Обработка исключительных ситуаций

Когда программа конструируется из раздельных модулей, и, особенно, когда эти модули находятся в независимо разработанных библиотеках, обработка ошибок должна быть разделена на две части:

- генерация информации о возникновении ошибочной ситуации, которая не может быть разрешена локально;
- обработка ошибок, обнаруженных в других местах.

```
try { ... } <список реакций>


  catch (объявление ситуации) { ... }
[ catch (объявление ситуации) { ... } ... ]


throw <выражение>;
```

```
terminate()

set_terminate()

abort()
```

```
FILE *open(char *fname)
 { FILE *f = fopen("fname, "r");
   if (!f) throw fname;
   return f;
  }
```

```
void main()
 { try
    { FILE *f1 = open("in1.txt");
      FILE *f2 = open("in2.txt");
     }
   catch (char *str)
    { printf("Impossible to open
                 file '%s'!\n", str);
      return;
     }
   ...
  }
```

```
class Ex1
 { private:
    int reason;
   public:
    Ex1() { ... }
    int Reason() { return reason; }
  };

class Ex2 { };
```

```
void f1()
 { ...
   if (...) throw Ex1(0);
   if (...) throw Ex1(2);
   ...
   if (...) throw Ex2();
  }

void f2()
{ ...
   if (...) throw Ex2();
  }
```

```
void main()
 { try
    { ...
      f1();
      ...
      f2();
      ...
     }
   catch(Ex1 ex)
    { switch (ex.Reason())
       { case 0: ...
         case 1: ...
         case 2: ...
        }
     }
   catch(Ex2 ex)
    { ... }
  }
```

```
try
 { throw E(); }
catch (H)
 { ... }
```

Обработчик будет вызван, если:

- *H* того же типа, что и *E*;
- *H* является однозначно доступным публичным базовым классом для *E*;
- *H* и *E* являются указателями, и 1 или 2 выполняется для типов, на которые они ссылаются;
- *H* является ссылкой, и 1 или 2 выполняется для типа, на который ссылается *H*.

```
class MathErr { ... };

class Overflow      : public MathErr
 { ... };

class Underflow     : public MathErr
 { ... };

class ZeroDivision : public MathErr
 { ... };
```

```
try
 { ... }
catch (Overflow)
 { ... }
catch (MathErr)
 { ... }
```

```
try
 { ... }
catch (MathErr)
 { if (...)
    ...
   else
    { ...
      throw;
     }
  }
```

```
try
 { // Делаем что-то }
catch (...)
 { // Обработка всех исключений }
```

```
try
 { }
catch (std::ios_base::failure)
 { }
catch (std::exception)
 { }
catch (...)
 { }
```

```
class Vector
 { ...
   public:
     class Size { ... };
     Vector(int n = 0);
   ...
  };

Vector::Vector(int n)
 { if (n < 0 || n > MAX_SIZE)
    throw Size();
   ...
  }
```

```
void f(Queue q)
 { try
    { for ( ; ; )
       { int x = q.Get();
         ...
       }
    }
   catch (Queue::Empty)
    { return; }
  }
```

```
int f(int n) throw (ex1, ex2);

int g(int n);

int h(int n) throw();
```

```
class Stack;

class StackEmpty
 { private:
    Stack *stack;
   public:
    StackEmpty(Stack *p) : stack(p) { }
    Stack* GetPtr() { return stack; }
  };

class StackFull
 { private:
    Stack *stack;
    int n;
   public:
    StackFull(Stack *p, int i) : stack(p), n(i) { }
    Stack* GetPtr() { return stack; }
    int GetValue()  { return n; }
  };
```

```
class Stack
 { private:
    enum { SIZE = 100 };
    int stack[SIZE];
    int *cur;
   public:
    Stack()  { cur = stack; }
    ~Stack() { }
    int  Push(int n);
    int  Pop();
    int  IsEmpty() const { return cur == stack; }
    int  operator >> (int& s) { s = Pop(); return s; }
    int  operator << (int s)  { return Push(s);      }
  };
```

```
int Stack::Push(int n)
 { if (cur - stack < SIZE)
    { *cur++ = n; return n; }
   else
    throw StackFull(this, n);
  }

int Stack::Pop()
 { if (cur != stack)
    return *--cur;
   else
    throw StackEmpty(this);
  }
```

```
void main()
 { Stack s;
   int n;

   try
    { s << 1;
      s << 2;
      s << 3;
      s << 4;
      s << 5;
      s >> n;
      printf("%d\n", n);
      s >> n;
      printf("%d\n", n);
      s >> n;
      printf("%d\n", n);
      s >> n;
      printf("%d\n", n);
     }
```

```
 catch (StackFull s)
  { printf("Attempt to put a value %d to the full
            stack at the address %p\n", s.GetValue(),
                                       s.GetPtr());
   }
 catch (StackEmpty s)
  { printf("Attempt to get a value from the empty
            stack at the address %p\n", s.GetPtr());
   }
}
```