try {
    std::cout << 2 / x; //Здесь произойдет генерация исключения
                // Последующие операторы выполняться не будут
   }
   catch (...) {
    std::cout << "Division by zero" << std::endl;
   }
}
```

- **Project Properties -> C/C++ -> Code Generation -> Modify the Enable C++ Exceptions to "Yes With SEH Exceptions"**

throw

- throw (бросить) - ключевое слово, "создающее" ("возбуждающее") исключение.

```
struct Range_error{
 int i;
 Range_error(int _i) {i = _i;}
};
char to_char(int i)
{
 if(i< 0 ||255 < i)
 throw Range_error(i);
 return i;
}
```

Перехватывать (catch)


```
 struct Range_error{
    int i;
    Range_error(int _i) {i = _i;}
   };
  char to_char(int i){
    if(i< 0 ||255 < i)
    throw Range_error(i);
    return i;
   }
void g(int i){
    try {
       char c = to_char(i);
    }
    catch (Range_error) {
       cerr << "проблема" << endl;
    }
   }
```

Перехватывать (catch)


```
struct Range_error{
    int i;
    Range_error(int _i) {i = _i;}
  };
char to_char(int i){
    if(i< 0 ||255 < i)
    throw Range_error(i);
    return i;
  }
 void h(int i){
    try {
       char c = to_char(i);
    }
    catch (Range_error x) {
      cerr << "проблема, to char (" << x.i << endl;
    }
  }
```

Исключения

- Если код в ***try-блоке,*** или код, вызываемый из него, генерирует исключение, будут проверяться обработчики этого блока ***try.***
- Если сгенерированное исключение имеет тип, указанный в одном из обработчиков, будет выполнен этот обработчик.
- Если исключение сгенерированно и ни один из ***try-блоков*** не перехватил его, выполнение программы прекращается

Выбор исключений

- Как правило, в каждой программе существует несколько возможных типов ошибок на этапе выполнения.
- Такие ошибки можно распределить между исключениями с различными именами.
- Это сводит к минимуму путаницу, связанную с их назначением.
- Никогда не пользуюсь встроенными типами, такими как ***int***, для описания исключения.
- В большой программе нет эффективного способа разделения исключений, обрабатывающих ***int***.

Выбор исключений

- Наш калькулятор должен обрабатывать две ошибки времени выполнения:
  - синтаксические ошибки;
  - попытку деления на ноль.

```
struct Zero_divide {};

struct Syntax_error {
    const char* p;
    Syntax_error(const char* q) { p = q; }
};
```

Выбор исключений


```
try {
  //...
  expr(false);
  // мы попадем сюда в том и только в том случае,
  // если expr() не возбудит исключения
  // ...
}
catch (Syntax error) {
  // обработка синтаксической ошибки
}
catch (Zero divide) {
  // обработка деления на ноль
}
// мы попадем сюда, если expr не сгенерировал исключения, либо если были
// сгенерированы исключения Syntax_error или Zero_divide
// (и их обработчики не сгенерировали исключения
// или некоторым другим способом не изменили потока управления).
```

Выбор исключений

- С точки зрения языка считается, что исключение обработано сразу после входа в его обработчик.
- Это сделано для того, чтобы любые исключения, сгенерированные во время выполнения обработчика, обрабатывались функцией с ***try-блоком,*** вызвавшим исключение.

Выбор исключений


```
class Input overflow {/*... */ },

void f()
{
   try {
   //...
   }
   catch (Input_overflow) {
   //...
      throw Input overflow();
   }
}
```

Производные классы

- Пусть в базе данных ВУЗа должна храниться информация о всех студентах и преподавателях.
- Как реализовать данную программу используя классы?

Производные классы (наследование)

- Понятие производного класса и связанные с ним механизмы языка предназначены для выражения иерархических отношений, то есть для отражения общности классов.
- Например, концепции студента и преподавателя связаны — они являются людьми (человек); то есть концепция человек является общей для них.
- Поэтому мы должны явно определить, что классы ***student*** и ***teacher*** имеют общий класс ***human***.
- Наследование позволяет избежать дублирования лишнего кода при написании классов.

Базовый класс


```
#include <string>
#include <iostream>

using namespace std;

class human {
   public:
     // Конструктор класса human
     human(string last_name, string name, string second_name);
     // Получение ФИО человека
     string human::get_full_name();
   private:
     string name; // имя
     string last_name; // фамилия
     string second_name; // отчество
};
```

Базовый класс


```
human::human(string last_name, string name, string second_name)
{
    this->last_name = last_name;
    this->name = name;
    this->second_name = second_name;
}
string human::get_full_name()
{
    string full_name;
    full_name = this->last_name + ' '  + this->name + ' ' + this->second_name;
    return full_name;
}

int main()
{
   human test("12", "13", "14");
   cout << test.get_full_name();
}
```

Наследование от базового класса

- Теперь создайте новый класс student, который будет наследником класса human.

```
class student : public human {
   public:
     // Конструктор класса Student
          student::student(string last_name, string name, string
second_name, vector<int> scores);
     // Получение среднего балла студента
     float student::get_average_score();
   private:
     // Оценки студента
     vector<int> scores;
};
```

Наследование от базового класса


```
student::student(string last_name,string name,string second_name,vector<int> scores):
    human(last_name,name,second_name)
    {
     this->scores = scores;
    }
float student::get_average_score(){
     // Общее количество оценок
     unsigned int count_scores = this->scores.size();
     // Сумма всех оценок студента
     unsigned int sum_scores = 0;
     // Средний балл
     float average_score;

     for (unsigned int i = 0; i < count_scores; ++i) {
         sum_scores += this->scores[i];
     }

     average_score = (float)sum_scores / (float)count_scores;
     return average_score;
}
```