13. Обработка на изключения

План:
Обработка на грешки
Изхвърляне на изключения
Прихващане на изключения
Развиване на стека
Спецификации на изключенията

Обработка на грешки

До сега за обработка на грешки - Макрос assert.
Как да се обработват (логически) грешки, възникващи във функция или клас?
Пример:

```
cout << future_value(1000, -100, -1);
```

** Не се проверяват условия за грешки и се изпълнява програмата.

```
double future_value(double initial_balance, double p, int n)
{  return initial_balance * pow(1 + p / 100, n);
}
```

** Проверяват се условията за грешка, не се прави нищо в случай на грешка (връща се фалшива стойност).

```
double future_value(double initial_balance, double p, int n)
{  if (p < 0 || n < 0) return 0;
   return initial_balance * pow(1 + p / 100, n);
}
```

** Проверяват се условията за грешка, съобщава се за грешката и се спира изпълнението на програмата.

```
double future_value(double initial_balance, double p, int n)
{  assert(p >= 0 && n >= 0);
   return initial_balance * pow(1 + p / 100, n);
}
```

Възникващи проблеми:
** Ако не се прави проверка за грешка, програмата е ненадеждна.
** Ако се върне фалшива стойност, програмата продължава да използва стойността - никой не знае каква е грешката.
** Много реални програми не могат да бъдат спирани като тези, които управляват ракета, самолет или медицинско устройство.

Изхвърляне на изключения

C++ има механизъм за уведомяване и обработка на изключения.
Когато дадена функция открие грешка, тя може да сигнализира за състояние (изхвърли изключение) към някоя друга част на програмата, чиято работа е да обработва грешки.

```
double future_value(double initial_balance, double p, int n)
{  if (p < 0 || n < 0)
   {  logic_error description("illegal future_value parameter");
      throw description;
   }
   return initial_balance * pow(1 + p / 100, n);
}
```

Може да не се задава име на обекта за обработка на изключения:

```
double future_value(double initial_balance, double p, int n)
{ if (p < 0 || n < 0)
    throw logic_error("illegal future_value parameter");
 return initial_balance * pow(1 + p / 100, n);
}
```

`logic_error` е стандартен клас за изключения, който е дефиниран в заглавния файл `<stdexcept>`.
Ключовата дума `throw` показва, че изпълнението на функциятя спира незабавно, но управлението не се връща на извикващата функция, а се търси частта от програмата за обработка на изключения.

Прихващане на изключения

```
try
{
   // code
}
catch (logic_error& e)
{
   // handler
}
```

Ако някоя от функциите в блока `try` изхвърли изключение `logic_error`, или вика друга функция, която изхвърля такова изключение, блокът на `catch(logic_error)` се изпълнява веднага.

```
while (more)
{  try
```

```
   {
      // code
   }
   catch (logic_error& e)
   {  cout << "A logic error has occurred "
           << e.what() << "\n";
      ........
   }
}
```

Член-функцията `what` от класа **`logic_error`** връща низа, който е зададен като параметър при конструиране на обекта в оператора `throw`.

```
try
{
 statements
}
catch (type_name1 variable_name1)
{
 statements
}
catch (type_name2 variable_name2)
{
 statements
}
...
catch (type_namen variable_namen)
{
 statements
}
```

Пример:

```
try
{
 double fvalue = future_value(init, interest, years);
 cout << "The future value is " << fvalue << endl;
}
catch(logic_error& e)
{
 cout << "Processing error " << e.what() << "\n";
}
```

```
//exception1.cpp
#include <iostream>
#include <stdexcept>
#include <cmath>
using namespace std;

double future_value(double initial_balance, double p, int n)
{  if (p < 0 || n < 0)
   {  logic_error description("illegal future_value parameter");
      throw description;
   }
   return initial_balance * pow(1 + p / 100, n);
}

int main()
{  bool more = true;
   while (more)
   {  double init, interest;
      int years;
      cout << "Enter initial value, interest and years: ";
      cin >> init >> interest >> years;

      try
      {  double fvalue = future_value(init, interest, years);
         cout << "The future value is " << fvalue << endl;
      }
      catch (logic_error& e)
      {  cout << "A logic error has occurred "
              << e.what() << "\n";
      }
      cout << "Retry? (y/n)";
      char input;
      cin >> input;
      if (input == 'n') more = false;
   }
return 0;
}
```

Добре е всяко изключение да се дефинира като отделен клас.
Пример:

```
class FutureValueError : public logic_error {
public:
   FutureValueError(const char reason[]);
};

FutureValueError::FutureValueError(const char reason[])
   : logic_error(reason){}
```

и функцията `future_value` може да изхвърли `FutureValueError` обект:

```
if (p < 0 || n < 0)
   throw FutureValueError("illegal parameter");
```

Тъй като `FutureValueError` е **`logic_error`**, то това изключение може да се прихване от **`catch(logic_error& e)`**.
Разбира се, може да построим и специално за него **`catch`** блок:

```
try
{
   // code
}
catch (FutureValueError& e)
{
   // handler1
}
catch (logic_error& e)
{
   // handler2
}
```

В този случай ще се изпълни само първия **`catch`** блок.

```
// exception2.cpp
#include <iostream>
#include <stdexcept>
#include <cmath>
using namespace std;

class FutureValueError : public logic_error {
```

```
public:
   FutureValueError(const char reason[]);
};

FutureValueError::FutureValueError(const char reason[])
   : logic_error(reason){}

double future_value(double initial_balance, double p, int n)
{  if (p < 0 || n < 0)
     throw FutureValueError("illegal future_value parameter");
   return initial_balance * pow(1 + p / 100, n);
}

void read(double& init, double& interest, int& years)
{  cout << "Enter initial value, interest and years: ";
   cin >> init >> interest >> years;
   if (years > 100) throw logic_error("too many years!");
   cout << init << " " << interest << " " << years << endl;
}

int main()
{  bool more = true;
   while (more)
    {  double init, interest;
       int years;
       try
       {  read(init, interest, years);
          double fvalue = future_value(init, interest, years);
          cout << "The future value is " << fvalue << endl;
       }
       /**/
       catch (FutureValueError& e)
       {  cout << "A FutureValueError has occurred: "
               << e.what() << "\n";
       }
       /**/
       catch (logic_error& e)
       {  cout << "A logic error has occurred: "
               << e.what() << "\n";
       }
```

```
        cout << "Retry? (y/n)";
        char input;
        cin >> input;
        if (input == 'n') more = false;
    }
    return 0;
}
```

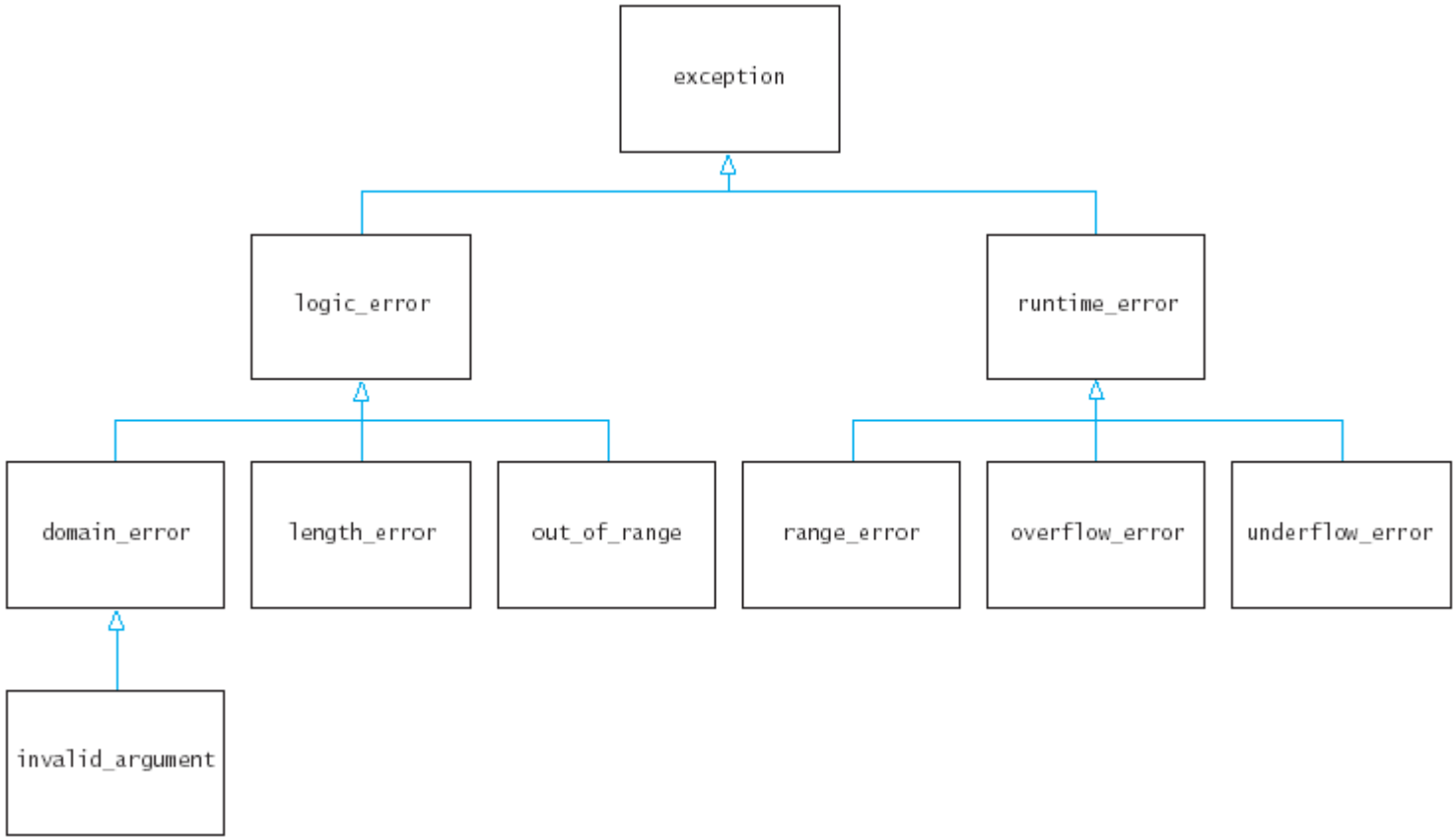
Развиване на стека

Често механизмът за обработка на изключения се прилага при вход на данни.
Пример (виж **product2.cpp**).

```
bool Product::read(fstream& fs)
{  getline(fs, name);
   if (name == "") return false; // end of file
   fs >> price >> score;
   if (fs.fail())
      throw runtime_error("Error while reading product");
   string remainder;
   getline(fs, remainder);
   return true;
}
```

Когато възникне логическа грешка, няма смисъл да се повтори същата операция, но при `runtime_error` има шанс да се повтори операцията и грешката да се избегне.
Има съществена разлика при обработката на край на файла и грешка при входа.

Йерархията на стандартните класове за обработка на изключения е следната:



Нека имаме следната функция (виж bestval2.cpp):

```
void process_products(fstream& fs)
{  vector<Product> products;
   bool more = true;
   while (more)
   {  Product p;
      if (p.read(fs)) products.push_back(p);
      else more = false;
   }
   // do something with products
}
```

Когато функцията `read` изхвърли изключение, функцията `process_products` прекъсва и механизмът за обработка на изключения търси подходящ **`catch`** блок.

При прекъсването се изпълняват всички деструктори на локалните обекти.
В случая обектът `products` (от класа vector) ще бъде унищожен.
Когато имаме указатели, трябва да се погрижим за унищожаването на обектите в динамичната памет.
Пример:

```
Product* p = new Product();
if (p->read())
{
   ...
}
delete p; // never executes if read throws an exception
```

Този код генерира "memory leak", ако функцията `read` изхвърли изключение (виж product1.cpp).
В такъв случай може да се използва следната конструкция:

```
Product* p = NULL;
try
{  p = new Product();
   if (p->read())
   {
      ...
   }
   delete p;
}
catch(...)
{  delete p;
   throw;
}
```

Блокът **`catch(...)`** обработва всяко изключение, а операторът **`throw`** без обект изпраща изключението да се обработва от друг **`catch`** блок.

```
// product2.cpp
#include <iostream>
#include <fstream>
#include <stdexcept>
#include <vector>
using namespace std;

class Product {
   public:
       Product();
```

```
        bool read(istream& fs);
        void print() const;
    private:
        string name;
        double price;
        int score;
};

Product::Product()
{ price = 0;
  score = 0;
}

bool Product::read(istream& fs)
{  cout << "Product: ";
   getline(fs, name);
   if (name == "") return false; // end of file
   fs >> price >> score;
   if (fs.fail())
      throw runtime_error("Error while reading product");
   string remainder;
   getline(fs, remainder);
   return true;
}

void Product::print() const
{ cout << name << " " << price << " " << score << endl;
}

void process_products(istream& fs)
{  vector<Product*> products;
   bool more = true;
   while (more)
   {  Product* p = new Product;
      try
      {  if (p->read(fs)) products.push_back(p);
         else more = false;
      }
      catch(runtime_error& r)  // runtime_error& r)
      {  cout << "1. " << r.what() << endl;
```

```
         for (int i = 0; i < products.size(); i++)
                                 delete products[i];
         throw;
         return;
       }
     }
   /* processing products */

   cout << "Print:" << endl;
   for (int i = 0; i < products.size(); i++)
   {  products[i]->print();
      delete products[i];
   }
}

 int main()
 {  try
    {  process_products(cin);
    }
    catch(runtime_error& r)  // runtime_error& r)
    { cout << "2. " << r.what() << endl;
    }
    return 0;
 }
```

Спецификации на изключенията

При дефинирането на функция, можем да определим кои изключения може да изхвърля тази функция.
Например:

```
void process_products(fstream& fs)
     throw (UnexpectedEndOfFile, bad_alloc)
```

означава, че само зададените две изключения са позволени на тази функция.

```
void process_products(fstream& fs)
     throw ()
```

задава забрана за всички изключения, а липса на **throw** означава разрешение за изключения от всеки тип.
Пример: except.cpp

- Stroustrup's example: `class Vector` (**`strous.cpp`**)
