7 例外処理

7–1 C言語assert()関数を用いた例外処理

一般に、

◇ プログラムや関数には正しく動作するための前提条件がある。
　⟹ それぞれの前提条件を守った上での利用が必要

◇ 前提条件が満たされなくなる状況を完全に除去するのは難しい。

⇨ C言語では、
　①assert()関数によって想定外の状況(例外)の生起を調べ、
　②そういった状況を検出したらすぐに強制終了
できる様になっている。

例7. 1 (C言語assert()を用いた例外処理,添字チェックの機能を備えた配

例6.2で「添字の有効性をチェックする機能を備えた配列」のテンプレートクラスSafeArray を定義する際、既に

①assert()関数を用いて例外発生の検出

②その場合の強制終了

の措置を講じている。

⇨ ソースコードを再掲すると...

(下線 ... assert()関数を呼び出した箇所)

```
[motoki@x205a]$  cat -n SafeArray.h
     1 /* 「添字の有効性をチェックする機能を備えた配列」をインス */
     2 /* タンスとする型パラメータ付きクラス SafeArray の仕様部 *
     3
     4 #ifndef ___Class_SafeArray
     5 #define ___Class_SafeArray
     6
     7 template<typename TYPE>
     8 class SafeArray {
```

```
 9   TYPE* basePtr;          //pointer to the 1st element
10   int size;               //array size
11 public:
12   explicit SafeArray(int n=100);        //create a Safe
13   SafeArray(const SafeArray<TYPE>& a);  //copy construc
14   SafeArray(const TYPE a[], int n);     //copy a standa
15   ~SafeArray() { delete[] basePtr; }
16   int getSize() { return size; }
17   typedef TYPE* iterator;
18   iterator begin() { return basePtr; }
19   iterator end() { return basePtr + size; }
20   TYPE& operator[](int i);    //range-checked element
21   SafeArray<TYPE>& operator=(const SafeArray<TYPE>& a);
22 };
23
24 //=============================================
25 /* 「添字の有効性をチェックする機能を備えた配列」をインス */
```

```
26 /* タンスとする型パラメータ付きクラス SafeArray の実装部 *
27
28 #include <cassert>
29
30 template<typename TYPE> //create a SafeArray of size n
31 SafeArray<TYPE>::SafeArray(int n): size(n)
32 {
33   assert(n > 0);
34   basePtr = new TYPE[size];
35   assert(basePtr != 0);
36 }
37
38 template<typename TYPE>          //copy constructor
39 SafeArray<TYPE>::SafeArray(const SafeArray<TYPE>& a)
40 {
41   size = a.size;
42   basePtr = new TYPE[size];
```

```
43    assert(basePtr != 0);
44    for (int i=0; i<size; ++i)
45      basePtr[i] = a.basePtr[i];
46  }
47
48  template<typename TYPE>          //copy a standard array
49  SafeArray<TYPE>::SafeArray(const TYPE a[], int n)
50  {
51    assert(n > 0);
52    size = n;
53    basePtr = new TYPE[size];
54    assert(basePtr != 0);
55    for (int i=0; i<size; ++i)
56      basePtr[i] = a[i];
57  }
58
59  template<typename TYPE>          //range-checked element
```

```
60 TYPE& SafeArray<TYPE>::operator[](int i)
61 {
62   assert (0<=i && i<size);
63   return basePtr[i];
64 }
65
66 template<typename TYPE>          //assignment operator
67 SafeArray<TYPE>& SafeArray<TYPE>::
                      operator=(const SafeArray<TYPE>& a)
68 {
69   assert (a.size == size);
70   for (int i=0; i<size; ++i)
71     basePtr[i] = a.basePtr[i];
72   return *this;
73 }
74
75 #endif
```

[motoki@x205a]$

これに関して、

- テンプレートクラス SafeArray が上の様に定義されていれば 、
  例外発生によって次の様な出力結果。

(下線 ... 例外発生を引き起こす箇所)

```
[motoki@x205a]$  cat -n useSafeArray_verAssert_1.cpp
     1 // テンプレートクラス SafeArray を利用した時に
                                        // 例外が起きる例
     2
     3 #include <iostream>
     4 #include "SafeArray_verAssert.h"
     5 using namespace std;
     6
```

```
 7 int main()
 8 {
 9   SafeArray<int> a(100);
10
11   for (int i=0; i<a.getSize(); i++) {
12     a[i] = i+10;
13   }
14   cout << "a = { " << a[0] << ", " << a[1]
                                    << ", " << a[2]
15        << ", ..., " << a[99] << " }" << endl;
16   cout << "---" << endl;
17
18   cout << "a = { " << a[0] << ", " << a[1]
                                    << ", " << a[2]
19        << ", ..., " << a[100] << " }" << endl;
20 }
```

assert()関数による例外検出→実行中止

```
[motoki@x205a]$  g++ useSafeArray_verAssert_1.cpp
[motoki@x205a]$  ./a.out
a = { 10, 11, 12, ..., 109 }
---
a.out: SafeArray_verAssert.h:62: TYPE&
                                 SafeArray<TYPE>::operator[](int)
     [with TYPE = int]: Assertion ‘0<=i && i<size’ failed.
```

中止

```
[motoki@x205a]$  cat -n useSafeArray_verAssert_2.cpp
     1 // テンプレートクラス SafeArray を利用した時に
                                          // 例外が起きる例
     2
     3 #include <iostream>
     4 #include "SafeArray_verAssert.h"
     5 using namespace std;
     6
     7 int main()
```

```
 8 {
 9   SafeArray<int>* a;
10
11   a = new SafeArray<int>(100000000);
12   for (int i=0; i<(*a).getSize(); i++) {
13     (*a)[i] = i+10;
14   }
15   cout << "*a = { " << (*a)[0] << ", " << (*a)[1]
                                    << ", " << (*a)[2]
16        << ", ..., " << (*a)[99999999] << " }"
                                          << endl;
17   delete a;          // (*a).~SafeArray<int>(); を伴う
18   cout << "---" << endl;          ↑
19                                  オブジェクト消滅なので
20   a = new SafeArray<int>(1000000000);
            ↑例外検出され例外オブジェクト発生→実行中止
21   for (int i=0; i<(*a).getSize(); i++) {
```

```
    22      (*a)[i] = i+10;
    23    }
    24    cout << "*a = { " << (*a)[0] << ", " << (*a)[1]
                                          << ", " << (*a)[2]
    25         << ", ..., " << (*a)[999999999] << " }"
                                                  << endl;
    26    delete a;          // (*a).~SafeArray<int>(); を伴う
    27 }
[motoki@x205a]$  g++ useSafeArray_verAssert_2.cpp
[motoki@x205a]$  ./a.out
*a = { 10, 11, 12, ..., 100000009 }
---
terminate called after                    オブジェクト
                    throwing an instance of 'std::bad_alloc'
  what():  std::bad_alloc
中止
[motoki@x205a]$  cat -n useSafeArray_verAssert_3.cpp
```

例外クラス
↓

```
 1 // テンプレートクラス SafeArray を利用した時に
                                        // 例外が起きる例
 2
 3 #include <iostream>
 4 #include "SafeArray_verAssert.h"
 5 using namespace std;
 6
 7 int main()
 8 {
 9   SafeArray<int> a(100);
10   for (int i=0; i<a.getSize(); i++) {
11     a[i] = i+10;
12   }
13   cout << "a = { " << a[0] << ", " << a[1]
                                    << ", " << a[2]
14        << ", ..., " << a[99] << " }" << endl;
15   cout << "---" << endl;
```

assert()関数による例外検出→実行中止

```
16
17    SafeArray<int> b(-100);
18    for (int i=0; i<b.getSize(); i++) {
19      b[i] = i+10;
20    }
21    cout << "b = { " << b[0] << ", " << b[1]
                                    << ", " << b[2]
22          << ", ..., " << b[99] << " }" << endl;
23 }
[motoki@x205a]$  g++ useSafeArray_verAssert_3.cpp
[motoki@x205a]$  ./a.out
a = { 10, 11, 12, ..., 109 }
---
a.out: SafeArray_verAssert.h:33:
                              SafeArray<TYPE>::SafeArray(int)
              [with TYPE = int]: Assertion ‘n > 0’ failed.
```

中止

```
[motoki@x205a]$
```

7–2 C++言語における例外処理, 例外クラスの

- C言語の下で
  assert() 関数を使えば、
  ◇ 例外的な状況を検出すると即座に「強制終了」
  ◇ プログラマ自身が例外を処理する選択肢はない。

- C言語の下で
  assert() 関数を使わずに頑健なプログラムを作ろうとすると、
  エラー検出とその回復のコードがプログラムのあちこちに埋め込まれ、
  本来の処理の流れが見えなくなってしまう恐れがある。

- C++言語 では、
  本来の処理の流れの明瞭さを損なわずにエラー検出等を行うために、
  例外処理の機構が導入されている。すなわち、

. . . . . .

- C++言語 では、
  本来の処理の流れの明瞭さを損なわずにエラー検出等を行うために、
  例外処理の機構が導入されている。すなわち、

  ① 想定外の状況が実行中に検出されると、
  $\longrightarrow$ 例外と呼ばれるデータを生成し送出

  ② 例外データが送出されると、
  $\longrightarrow$ 例外を処理してくれるコードが探される。
  (必要なら関数呼出しの履歴を逆にたどる等もする)

  ③ 例外処理のコードが
  見つかれば $\longrightarrow$ それを実行。

  見つからなければ、 $\longrightarrow$ C++言語のエラー処理ルーチン が
  エラー報告を出して実行を中止。

例7. 2 (C++における例外処理,添字チェックの機能を備えた配列のクラ

例7.1で示したテンプレートクラスSafeArray の定義中の

assert() 関数呼出し部を、

C++言語の例外処理の機構に沿って書き換える

と次の様になる。

(比較のため、元々あったassert() 関数呼出しはコメントアウトだけ)

```
[motoki@x205a]$  cat -n SafeArray_verThrow.h
     1 /* 「添字の有効性をチェックする機能を備えた配列」をインス */
     2 /* タンスとする型パラメータ付きクラス SafeArray の仕様部 *
     3
     4 #ifndef ___Class_SafeArray
     5 #define ___Class_SafeArray
     6
     7 template<typename TYPE>
     8 class SafeArray {
     9   TYPE* basePtr;                    //pointer to the 1st
    10   int size;                         //array size
```

```
11 public:
12   //例外クラスの定義
13   class IndexOutOfBounds {
14   public:
15     int index;
16     IndexOutOfBounds(int index): index(index) {}
17   };
18   //関数群
19   explicit SafeArray(int n=100);          //create a Safe
20   SafeArray(const SafeArray<TYPE>& a);    //copy constru
21   SafeArray(const TYPE a[], int n);       //copy a standa
22   ~SafeArray() { delete[] basePtr; }
23   int getSize() { return size; }
24   typedef TYPE* iterator;
                    //「using iterator = TYPE*;」ではダメ
25   iterator begin() { return basePtr; }
```

SafeArray クラスに付随したものなので、このクラスの中で局所的に定義

←

←配列の添字として指定された範囲外の値

```
26    iterator end() { return basePtr + size; }
27    TYPE& operator[](int i);     //range-checked element
28    SafeArray<TYPE>& operator=(const SafeArray<TYPE>& a);
29 };
30
31 //=============================================================
32 /* 「添字の有効性をチェックする機能を備えた配列」をインス */
33 /* タンスとする型パラメータ付きクラス SafeArray の実装部 */
34
35 //#include <cassert>
36
```

```
37 template<typename TYPE> //create a SafeArray of size n
38 SafeArray<TYPE>::SafeArray(int n): size(n)
39 {
40   //assert(n > 0);
41   if (n <= 0)
42     throw "Invalid safeArray size was specified in a co
```

↑ 例外データ(エラーメッセージ)を送出して、一連の例外処理を引き起こす

```
43   basePtr = new TYPE[size];
                          //失敗するとstd::bad_alloc例外を送出
44   //assert(basePtr !=0);
```

← 古いC++では左のassert()の代わりの処置が必要

```
45 }
46
47 template<typename TYPE>          //copy constructor
48 SafeArray<TYPE>::SafeArray(const SafeArray<TYPE>& a)
49 {
```

```
50    size = a.size;
51    basePtr = new TYPE[size];
                          //失敗するとstd::bad_alloc例外を送出
52    //assert(basePtr != 0);  ←古いC++では左のassert()の代わりの処置が必要
53    for (int i=0; i<size; ++i)
54      basePtr[i] = a.basePtr[i];
55  }
56
57  template<typename TYPE>          //copy a standard array
58  SafeArray<TYPE>::SafeArray(const TYPE a[], int n)
59  {
60    //assert(n > 0);
61    if (n <= 0)
62      throw "Invalid safeArray size was specified in a co
        ↑          例外データ(エラーメッセージ)を送出して、一連の例外処理を引き起こす
63    size = n;
```

```
64    basePtr = new TYPE[size];
                          //失敗するとstd::bad_alloc例外を送出
65    //assert(basePtr != 0);  ←古いC++では左のassert()の代わりの処置が必要
66    for (int i=0; i<size; ++i)
67      basePtr[i] = a[i];
68  }
69
70  template<typename TYPE>              //range-checked element
71  TYPE& SafeArray<TYPE>::operator[](int i)
72  {
73    //assert (0<=i && i<size);
74    if (i<0 || size<=i)
75      throw IndexOutOfBounds(i); //例外オブジェクトを送出
            ↑                    例外オブジェクトを送出して、一連の例外処理を引き起こす
76    return basePtr[i];
77  }
```

```
78
79 template<typename TYPE>          //assignment operator
80 SafeArray<TYPE>& SafeArray<TYPE>::
                        operator=(const SafeArray<TYPE>& a)
81 {
82   //assert (a.size == size);
83   if (a.size != size)
84     throw "Assignment was tried between different size
         ↑          例外データ(エラーメッセージ)を送出して、
                    一連の例外処理を引き起こす
85   for (int i=0; i<size; ++i)
86     basePtr[i] = a.basePtr[i];
87   return *this;
88 }
89
90 #endif
[motoki@x205a]$
```

これに関して、

- throw文は、
  - ◇ ①プログラム実行の通常の流れを止め、
    ②検出された例外を表すデータを生成し、
    ③その例外データを処理するコードを探す、
    という例外処理の一連の動作を引き起こす。
  - ◇ 例外データとしてint型を始め任意の型のデータを送出可

- テンプレートクラス SafeArray が上の様に定義されていれば 、
  例外処理のコードも組み込んだ次の様なプログラムを書ける。

```
[motoki@x205a]$  cat -n useSafeArray_verThrow.cpp
     1 // テンプレートクラス SafeArray を利用した時に
                                          // 例外が起きる例
     2
     3 #include <iostream>
     4 #include <cstdlib>        // for abort()
```

```
 5 #include "SafeArray_verThrow.h"
 6 using namespace std;
 7
 8 int main()
 9 {
10   try {          tryブロック,例外データ送出を監視する区間
11     SafeArray<int> a(100);
12
13     for (int i=0; i<a.getSize(); i++) {
14       a[i] = i+10;
15     }
16     cout << "a = { " << a[0] << ", " << a[1]
                                    << ", " << a[2]
17          << ", ..., " << a[99] << " }" << endl;
18     cout << "---" << endl;
19
20     cout << "a = { " << a[0] << ", " << a[1]
```

```
                                        << ", " << a[2]
21          << ", ..., " << a[100] << " }" << endl;
22    }  IndexOutOfBounds型例外オブジェクト送出→例外処理
23    catch (const bad_alloc& x) {                  例外ハンドラ
                          ↑捕捉された例外データと
                            型が合致するか調べられる
24      cerr << "bad_alloc exception was thrown."
        ↑標準エラー出力                              << endl;
25      //abort();          ←通常は強制終了だが、ここでは次
                              の例を示すためコメントアウト
26    }
27    catch (const SafeArray<int>::IndexOutOfBounds& e){
28      cerr << "IndexOutOfBounds exception: index="
                                        << e.index << endl;
29      //abort();
30    }
31    catch (const char* error) {
32      cerr << error << endl;
```

```
33      //abort();
34    }
35    catch ( ... ) {          ←全てに合致,ワイルドカード
36      cerr << "Some exception was caught." << endl;
37      //abort();
38    }          適用可能な例外ハンドラがない場合は、tryブロッ
                 クを含む関数の呼び出し元の関数内で探される
39    cout << "=====================================
40
41    try {
42      SafeArray<int> a(100000000);
43      for (int i=0; i<a.getSize(); i++) {
44        a[i] = i+10;
45      }
46      cout << "a = { " << a[0] << ", " << a[1]
                                          << ", " << a[2]
47           << ", ..., " << a[99999999] << " }"
```

```
                                                    << endl;
48      cout << "---" << endl;
49
50      SafeArray<int> b(1000000000);
                  bad_alloc型例外オブジェクト送出→例外処理
51      for (int i=0; i<b.getSize(); i++) {
52        b[i] = i+10;
53      }
54      cout << "b = { " << b[0] << ", " << b[1]
                                          << ", " << b[2]
55           << ", ..., " << b[999999999] << " }"
                                                    << endl;
56    }
57    catch (const bad_alloc& x) {
58      cerr << "bad_alloc exception was thrown."
                                                    << endl;
59      //abort();
```

```
60    }
61    catch (const SafeArray<int>::IndexOutOfBounds& e){
62      cerr << "IndexOutOfBounds exception: index="
                                          << e.index << endl;
63      //abort();
64    }
65    catch (const char* error) {
66      cerr << error << endl;
67      //abort();
68    }
69    catch ( ... ) {
70      cerr << "Some exception was caught." << endl;
71      //abort();
72    }
73    cout << "=====================================
74
75    try {
```

```
76      SafeArray<int> a(100);
77      for (int i=0; i<a.getSize(); i++) {
78        a[i] = i+10;
79      }
80      cout << "a = { " << a[0] << ", " << a[1]
                                        << ", " << a[2]
81           << ", ..., " << a[99] << " }" << endl;
82      cout << "---" << endl;
83
84      SafeArray<int> b(-100);
                例外データ(エラーメッセージ)送出→例外処理
85      for (int i=0; i<b.getSize(); i++) {
86        b[i] = i+10;
87      }
88      cout << "b = { " << b[0] << ", " << b[1]
                                        << ", " << b[2]
89           << ", ..., " << b[99] << " }" << endl;
```

```
90    }
91    catch (const bad_alloc& x) {
92      cerr << "bad_alloc exception was thrown."
                                                << endl;
93      //abort();
94    }
95    catch (const SafeArray<int>::IndexOutOfBounds& e){
96      cerr << "IndexOutOfBounds exception: index="
                                          << e.index << endl;
97      //abort();
98    }
99    catch (const char* error) {
100     cerr << error << endl;
101     //abort();
102   }
103   catch ( ... ) {
104     cerr << "Some exception was caught." << endl;
```

```
105     //abort();
106   }
107   cout << "=======================================
108
109   try {
110     SafeArray<int> a(100), b(100), c(200);
111     for (int i=0; i<a.getSize(); i++) {
112       a[i] = i+10;
113     }
114     b = a;
115     cout << "b = { " << b[0] << ", " << b[1]
                                          << ", " << b[2]
116          << ", ..., " << b[99] << " }" << endl;
117     cout << "---" << endl;
118
119     c = a;  例外データ(エラーメッセージ)送出→例外処理
120     cout << "c = { " << c[0] << ", " << c[1]
```

```
                                                << ", " << c[2]
121             << ", ..., " << c[99] << " }" << endl;
122       cout << "---" << endl;
123     }
124     catch (const bad_alloc& x) {
125       cerr << "bad_alloc exception was thrown."
                                                    << endl;
126       abort();
127     }
128     catch (const SafeArray<int>::IndexOutOfBounds& e){
129       cerr << "IndexOutOfBounds exception: index="
                                          << e.index << endl;
130       abort();
131     }
132     catch (const char* error) {
133       cerr << error << endl;
134       abort();                                  ←強制終了
```

```
   135   }
   136   catch ( ... ) {
   137     cerr << "Some exception was caught." << endl;
   138     abort();
   139   }
   140 }
[motoki@x205a]$  g++ useSafeArray_verThrow.cpp
[motoki@x205a]$  ./a.out
a = { 10, 11, 12, ..., 109 }
---
IndexOutOfBounds exception: index=100
================================================
a = { 10, 11, 12, ..., 100000009 }
---
bad_alloc exception was thrown.
================================================
a = { 10, 11, 12, ..., 109 }
```

```
---
Invalid safeArray size was specified in a constructor.
=============================================
b = { 10, 11, 12, ..., 109 }
---
Assignment was tried between different size safeArray object
```

中止 ←abort()関数による強制終了

```
[motoki@x205a]$
```

ここで、

◇ プログラム **23～38行目,57～72行目,91～106行目,124～139行目** に置かれた例外ハンドラ群は全て同じものである。

そうなった理由：一般に、

- 定義したクラスは色々な用途に使う可能性
- 例外処理の内容も利用毎に異なる可能性

⇨
- `SafeArray`テンプレートクラスの定義中に`try-chatch`の構文を置くのを避け、
  - クラスを利用するプログラム上で例外処理

C++言語における例外処理の機構(まとめ):

本来の処理の流れの見通しを保った上で例外処理を行いたい場合、C++言語では次の形のコードを書く。

```
try {
    本来の処理
}                                              } tryブロック
catch ( 適用可能な例外データの指定 ) {
    発生した例外に対する処置
}                                              } 例外ハンドラ
        ⋮
catch ( 適用可能な例外データの指定 ) {
    発生した例外に対する処置
}                                              } 例外ハンドラ
```

ここで、

- tryブロックの中には 、基本的には

  > 例外的状況の発生を考慮に入れない本来の処理

  を書き、更にその中の所々に

  > 点検を行い、もし想定外の状況になっていれば、
  > ①検出された例外を表すデータ (例外データ) を生成し、
  > ②それをthrow文を用いて送出する
  > コード

  を挿入する。但し、例外データを生成し送出する際のthrow文は次の形式で書く。

  ```
  throw 式 ;
  ```

- tryブロック直後に並んだ 例外ハンドラは 、
  - ◇ tryブロック内で例外データが送出された場合のためのコードで、
  - ◇ 送出された例外データを分担して処理する。

キーワードcatchの直後の 適用可能な例外データの指定 の部分は
◇ 例外ハンドラの対応可能な例外データの型を明示するもので、
◇ 次の3つの形を基本として必要に応じて const 指定や参照宣言

データ型 変数名(仮引数)
データ型
...

この内
1番目の書き方の場合... 例外データと仮引数の結合が為された上で
catchブロック内の例外処理が施される。
3番目の書き方 ... ワイルドカードを表す。

- catch (適用可能な例外データの指定) 直後の catchブロックは 、
◇ 対応可能な例外データに対する処理内容を書く部分である。
このブロックの中では、
次の様に書いて処理中の例外データ の再送出も可。

```
throw;
```

- tryブロック以降の全体の処理の流れ は、次の通り。

① 本来の処理の流れを記述したtryブロックの処理が試行される。
エラー等検出 ⟶ 例外データが送出されtry ブロックの処理は中止

② tryブロック終了時点に例外データが発生していれば、
⟶ 例外ハンドラが上から順に調べられる。

◇ もし、キーワードcatchの直後の( )の中に送出された例外データの型と同じ(またはその基底クラスの)型が指定されていたり、文字列「...」が指定されていたりした場合は、その例外ハンドラが例外への対処を担当する例外ハンドラとなる。

キーワードcatchの直後の( )の中に仮引数も指定されていれば、
⟶ 例外データと仮引数の結合が為される。

そして、catchブロック内の例外処理が施される。

◇ 対処する例外ハンドラが見つからない場合は、
⟶ 実行中の関数を呼び出した関数に対して例外データが送出

例外クラスのライブラリ：C++言語では、例外オブジェクトを表すためのクラスが標準ライブラリの中に用意されている。例えば次の通り。

exception ··· 全ての例外クラスの基底クラスで、ヘッダ<exception>で定義されている。このクラスの派生クラスとして次のクラスが用意されている。

- bad_exception ··· 例外指定に対する違反を表すオブジェクトのクラスで、ヘッダ<exception>で定義されている。
- bad_alloc ··· 記憶域確保の失敗を表すオブジェクトのクラスで、ヘッダ<new> で定義されている。
- bad_cast ··· 動的キャストの失敗を表すオブジェクトのクラスで、ヘッダ<typeinfo> で定義されている。
- bad_typeid ··· typeid式中に空ポインタが含まれることを表すオブジェクトのクラスで、ヘッダ<typeinfo>で定義されている。

- logic_error ··· プログラム実行前に検出可能な論理エラーを表すオブジェクトのクラスで、ヘッダ<stdexcept> で定義されている。このクラスの派生クラスとして更に次のクラスが用意されている。
  - ◇ domain_error ··· ドメインエラー (<stdexcept>)
  - ◇ invalid_argument ··· 不正な実引数 (<stdexcept>)
  - ◇ length_error ··· 最大長を超えた長さのオブジェクト生成 (<stdexcept>)
  - ◇ out_of_range ··· 実引数の値が範囲外 (<stdexcept>)
- runtime_error ··· プログラム実行前に検出できない実行時エラーを表すオブジェクトのクラスで、ヘッダ<stdexcept>で定義されている。このクラスの派生クラスとして更に次のクラスが用意されている。
  - ◇ range_error ··· 内部計算で発生する範囲エラー (<stdexcept>)
  - ◇ overflow_error ··· 算術的なオーバーフローエラー (<stdexcep
  - ◇ underflow_error ··· 算術的なアンダーフローエラー (<stdexce