C++ と例外安全

@suibaka

2014 年 5 月

目次

0 序論

本記事は、「C++ における例外安全の重要性、およびそれを実現するための手法」を解説するものである. 言わずと知れた「Exceptional C++」シリーズは、C++03 における例外安全について述べられていた名著であるが、本記事は最新の C++ 規格に則った、例外安全へのより新しいアプローチを示す.

第 1 章では、例外安全とは何かについて述べる. 第 2 章では、例外安全なプログラムを組むための考え方を示す.

なお、本記事の読者は

- C++ の基本構文を理解している者
- C++11 で導入された言語機能、特にムーブセマンティクスを理解している者
- より強固で安全なプログラムを組もうと考えている者

を想定している.

1 例外安全概要

1.1 例外安全とは

例外安全とは、プログラムが予期せぬエラーを引き起こし、例外を投げた場合でも、プログラムの状態が破壊されず、かつリソースリークを起こさないことを指す.

例えば、以下のようなコードは例外安全ではない.

```
void f() {
  int* ip = new int(10);

  // 例外をなげうる処理
  some_process();

  delete ip;
}
```

関数 some_process() が例外を投げた場合、delete が呼び出される前に関数を脱出するため、動的に確保されたメモリが解放されずに残されてしまう. さらに、脱出した時点で、確保されたメモリの情報を失ってしまい、関数 f の呼び出し側でメモリリークに対応できない.

1.2　例外安全保証について

プログラムが例外安全であるということは、それが「基本的保証」「強い保証」「no-fail 保証」[*1]のいずれかを満たしていることである. 以下に、それぞれが何を意味するかをまとめた.

基本的保証
: 失敗した操作によってプログラムの状態が変更されても、リソースリークが発生しないこと. また、影響を受けたオブジェクト等が矛盾のない状態で利用可能であること (内部状態の整合性が保たれているという). ただしこの時、その状態が予測可能である必要はない.

強い保証
: 操作に失敗したとしても、操作対象のオブジェクトの状態が一切変更されない (操作直前の状態に戻る) こと. つまり、操作が終わった時、オブジェクトは操作が完全に成功した状態か、操作直前の状態のどちらかに必ずなっている.

no-fail(no-throw) 保証
: 操作によって例外が発生することはないこと.

例外中立
: ある関数内部で行われた操作によって例外が投げられた場合、その例外を関数の呼び出し元に伝播させること. すなわち、関数は例外を捕捉して特別な処理を行わない.

4 つ目の「例外中立」は保証ではないのだが、例外安全を語る上で外せない単語であるため、覚えておくとよいだろう.

2　例外安全へのアプローチ

この章では、例外安全なプログラムを書くための指針、及び手法を説明する.

2.1　try/catch はどこに書くべきか

`throw` をどこに書くか、これは明らかだ. 予期せぬ失敗が起こった時に、それを通知するために書けばよい.

では、`try/catch` はどこに書けばよいか. 少なくとも、何も考えずに例外をなげうる操作のたびに書くというのでは話にならない. `try/catch` を書くのは、大抵、以下の場合においてである.

- 投げられた例外を、別の例外に変換する場合.
- 操作の一部で例外が投げられても、それを許容出来、処理を続けたい場合.
- 投げられた例外に対して、何かしら後処理ができ、そうしたい場合.

当然だが、後処理もできないような場所で例外を `catch` しても意味が無い. その場合は、例外中立でな

[*1] これらの保証の名前は、特に厳密に決まっていないが、本記事ではこれらで統一する.

ければならない. そして、適切な処理が行える場所で初めて、例外を catch する.

```
// ライブラリ側のコード. some_process() は例外をなげうる関数
resource load_resource(int id) {
  // ...
  some_process(); // ここで例外が投げられた時、どう対処するかは
                  // ライブラリ使用者に委ねるべき. 考えられる選択肢は、
                  // 1: catch して後処理をした後、例外を rethrow あるいは変換する
                  // 2: 何もしない (例外中立)
                  // のどちらか.
  // ...
}

// ユーザー側のコード
void f() {
  try {
    auto rc = load_resource(RESOURCE_ID);
  } catch (...) {
    // ここで、失敗に対する処理を行う (ログを取る、rethrow する、etc...)
  }
}
```

2.2 catch で受ける型は参照で

catch で受ける型は参照型にすべきである. 例外的に、組み込み型やポインタであれば、参照でなくともよい[*1]. これを守ることで、コピーによって例外が投げられたり、スライシングが起こったりすることを防ぐことが出来る.

```
class my_exception : public std::exception { /* ... */ };
void g() { throw my_exception; }
void f() {
  try {
    g();
  } catch(my_exception& const ex) { // catch(my_exception ex) は良くない形
    // ...
  }
}
```

[*1] 組み込み型やポインタを受ける catch を書くことは全くといっていいほど無いので、気にしなくともよい

2.3　デストラクタで例外を投げない

これは有名なので、あえて細かく説明はしない. デストラクタが例外を投げると、どのように工夫しても例外安全なプログラムは組めなくなる. 同様に、リソースの解放を行う関数も、例外を投げるべきではない.

2.4　ムーブコンストラクタ、ムーブ代入演算子で例外を投げない

ムーブコンストラクタ、及びムーブ代入演算子は例外を投げるべきではない. これらが例外を投げる可能性があると、複数のオブジェクトを同時にムーブしたりするときに、例外安全の強い保証が満たせなくなってしまう. また、これらが例外を投げない場合、例外安全なプログラムが組みやすくなるという側面もある.

例えば、以下のようにコピー代入演算子の実装や、ローカル変数を関数外に返す時などに有用である[*2].

```
class foo {
public:
  // ...
  foo& operator=(foo&&) = default; // 例外を投げない
  foo& operator=(foo& const rhs) {
    return *this = foo(rhs); // copy & move
  }
};

some_class f() {
  some_class s;
  // ...
  return s; // 自動で move される
}
```

ここで使われている copy and move の手法はかなり有用であるから、ぜひ覚えておいてほしい. ただし注意点として、この手法を用いる場合はムーブ代入演算子を定義して置かなければならない. `= default` で定義できる場合はいいが、できない時もあるので気をつけよう.

[*2] C++03 以前では、ローカル変数を戻り値として使用する場合、コピーが発生してしまった (RVO がはたらく場合もある) が、最新の C++ では可能なら自動でムーブされるようになっている. コピーは例外をなげうるため、ムーブが行えると楽.

2.5 swap 関数で例外を投げない

関数 swap に関しても、例外を投げてはならない. なぜなら、swap 関数は、例外安全なコピー代入演算子を実装するために必要な関数であるからだ.

実は、C++03 時代の swap を用いた手法は、ムーブセマンティクスで一部代用可能である. しかしながら、C++11 の環境で書いていたとしても、C++03 時代のライブラリを使っていて、それがムーブに対応していないことも有るだろう. あるいは、そもそも C++11 が使えない環境であるかもしれない. そのような場合では、例外を投げない swap 関数が必須になってくる.

```
class widget {
  T1 t1;
  T2 t2;
public:
  widget() = default;
  widget(T1 t1_, T2 t2_) : t1(std::move(t1_)), t2(std::move(t2_))
  {}
  widget(widget const&) = default;
  widget(widget&&) = default;

  widget& operator=(widget rhs) noexcept {
    this->swap(rhs);
    return *this;
  }

  void swap(widget& other) noexcept {
    using std::swap;
    swap(t1, other.t1);
    swap(t2, other.t2);
  }
  friend void swap(widget& lhs, widget& rhs) noexcept {
    lhs.swap(rhs);
  }
};
```

ちなみに、標準で用意されている std::swap は、C++11 からはムーブで行われるようになっている.

```
template <typename T>
void swap(T& a, T& b)
```

```
  noexcept(noexcept(
    std::is_nothrow_move_constructible<T>::value &&
    std::is_nothrow_move_assignable<T>::value))
{ /* ... */ }
```

見てもらえれば分かる通り、ムーブコンストラクタとムーブ代入演算子が例外を投げないのであれば、自前で swap を定義しなくとも例外安全な swap が手に入るようになっている.

2.6　コピー代入演算子は強い保証を満たそう

代入演算子は、強い保証を満たすべきである. これは、操作が失敗したことによって、オブジェクトの内部情報が永遠に失われるのを防ぐためである.

2.7　スマートポインタを使おう

スマートポインタを使用することで delete 忘れといったミスを防ぐ事が出来る. また、例外が投げられてスコープを脱出した時も自動でメモリを開放してくれる. 使用方法も通常のポインタと何ら変わりがないので、理由がないならこちらを使うべきだ.

```
{
  std::shared_ptr<int> p = std::make_shared(10);
  int i = *p; // i == 10
  p = std::make_shared(20); // もともと指していたメモリは解放
} // スコープを抜ける時、自動でメモリを解放
```

2.8　出来る限り new を書かない

もちろん、動的確保をするなといっているわけではない. しかしながら、new を直接書くようなコードは、思わぬメモリリークを招く結果になることがある. 例えば、以下のコードを考える.

```
void f(int* p1, int* p2);

f(new int(10), new int(20)); // (1)

using std::shared_ptr;
void g(shared_ptr<int> p1, shared_ptr<int> p2);

g(shared_ptr<int>(new int(10)), shared_ptr<int>(new int(20))); // (2)
```

(1) は一見何も問題がないように見えるが、このコードは例外安全ではない. なぜなら、一方の new が成功した後、もう一方の new が失敗して例外を投げると、最初に確保されたメモリが破棄されないまま放置されるからだ.

(2) は、(1) の問題に対してスマートポインタを用いることで、問題を回避しているように思える. が、残念ながらこれも同様に例外安全でない. これは C++ の関数の引数式の評価順に起因する.

```
f(g(expr1), g(expr2));
```

上記の関数呼び出しで、expr1、expr2、g(expr1)、g(expr2) の評価順はどのようになるだろうか.

C++ では、この関数呼び出しにおいて、次の規則に従っている限り、引数式は「どのような順で評価されても良い」という事になっている.

- expr1 は、g() が呼び出される前に評価されていなければならない
- expr2 も同様.
- g() は、f() が呼び出される前に処理を完了していなければならない
- expr1 と expr2 は交互に評価されても良い.

ここで重要なのは、一番最後である. この規則によれば、(2) の場合でも、new int(10) が呼び出された後、shared_ptr のコンストラクタが呼ばれる前に new int(20) が評価される可能性がある (その逆もありうる).

この問題は、スマートポインタを作成するヘルパー関数を用いることで解決できる.

```
g(std::make_shared(10), std::make_shared(20));
```

make_shared() はそれ自体が例外安全の強い保証を満たしているから、メモリリークの心配は無い.

もちろん、以下のように書いても良い.

```
shared_ptr<int> p1(new int(10));
shared_ptr<int> p2(new int(20));
g(p1, p2);
```

が、一般的に make_shared のほうが高速であるし、わざわざ new を使う必要もない.

大切なのは、new をできるだけ用いずにスマートポインタ、及びそれに関連するヘルパー関数を使うということである.

2.9 RAII

RAII(Resource Acquisition Is Initialization) とは、リソースの確保と解放を、それぞれオブジェクトの初期化と破棄と同時に行う (コンストラクタでリソースを確保し、デストラクタで解放する) 手法である. これも、リソースリークを防ぐことにつながる. スマートポインタも一種の RAII である. 以下のコードは、RAII を用いたリソース管理の例である.

```
some_data load(std::string path);
```

```
void release(some_data& const data);

class resource {
public:
  resource(std::string path)
    : data_(load(std::move(path)))
  {}
  ~resource() {
    release(data_)
  }

private:
  some_data data_;
};

void f() {
  resource rc("data/img/01.png");
  bool success = false;
  // ...
  if(success) throw runtime_error("Error!");
} // 例外が投げられても、正しく some_data の解放が行われる
```

2.10　強い保証が必要な時は

強い保証を満たすためのアプローチを考える前に、言っておくことがある.

それは、強い例外安全を保証する場合、しばしばパフォーマンスの犠牲を強いるということである. また、大抵の場合、強い保証を満たさなくとも、基本的保証で事足りる事が多い. だから、すべての関数が強い保証を満たす必要はない.

2.10.1　関数が持つ役割を一つにする

単一の関数内に関連性のない複数の処理があると、例外安全の強い保証を満たすことができない場合が多い. そのような場合には、それぞれの処理ごとに関数を分割することが必要になってくる.

また、処理を分割することで、強い保証だけでなく、プログラムの再利用性も高めることが出来る.

2.10.2 例外をなげうる操作とそうでない操作の線引きをする

例外安全の強い保証を満たす関数を書く時には、例外を投げる操作と投げない操作を分けて考え、適切な順番で呼び出すことが必要である.

このことを、強い保証を満たすべきである関数の代表例である、コピー代入演算子で確認してみる.

```
// T1, T2 は共に例外を投げない swap メンバ関数を持っている
class widget {
public:
  widget& operator=(widget const& rhs) {
    T1 tmp1(rhs.t1); // (1)
    t1.swap(tmp1);   // no throw
    T2 tmp2(rhs.t2); // (2)
    t2.swap(tmp2);   // no throw
  }
private:
  T1 t1;
  T2 t2;
};
```

上記のコードで、例外をなげうる操作は (1) と (2) である. 上記のように、例外を投げる操作と投げない操作が不規則に並んでいると、強い保証を満たすことができない. 実際、T2 のコピーコンストラクタが例外を投げると、この関数は強い保証を満たすことができない. なぜなら、t1 に関する処理は完了していて、巻き戻すことができないからだ.

解決策は、最初に述べたとおり、例外を投げる操作と、投げない操作を分けることである. 例外をなげうる操作を完了してから、例外を投げない操作だけを用いてオブジェクトの状態を変更する.

```
T1 tmp1(rhs.t1);
T2 tmp2(rhs.t2);
//-----線引き (上:例外をなげうる 下:例外を投げない)-----//
t1.swap(tmp1);
t2.swap(tmp2);
```

この考え方は、もちろんコピー代入演算子以外でも有用である.

2.11 C スタイルの戻り値によるエラーハンドリングと例外との比較

例外機構がある C++ においても、C スタイルの戻り値によるエラーハンドリングを行うことがある. C スタイルのエラーハンドリングとは、以下のようなコードを指す.

```
error_code f() {
  if(!process1()) {
    return FAILED_PROCESS1;
  }
  if(!process2()) {
    return FAILED_PROCESS2;
  }
  return SUCESS;
}

int main() {
  error_code error = f();
  if(error == SUCESS) {
    // ...
  } else if(error = FAILED_PROCESS1) {
    // ...
  } else if(error = FAILED_PROCESS2) {
    // ...
  }
}
```

プログラマがこのようなエラーハンドリングを行う主な理由は、例外を用いたものよりも処理速度が早いとされているからだ. しかしながら、この手法を用いた場合、コードの見通しが悪くなったり、コードが長くなるという欠点もある. そのコードの一例 [7] が以下のものである.

```
// 可読性に欠け、かつ冗長である
static acl_t CreateReadOnlyForCurrentUserACL(void) {
  acl_t theACL = NULL;
  uuid_t theUUID;
  int result;
  result = mbr_uid_to_uuid(geteuid(), theUUID);
  if (result == 0) {
    theACL = acl_init(1);
    if (theACL) {
      Boolean freeACL = true;
      acl_entry_t newEntry;
      acl_permset_t newPermSet;
      result = acl_create_entry_np(&theACL, &newEntry, ACL_FIRST_ENTRY);
      if (result == 0) {
```

```
        result = acl_set_tag_type(newEntry, ACL_EXTENDED_ALLOW);
        if (result == 0) {
          result = acl_set_qualifier(newEntry, (const void *)theUUID);
          if (result == 0) {
            result = acl_get_permset(newEntry, &newPermSet);
            if (result == 0) {
              result = acl_add_perm(newPermSet, ACL_READ_DATA);
              if (result == 0) {
                result = acl_set_permset(newEntry, newPermSet);
                if (result == 0)
                  freeACL = false;
              }
            }
          }
        }
        if (freeACL) {
          acl_free(theACL);
          theACL = NULL;
        }
      }
    }
    return theACL;
}
```

この関数を、例外と簡単なラッパークラスおよび関数を用いて書き直すと、次のようになる.

```
// 例外を用いた場合
static acl_t CreateReadOnlyForCurrentUserACL() {
  ACL theACL(1);
  acl_entry_t newEntry;
  acl_create_entry_np(&theACL.get(), &newEntry, ACL_FIRST_ENTRY);

  acl_set_tag_type(newEntry, ACL_EXTENDED_ALLOW);

  uuid_t theUUID;
  mbr_uid_to_uuid(geteuid(), theUUID);
  acl_set_qualifier(newEntry, (const void *)theUUID);
  acl_permset_t newPermSet;
```

```
    acl_get_permset(newEntry, &newPermSet);

    acl_add_perm(newPermSet, ACL_READ_DATA);
    acl_set_permset(newEntry, newPermSet);

    return theACL.release();
  }
```

明らかに、例外を用いたほうが見通しもよく、なおかつすべての if 節を排除することによって、フローの単純化にも成功している. また、この関数は例外中立であるから、すべての例外を適切に呼び出し元に伝播している.

しかし、そもそも C スタイルのエラーハンドリングを行うのは速度が理由であった. ここで問題になるのが、例外機構を使うことで速度はどう変わるのかである.

実は、一概に早くなるとも遅くなるとも言えない. 確かに、C スタイルのエラーハンドリングのほうが早い場合も有るのだが、場合によっては例外を用いたほうが早い場合もある. これはコンパイラがどのように実装されているか[*3]によって変わる.

では、C スタイルをあえて選択する理由はなんだろうか. それは、実行バイナリのサイズが小さいということだ. 一般に、例外を用いたコード (というよりは、例外機構を無効にせずにコンパイルされてできたもの) は、実行バイナリのサイズが肥大化する. この場合には、例外ではなく、C スタイルのエラーハンドリングを選択することになる.

これらを総合すると、C++ におけるエラーハンドリングは、バイナリサイズが気にならないのであれば、例外を用いたほうがメリットが大きい、ということになる.

2.12　例外指定について

例外指定とは、ある関数がなげる例外を宣言時に予め明示しておくことである.

2.12.1　C++03 での例外指定

```
void f() throw(int, std::exception);
```

上記の関数 f の実装者は、投げられる例外をあらかじめ int か std::exception であると明示している. そして、「おそらく」関数 f は、それらの例外のいずれかしか投げない.

さて、ここで「おそらく」と述べたのは、例外指定されていない型が throw されることもあるからである. 例外指定は、投げられる例外を制限できる機能ではない. 指定された例外以外の例外が投げられた場合、関数 std::unexpected が呼ばれる. std::unexpected は、デフォルトでは std::terminate を呼び出す.

また、例外指定で何も指定しないことも可能である. このように指定することで、その関数がおそらく

[*3] setjmp/longjmp method と zero-cost method がある

例外を投げないことを明示できる. これを無例外指定という.

```
void g() throw();
```

さて、実は例外指定は、無例外指定以外は役に立たない. もしも、投げられる可能性のある例外を、ユーザーに明示したいのであれば、例外指定ではなく、ドキュメントに書くなどすべきである.

2.12.2 C++11 での例外指定

C++11 から、新たなキーワード noexcept が追加されている. これは、コンパイラによる最適化のヒントのために導入された機能である.

指定子としての noexcept

```
void f() noexcept;         // 無例外指定. throw() と殆ど同じ.
void g() noexcept(true);   // 上と同じ
void h() noexcept(false); // どんな例外でも投げうる
```

ここで、noexcept は、C++03 の throw(std::exception, int) のような使い方はできない. もちろん、このような例外指定は無意味であるから、使えなくても問題になることはない.

演算子としての noexcept

```
void f();
void g() noexcept;

static_assert(noexcept(f()), ""); // error
static_assert(noexcept(g()), ""); // ok
```

noexcept(未評価式) で、未評価式が例外を投げる可能性があれば false、そうでなければ true を返す. これは単に、noexcept と指定されている関数を呼び出すかどうかで判断される.

実際の使用例

```
template <typename T>
struct foo {
  void f(T t) noexcpet(std::is_nothrow_constructible<T>::value);
  void swap(foo& other) noexcept(noexcept(swap(t, other.t)));
private:
  T t;
};
```

この例では、関数 f は、型 T が例外を投げないコンストラクタをもつ場合、noexcept(true) となり、無例外指定をしたことになる. 逆に、そうでなければ、noexcept(false) となり、いかなる例外も投げう

るという指定になる.

同様に、関数 swap は、swap(t, other.t) が noexcept であれば、noexcept(true) となる.

2.12.3　noexcept VS. throw()

noexcept と throw() は多くの場面で同じ働きをするが、noexcept が優れている点がある.

1. noexcept で無例外指定をされた関数は、throw() で指定されたものより処理速度が早い. というのも、スタックを unwind する必要がなく、いつでも unwinding を中止することができるから[*4]だ. throw() では確実に unwinding が行われる.
2. noexcept では、noexcept(*some-condition*) のように条件式が書けるため、先ほどの swap のように template と組み合わせることで、より強力なプログラムが書ける.

なお、Dynamic Exception Specifications[*5]と std::unexpected() は、C++11 から deprecated になっている.

無例外指定をしたい場合は、throw() ではなく、noexcept を使うべきである.

stack unwinding についての補足

return する、スコープの終端に到達する、あるいは例外が投げられてスコープを脱出する時、そのスコープ内で生成されたオブジェクトはデストラクタが呼ばれ破棄される. スコープから脱出した時にローカルオブジェクトを破棄してデストラクタを呼ぶ、この処理のことを stack unwinding という. stack unwinding は、goto を用いた場合にも行われる[*6].

余談になるが、コンパイラの unwind を実装方法には、SjLj(setjmp/longjmp) と dwarf2 があるらしい. しかしながら、N3337 の §18.10/4 によれば、C++ では longjmp は stack unwind をしない[*7]ため、C++ では使うべきではないとされている. 以下にその文面を引用する.

> The function signature longjmp(jmp_buf jbuf, int val) has more restricted behavior in this International Standard. A setjmp/longjmp call pair has undefined behavior if replacing the setjmp and longjmp by catch and throw would invoke any non-trivial destructors for any automatic objects.

2.12.4　どの関数に無例外指定をするか

一般的に、以下の状況では、無例外指定をするべきではない.

- パフォーマンスの低下が、どうしても許容できない場合.

[*4] stack unwinding が無いのは危険な気もするが、そもそも noexcept 指定されているにもかかわらず例外を投げるのは明らかにバグである

[*5] throw() や throw(int, std::exception) などのこと

[*6] N3337 §6.6/2 と、§6.7 の example を参照

[*7] こちらの記事も参考になる - http://stackoverflow.com/questions/1376085/c-safe-to-use-longjmp-and-setjmp

- std::terminate が呼ばれてほしくない場合.
- 将来的にその関数に変更が加わる可能性がある場合.
- 処理の内容が、本当に例外を投げないか疑わしい場合.

確信を持って noexcept 指定できないなら、しないほうがいいだろう、ということだ. むやみにしてしまうと、制約のために拡張性が失われたり、予期せぬ例外でプログラムがクラッシュしてしまう.

逆に、noexcept 指定されるべき関数も存在する. ここまで読んでいればすぐわかると思うが、デストラクタ、ムーブに関わる特殊関数、そして swap である.

2.12.5 例外指定まとめ

- throw(foo, bar) の形で例外指定をしない. これらには殆ど意味が無い.
- throw() ではなく noexcept を使う.
- noexcept を指定する関数は、そうでなければならない関数、あるいは確固たる自信をもって例外を投げないと言い切れる関数に限る. リソースの解放を行うクリーンアップ関数などがその例.

2.13 別スレッドの例外を捕捉する

別スレッドの例外を捕捉するには、std::exception_ptr を使う.

```
class my_exception {};

void f(int& ret, std::exception_ptr& ep) {
  int i = 0;
  ep = nullptr;
  try {
    // ...
    if(i == -1) {
      throw my_exception();
    }
    ret = i;
  } catch(...) {
    ep = std::current_exception();
  }
}

int main() {
  int ret = 0;
  std::exception_ptr ep;
```

```
  std::thread t(f, std::ref(ret), std::ref(ep));
  t.join();

  try {
    if(ep != nullptr) {
      std::rethrow_exception(ep):
    }
  } catch(std::exception const& ex) {
    std::cerr << ex.what() << std::endl;
  } catch(my_exception const& ex) {
    std::cerr << "failed" << std::endl;
  }
}
```

別スレッドで投げられた例外オブジェクトを std::current_exception を用いて std::exception_ptr に格納する. そしてその後、メインスレッドで std::rethrow_exception を用いて、std::exception_ptr に格納された実際の例外オブジェクトを取り出す.

3　終わりに

いかがだっただろうか. 例外安全を意識したことがなかった人の中には、めんどくさいなと思った人もいることだろう. しつこいようだが、例外安全であることは非常に重要なことである. 一度例外安全なコードを書けば、それ以降そのコードはみなさんの強力な味方になってくれよう. 本記事で紹介した手法などを用いて、ぜひとも例外安全に挑んでいただきたい.

それではみなさん、よい C++ ライフを.

参考文献

[1] Tom Gargill, “Exception Handling: A False Sense of Security’.” C++ Report, Nov-Dec, 1994

[2] Herb Sutter, “Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions” Nov 28, 1999

[3] Herb Sutter, “More Exceptional C++: 40 New Engineering Puzzles, Programming Problems, and Solutions” Dec 27, 2001

[4] Herb Sutter, “Exceptional C++ Style: 40 New Engineering Puzzles, Programming Problems, and Solutions” Aug 12, 2004

[5] Doug Gregor, “Deprecating Exception Specifications” Mar 12, 2010

[6] zakkas783, “C++ の例外ハンドリングとパフォーマンス” Mar 15, 2011

[7] Jon Kalb, “Exception-Safe Coding” C++Now!2012