Примерни програми

Любомир Чорбаджиев[1]
lchorbadjiev@elsys-bg.org

Revision : 1.4 $Date: 2005/10/11 11:01:20 $

Съдържание

Съдържание

1. Обработка на изключения

1.1. Обработка на грешки

Обработка на грешки

- По време на изпълнение на програмата дадена функция може да открие възникването на ненормална, грешна ситуация.
- Причината за възникването на такава ситуация може да бъде различна — неправилни входни данни, препълване на диска, изчерпване на наличната динамична памет, невъзможност да се отвори файл и т.н.
- По какъв начин функцията трябва да реагира на такава ситуация?

Обработка на грешки

- C-подход: функцията, открила ненормална ситуация да върне резултат, който сигнализира за наличието на грешка.
- Голяма част от функциите в стандартната C библиотека са организирани точно по този начин.

```
FILE* fopen(const char* filename,
            const char* mode);
int fputc(int c, FILE* file);
int fputs(const char* str, FILE* file);
int fgetc(FILE* file);
```

Обработка на грешки в класа Stack

Първоначална версия — липсва обработка на грешки.

```
1  class Stack {
2  ...
3  public:
4  ...
5    void push(int val) {
6      if(top_<STACK_SIZE) {
7        data_[top_++]=val;
8      }
9    }
10 ...
11 };
```

Обработка на грешки в класа Stack

```
  int push(int val) {
    if(top_<STACK_SIZE) {
      data_[top_++]=val;
      return 0;
    }
    return -1;// Грешка: стека е пълен
  }
```

Обработка на грешки в класа Stack

Първоначална версия — липсва обработка на грешки.

```
class Stack {
...
public:
...
  int pop(void) {
    if(top_>0) {
      return data_[--top_];
    }
    return 0;
  }
...
};
```

Обработка на грешки в класа Stack

```
  int pop(int& val) {
    if(top_>0) {
      val=data_[--top_];
      return 0;
    }
    return -1;// Грешка: стека е празен
  }
```

Обработка на грешки

- Разгледаният подход за обработка на грешки е тежък и тромав.
- При всяко извикване на функция, резултатът от тази функция трябва да се изследва за възможни настъпили грешки. Това прави кода на програмата труден за разбиране и поддържане.
- Друг недостатък на разглеждания подход е, че в него няма стандарти. Това прави трудно еднотипното обработване на грешки.

1.2. Генериране и обработка на изключения

Генериране и обработка на изключения

- Механизмът за обработката на изключения в C++ предоставя стандартни, вградени в езика средства за реагиране на ненормални, грешни ситуации по време на изпълнение програмата.
- Механизмът на изключенията предоставя еднообразен синтаксис и стил за обработка на грешки в програмата.
- Елиминира нуждата от изрични проверки за грешки и съсредоточава кода за обработка на грешки в отделни части на програмата.

Генериране на изключение

- При възникване на ненормална ситуация в програмата, програмистът сигнализира за настъпването ѝ чрез генерирането на изключение.
- Когато се генерира изключение, нормалното изпълнение на програмата се прекратява докато изключението не бъде обработено.
- В C++ за генериране на изключение се използва ключовата дума **throw**.

Генериране на изключение

```
class StackError { ... };
class Stack {
...
public:
...
  int pop(void) {
    if(top_<=0)
      throw StackError;
    return data_[--top_];
  }
```

```
...
}
```

Обработване на изключения

- Най-често изключенията в програмата се генерират и обработват от различни функции.
- След като изключението бъде обработено изпълнението на програмата продължава нормално. Възстановяването на изпълнението на програмата обаче става не от точката на генериране на изключението, а от точката, където изключението е било обработено.
- В C++ обработката на изключенията се изпълнява в **catch**-секции.

Обработване на изключение

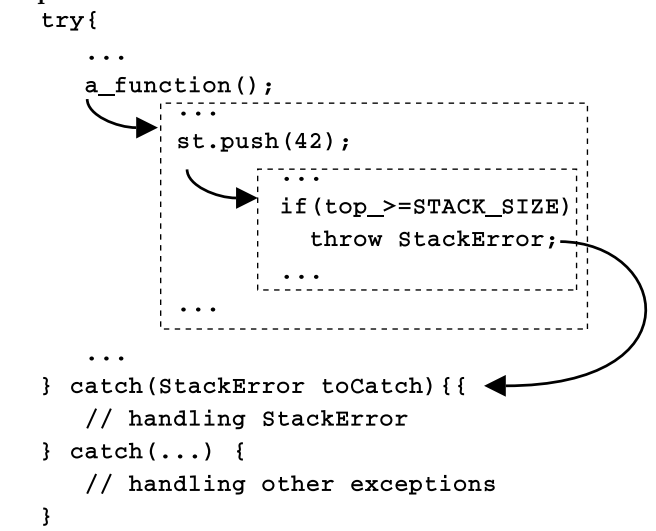
```
catch(StackError ex) {
  log_error(ex);
  exit(1);
}
```

Обработване на изключение

- Всяка една **catch**-секция трябва да се асоциира с **try**-блок. В един **try**-блок се групират един или повече оператори, които могат да генерират изключения с една или повече **catch**-секции.

```
try {
  // Използване на обекти от класа Stack
  ...
} catch(StackError ex) {
  // Обработка на грешка при използването на стека
  ...
} catch(...) {
  // Обработка на всички останали грешки
  ...
}
```


Обработване на изключение



1.3. Пример за използване на изключения

Пример за използване на изключения

```
class StackError {};
const int STACK_SIZE=10;
class Stack {
  int data_[STACK_SIZE];
  int top_;
public:
  Stack() {
    top_=0;
  }
  void push(int val) {
    if(top_>=STACK_SIZE)
      throw StackError();
    data_[top_++]=val;
  }
```

Пример за използване на изключения

```
  int pop(void) {
```

```
16      if(top_<=0)
17        throw StackError();
18      return data_[--top_];
19    }
20    bool is_empty() {
21      return top_==0;
22    }
23    bool is_full() {
24      return top_==STACK_SIZE;
25    }
26  };
```

Пример за използване на изключения

```
27  #include <cstdlib>
28  #include <iostream>
29  using namespace std;
30  int main(int arch, char* argv[]) {
31    char* msg="Hello Cruel World!";
32    char buff[10];
```

```
33    try {
34      Stack st;
35      for(char* p=msg;*p!='\0';p++)
36        st.push(*p);
37      char* p=buff;
38      while(!st.is_empty())
39        *p++=st.pop();
40      *p='\0';
41    } catch(StackError ex) {
42      cerr<<"StackError catched..."<<endl;
43      exit(1);
44    } catch(...) {
45      cerr<<"Unknown error catched..."<<endl;
46      exit(1);
47    }
48    return 0;
49  }
```


2. Рационални числа: class Rational

2.1. Дефиниция

Дефиниция на рационални числа

- Множеството на рационалните числа преставлява можеството на частните a/b, където a и b са цели числа и $b \neq 0$. Числото a се нарича *числител*, а числото b – *знаменател*.
- Примери за рационални числа:

$$\frac{1}{2},\ \frac{5}{4},\ \frac{-6}{2},\ \frac{-3}{-4}$$

- Рационалните числа се представят от отношението между числителя и знаменателя. Следователно, всяко рационално число може да бъде представено по различни начини. Например

$$\frac{3}{4} = \frac{6}{8} = \frac{12}{16} = \frac{15}{20} = \frac{18}{24}\ldots$$

  са различни преставяния на едно и също рационално число.

2.2. Редуцирана форма

Редуцирана форма

Нека е дадено рационалното число $\frac{a}{b}$. *Редуцирана форма* на това рационално число се нарича прествянето във вида $\frac{a'}{b'} = \frac{a}{b}$, за което е изпълнено следното:

$$a' = \frac{a}{\mathrm{GCD}(a,b)},\ b' = \frac{b}{\mathrm{GCD}(a,b)}, \tag{1}$$

където $\mathrm{GCD}(a,b)$ е най-големият общ делител на a и b.

2.3. Алгоритъм на Евклид за намиране на НОД

Алгоритъм на Евклид за намиране на НОД

Има различни алгоритми за намиране на най-голям общ делител (НОД). Един от най-простите и най-ефективни алгоритми е алгоритмът на Евклид.

1: **procedure** EUCLID(a, b) ▷ Намиране на НОД за a и b
2: $r \leftarrow a \bmod b$

```
3:    while r ≠ 0 do                    ▷ Ако остатъкът r е 0, то край
4:        a ← b
5:        b ← r
6:        r ← a mod b
7:    end while
8:    return b                          ▷ НОД е равен на стойността на b
9: end procedure
```

2.4. Нормална форма

Нормална форма

Рационалните числа могат да имат отрицателни числител и знаменател. Например:

$$\frac{-3}{-4} = \frac{3}{4}, \quad \frac{3}{-4} = \frac{-3}{4}. \tag{2}$$

Норамална форма на дадено рационалното число ще наричаме неговата редуцирана форма, в която знаменателят е положителен.

2.5. Класът `Rational`

```
1  #include <iostream>
2  using namespace std;
3
4  class RationalError{};
5
6  class Rational {
7    long num_, den_;
8
9    long gcd(long r, long s) {
10     while(s!=0) {
11       long temp=r;
12       r=s;
13       s=temp % s;
14     }
15     return r;
16   }
```

2.6. Класът `Rational`


```
18   void reduce(void) {
19     if(num_==0){
20       den_=1;
21     } else {
22       long tempnum=(num_<0)?-num_:num_;
23       long g=gcd(tempnum,den_);
24       if (g>1){
25         num_/=g;
26         den_/=g;
27       }
28     }
29   }
```

2.7. Класът `Rational`

```
31   void standardize(void) {
32     if(den_<0) {
33       den_=-den_;
34       num_=-num_;
35     }
36     reduce();
37   }
```

2.8. Класът `Rational`

```
38 public:
39   Rational(int num=0, int den=1){
40     num_=num;
41     den_=den;
42
43     if(den_==0)
44       throw RationalError();
45     standardize();
46   }
47
48   long get_numerator() {return num_;}
49   long get_denominator() {return den_;}
50
51   void dump() {
52     cout << "(" << num_ << "/" << den_ << ")";
53   }
```

2.9. Събиране и изваждане

Събиране и изваждане

– Сумата на две рационални числа $\frac{a_1}{b_1}$ и $\frac{a_2}{b_2}$ се нарича рационалното число $\frac{A}{B}$, което се получава по следния начин:

$$\frac{A}{B} = \frac{a_1}{b_1} + \frac{a_2}{b_2} = \frac{a_1 b_2 + a_2 b_1}{b_1 b_2}. \tag{3}$$

– Разликата на две рационални числа $\frac{a_1}{b_1}$ и $\frac{a_2}{b_2}$ се нарича рационалното число $\frac{A}{B}$, което се получава по следния начин:

$$\frac{A}{B} = \frac{a_1}{b_1} - \frac{a_2}{b_2} = \frac{a_1 b_2 - a_2 b_1}{b_1 b_2}. \tag{4}$$

2.10. Умножение и делене

Умножение и делене

– Произведение на две рационални числа $\frac{a_1}{b_1}$ и $\frac{a_2}{b_2}$ се нарича рационалното число $\frac{A}{B}$, което се получава по следния начин:

$$\frac{A}{B} = \frac{a_1}{b_1} \cdot \frac{a_2}{b_2} = \frac{a_1 a_2}{b_1 b_2}. \tag{5}$$

– Частно на две рационални числа $\frac{a_1}{b_1}$ и $\frac{a_2}{b_2}$ се нарича рационалното число $\frac{A}{B}$, което се получава по следния начин:

$$\frac{A}{B} = \frac{a_1}{b_1} : \frac{a_2}{b_2} = \frac{a_1 b_2}{b_1 a_2}. \tag{6}$$

2.11. Класът `Rational`

```
55   void add(Rational r) {
56     num_=num_*r.den_+den_*r.num_;
57     den_=den_*r.den_;
58     standardize();
59   }
60   void sub(Rational r) {
61     num_=num_*r.den_-den_*r.num_;
62     den_=den_*r.den_;
63     standardize();
64   }
```



```
65   void multiply(Rational r) {
66     num_*=r.num_;
67     den_*=r.den_;
68     standardize();
69   }
70   void device(Rational r) {
71     num_*=r.den_;
72     den_*=r.num_;
73     standardize();
74   }
75 };
```

```
77 int main(int argc, char* argv[]) {
78   Rational r(1,2),p(2,3), q(4,2), s(-3,-9);
79
80   r.dump();p.dump();q.dump();s.dump();
81   cout << endl;
82
83   r.add(p);
84   r.dump();
85   cout << endl;
```

```
87   p.add(s);
88   p.dump();
89   cout << endl;
90
91   r.multiply(q);
92   r.dump();
93   cout << endl;
94
95   return 0;
96 }
```

3. Псевдо-случайни числа: namespace Random

3.1. Методи за генериране на псевдо-случайни числа

Генериране на псевдо-случайни числа

> *...random numbers should not be generated with a method choosen at random.* Donald Knuth, The Art of Computer Programming, volume 2.

- *Линеен конгруентен метод*: Същността на метода се заключава с следното — избират четири "магични" числа:
  - m — модул, $m > 0$;
  - a — множител, $0 < a < m$;
  - c — добавка, $0 < c < m$;
  - X_0 — начална стойност, $0 \leq X_0 < m$.

  Желаната последователност от псевдо-случайни числа X_n се получава, като се използва формулата:

$$X_n = (aX_{n-1} + c) \bmod m, n > 0. \quad (7)$$

Стандартни функции за генериране на псевдо-случайни числа

- В стандартната `C` и `C++`-библиотека са дефинирани набор от функции, които генерират последователности от псевдо-случайни числа — `rand()`, `srand()`, `RAND_MAX`. Тези функции са декларирани в заглавия файл `<stdlib.h>` и `<cstdlib>` съответно.
- **int** `RAND_MAX` — най-голямото случайно число, което може да се генерира от функцията за генериране на случайни числа.
- **int** `rand(void)` — следващото псевдо-случайно число. Стойностите, които връща тази функция са между `0` и `RAND_MAX`.

Стандартни функции за генериране на псевдо-случайни числа

- **void** `srand(int seed)` — установява стартова стойност за серията от псевдо-случайни числа, генерирани от функцията `rand()`.
- Ако функцията `rand()` се извика без предварително да е извикана `srand()`, то функцията `rand()` ще генерира едни и същи последователности от случайни числа при всяко пускане на програмата.
- За да бъдат различни последователностите от псевдо-случайни числа обикновено се вика `srand(time())`.

3.2. namespace Random

Заглавен файл Random.hpp

```
1  #ifndef RANDOM_HPP__
2  #define RANDOM_HPP__
3
4  namespace Random {
5    void init(unsigned long seed=0);
6    int next_int();
7    int next_int(int max);
8    int next_int(int min, int max);
9    double next_double();
10 };
11
12 #endif
```

Файл Random.cpp

```
#include <cstdlib>
#include <ctime>

#include "Random.hpp"
namespace Random {
  void init(unsigned long seed) {
    unsigned int s =
      seed==0?std::time(0):seed;
    std::srand(s);
  }

  int next_int() {
    return std::rand();
  }
```

Файл `Random.cpp`

```
  double next_double() {
    return static_cast<double>(next_int())/
      static_cast<double>(RAND_MAX);
  }

  int next_int(int max) {
    return next_int() % max;
  }

  int next_int(int min, int max) {
    return min + next_int() % (max-min);
  }
};
```

4. Примери за използване на **namespace** `Random`

4.1. Хвърляне на монета

Хвърляне на монета

– Като пример за използване на **namespace** `Random` нека разгледаме задачата за n хвърляния на монета. Въпросът е колко пъти ще се падне ези?

– За да имитираме хвърляне на монета генерираме случайно цяло число, като използваме `Random::next_int(2)`. Стойностите, които ще връща тази функция са `0` или `1`. Приемаме, че ако генерираното псевдо-случайно число е `1`, то това означава, че се е паднало ези.

Хвърляне на монета

– Да приемем, че искаме да хвърлим монетата `10` пъти. Тогава следният фрагмент пресмята колко пъти се е паднало ези:

```
head_count=0;
for(int i=0;i<10;i++) {
  head_count+=Random::next_int(2);
}
```

– Тъй като хвърлянето на монета е случайно събитие, то при всяко пускане на горния фрагмент стойността на `head_count` ще бъде случайно цяло число в интервала $[0, 10]$.

– За да наблюдаваме някакви закономерности в този експеримент трябва да го направим голям брой пъти.

Инициализация

```
#include <iostream>
#include <iomanip>
using namespace std;

#include "Random.hpp"
const unsigned int COINS_COUNT=10;
const unsigned int TOSS_COUNT=10000;

int main(int argc, char* argv[]) {
  Random::init();
  int head[COINS_COUNT+1];
  for(unsigned i=0;i<COINS_COUNT+1;i++)
    head[i]=0;
```

Хвърляне на монета

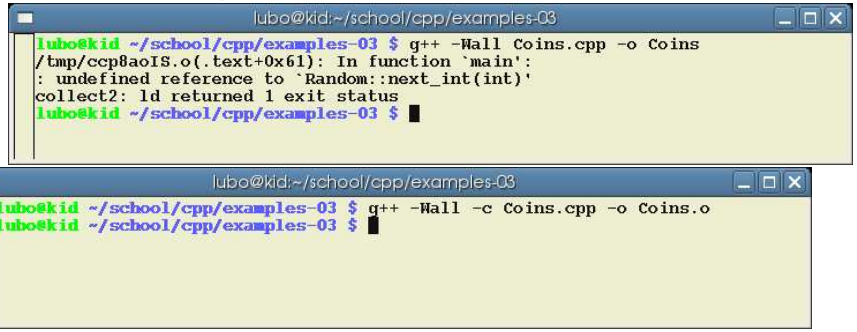
```
  for(unsigned i=0;i<TOSS_COUNT;i++) {
    int head_count=0;
```

```
16      for(unsigned j=0;j<COINS_COUNT;j++)
17        head_count+=Random::next_int(2);
18      head[head_count]++;
19    }
```

Извеждане на резулатата

```
20    for(unsigned i=0;i<COINS_COUNT+1;i++) {
21      int pos=static_cast<int>(
22          (static_cast<double>(head[i])/
23           TOSS_COUNT)*100.0);
24      cout << setw(2)<< i << "␣"
25           << setw(7) << head[i];
26      for(int j=0;j<pos;j++) {
27        cout << "␣";
28      }
29      cout << "*" << endl;
30    }
31
32    return 0;
33  }
```

Разделно компилиране



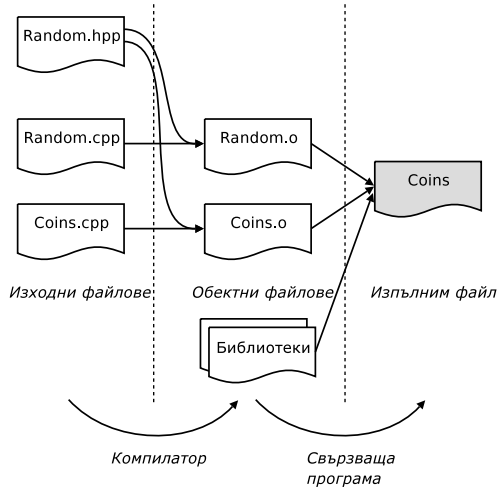
Разделно компилиране

- Следната команда създава обектен файл `Coins.o`.

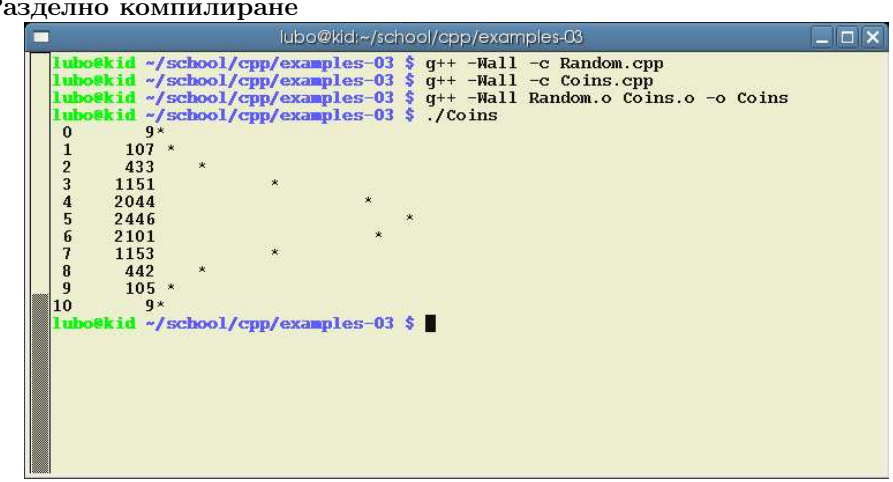
  `g++ -Wall -c Coins.cpp`

- Следната команда се опитва да създаде изпълним файл `Coins`.

`g++ -Wall Coins.cpp -o Coins`



Разделно компилиране



4.2. Тесте карти

Тесте карти

- Като пример за използване на **namespace** `Random` нека разработим модел на тесте от 52 карти.

- Всяка една карта принадлежи на определен цвят: спатии (clubs), каро (diamonds), купа (hearts) или пика (spades).
- Всяка карта освен принадлежността ѝ към определен цвят се характеризира и със стойност, която може да бъде: асо (ace), 2, 3,...,10, вале (jack), дама (queen), поп (king).
- Целта е да създадем клас `CardDeck`, който да моделира тесте от 52 карти. Класът трябва да има методи за разместване на картите и за раздаване на карти.

Дефиниция на enum `Suit`

```
#include <iostream>
using namespace std;
#include "Random.hpp"

enum Suit {
  CLUBS=0, DIAMONDS,
  HEARTS, SPADES
};
```

Дефиниция на class `Card`

```
class Card {
  Suit suit_;
  int face_;
public:
  void set_card(int card) {
    suit_=static_cast<Suit>(card/13);
    face_=card%13;
  }
  Card(int card=0) {
    set_card(card);
  }
```

```
  Suit get_suit() {
    return suit_;
  }
  int get_face() {
    return face_;
  }
```



```
  void print() {
    static const char FACES[][3]={
      "A","2","3","4","5","6","7","8","9",
      "10","J","Q","K"};
    static const char SUITS[][9]={
      "Clubs","Diamonds","Hearts","Spades"};
    cout << FACES[face_]
         << "(" << SUITS[suit_] << ")";
  }
};
```

Дефиниция на class `Deck`

```
class Deck {
  Card cards_[52];
  int next_;
public:
  Deck(void) {
    for(int i=0;i<52;i++) {
      cards_[i].set_card(i);
    }
    next_=0;
  }
```

Дефиниция на class `Deck`

```
  void shuffle() {
    for(int i=0;i<52;i++) {
      int rint=Random::next_int(52);
      Card temp=cards_[rint];
      cards_[rint]=cards_[i];
      cards_[i]=temp;
    }
    next_=0;
  }

  Card deal_one() {
    return cards_[next_++];
  }
};
```

Главна функция

```
65 int main(int argc, char* argv[]) {
66   Random::init();
67
68   Deck my_deck;
69   my_deck.shuffle();
70   for(int i=0;i<5;i++) {
71     Card c=my_deck.deal_one();
72     c.print();
73     cout << endl;
74   }
75
76   return 0;
77 }
```

Разделно компилиране