Programowanie Obiektowe

Marcin Kamil Bączyk

Wykład 11

12 grudnia 2019

Treść dzisiejszego wykładu

- obsługa sytuacji wyjątkowych
- osercje
  - dynamiczne
  - statyczne

Sytuacje wyjątkowe

Sytuacja wyjątkowa jest to zdarzenie, zaistniałe w trakcie wykonywania programu, które powoduje, że prawidłowe wykonanie funkcji/metody nie jest możliwe.

Sytuacje wyjątkowe

Sytuacja wyjątkowa jest to zdarzenie, zaistniałe w trakcie wykonywania programu, które powoduje, że prawidłowe wykonanie funkcji/metody nie jest możliwe.

Przykładowe sytuacje wyjątkowe:

- brak możliwości otwarcia pliku
- brak możliwości przydzielenie pamięci
- przekroczenie dopuszczalnego rozmiaru stosu

Sytuacje wyjątkowe wymagają specjalnej reakcji ze strony użytkownika / programu.

W jaki sposób użytkownik powinien reagować na sytuacje wyjątkową?

Prosty przykład

```
int funA(int x, int* err_no){
    /*...*/
    *err_no = 1;
    return x;

    /*...*/
    *err_no = 0;
    return x;
}

int funB(int x){
    int err_no;
    for(unsigned int i = 0; i < 100; ++i)
        for (unsigned int j = 0; j < 100; ++j)
            x = funA(x, &err_no);

    return x;
}
```

Jaką wartość powinna zwrócić funkcja `funB` *gdy funkcja* `funA` *zwróci informację o błędzie?*

Sytuacje wyjątkowe

W języku C++ istnieje mechanizm **zgłaszania** (ang. *throw*) **wyjątków** (ang. *exception*).
Wyjątki są zgłaszane, gdy dany fragment programu nie jest wstanie poradzić sobie z zaistniałą sytuacją wyjątkową.
Zgłoszony wyjątek może zostać **przechwycony** (ang. *catch*) i obsłużony w miejscu programu, w którym dostępne są niezbędne informacje do obsługi tego wyjątku.

W jaki sposób może zostać obsłużony wyjątek braku możliwości otwarcia pliku o wskazanej nazwie?

Wyjątek

W języku C++ **wyjątek** jest **obiektem**.

- Zgłaszany wyjątek może być obiektem dowolnego typu, w tym również typów wbudowanych.
- Klasa reprezentująca wyjątek danego typu powinna być wstanie dostarczyć jak najwięcej informacji dotyczących zaistniałej sytuacji wyjątkowej.
- Po zgłoszeniu wyjątku sterowanie jest przekazywane do najbliższej procedury obsługi wyjątku określanej na podstawie typu zgłaszanego wyjątku.

Zgłaszanie wyjątku

`throw obiekt`

Dana funkcja lub metoda może zgłosić wiele wyjątków, różnego rodzaju. Jednorazowo zgłoszony może być tylko jeden wyjątek:

```
struct Exception1 { /*...*/ };
struct Exception2 { /*...*/ };

/*...*/

void fun()
{
    /*...*/ const Exception1 e1;
    /*...*/ throw Exception1();
    /*...*/ throw Exception2();
    /*...*/ throw e1;
    /*...*/ throw 1;
    /*...*/ throw std::string("Uwaga:wyjatek");
}
```

Zgłaszanie i przechwytywanie wyjątków

Aby obiekty danej klasy mógł być zgłoszony jako wyjątek klasa ta musi mieć zdefiniowany konstruktor kopiujący oraz dostępny destruktor:

```
struct Exception3
{
    Exception3(void) = default;
    Exception3(const Exception3&) = delete;
private:
    ~Exception3(void) {};
};

/*...*/

void fun()
{
    /*...*/
    // throw Exception3(); // blad
    /*...*/
}
```

Zgłaszanie i przechwytywanie wyjątków

Istnieje możliwość rzucenia obiektem, dla którego pamięć przydzielona została na stercie, np. za pomocą operatora `new`:

```
struct Exception3
{
    Exception3(void) = default;
    Exception3(const Exception3&) = delete;
private:
    ~Exception3(void) {};
};

/*...*/

void fun()
{
    /*...*/
    throw new Exception3(); // OK, ale...
    /*...*/
}
```

Jakie problemy mogą być związane z rzuceniem obiektu poprzez jego uchwyt (wskaźnik do obiektu)?

Zgłaszanie i przechwytywanie wyjątków

Zgłoszenie wyjątku wiąże się z przerwaniem wykonywania danej funkcji i przekazanie sterowania do odpowiedniego bloku obsługi wyjątku w funkcji wywołującej.

Przechwytywanie wyjątków

```
catch(const typ_lapanego_wyjatku& e)
{
    /*...*/
}
```

Standard języka C++ zaleca zgłaszanie wyjątków przez wartość a przechwytywanie przez stałą referencję. Wykorzystanie referencji powoduje, że możliwe staje się przechwytywanie również obiektów klas pochodnych. Słowo kluczowe `const` pokazuje intencje programisty, że nie ma zamiaru zmieniać obiektu wyjątku.

Zgłaszanie i przechwytywanie wyjątków

Przechwytywanie (`catch`) wyjątku musi następować zaraz po bloku `try`.
Na podstawie typu wyjątku
`catch(const typ_lapanego_wyjatku& e)`
rozpoznawany jest blok, który ma obsłużyć dany wyjątek.

```
void fun2()
{
    try
    {
        fun();
    }
    catch (const Exception1\& e) { /*...*/ }
    catch (const Exception2\& e) { /*...*/ }
    catch ( int n) { /*...*/ }
    catch (const std::string\& e) { /*...*/ }
    catch (Exception3* e) { /*...*/ } // jak obsluzyc ten wyjatek
}
```

Zgłaszanie i przechwytywanie wyjątków

- Poszczególne bloki obsługi wyjątków są analizowane w podanej kolejności.
- Brak obsługi danego wyjątku powoduje jego propagacje w górę hierarchii programu.
- Brak obsługi wyjątku w programie przerywa jego wykonywanie.
- Wykrycie bloku obsługi danego wyjątku zaprzestaje analizy pozostałych i wejście do procedury obsługi wątku.
- Po wykonaniu procedury obsługi wyjątku sterowanie jest przenoszone do kodu występującego za ostatnią procedurą (blokiem `catch`) obsługi wyjątku.

Zgłaszanie i przechwytywanie wyjątków

```
void fun3()
{
    try
    {
        fun(); // zglasza wyjatek typu Exception2
    }
    catch (const Exception1& e) { /*...*/ }
    catch (int n) { /*...*/ }
    catch (Exception3* e) { /*...*/ }
}

void fun4(void)
{
    try
    {
        fun3();
    }
    catch (const Exception2& e) { /*...*/ } // <-- program
    // przechodzi do pierwszej sekcji w ktorej moze obsluzyc
    // dany wyjatek
    catch (const std::string& e) { /*...*/ }
}
```

Zgłaszanie i przechwytywanie wyjątków

Blok przechwytywania dany wyrażeniem `catch(...)` przechwytuje wszystkie niezłapane dotąd wyjątki.

```
void fun5(void)
{
    try
    {
        fun3();
    }
    catch (const std::string& e) { /*...*/ }
    catch (...) { /*...*/} // <-- tym razem wyjatek
    // typu Exception2 zostanie obsluzony w tym bloku
    /* ... */
}
```

Zgłaszanie i przechwytywanie wyjątków

Blok przechwytywania dany wyrażeniem `catch(...)` przechwytuje wszystkie niezłapane dotąd wyjątki.

```
void fun5(void)
{
    try
    {
        fun3();
    }
    catch (const std::string& e) { /*...*/ }
    catch (...) { /*...*/} // <-- tym razem wyjatek
    // typu Exception2 zostanie obsluzony w tym bloku
    /* ... */
}
```

Po wykonaniu procedury obsługi wyjątku sterowanie jest przenoszone do kodu występującego za ostatnią procedurą (blokiem `catch`) obsługi wyjątku.

Zgłaszanie i przechwytywanie wyjątków

- Jeżeli dany obiekt przechwytywany jest przez wartość, to w ramach pojedynczego bloku obsługi wyjątków jest zmienną lokalną i jest inicjowany przez wywołanie konstruktora kopiującego.
- Utworzony w ten sposób obiekt, usuwany jest po zakończeniu procedury jego obsługi.
- Przechwycony wyjątek po częściowym jego obsłużeniu (np. zwalnianie zaalokowanej pamięci), może zostać ponownie zgłoszony.

Zgłaszanie i przechwytywanie wyjątków

- W zależności od sposobu ponownego zgłoszenia wyjątku, następuje rzucenie nowym (po skopiowaniu) bądź tym samym obiektem:

```
catch (const Exception2& e) {
    /*...*/
    throw e; // tworzona jest kopia obiektu
    // i zglaszany jest obiekt typu Exception2
}

catch (const Exception2& e) {
    /*...*/
    throw; // jako wyjatek zglaszany jest ponownie
    // obiekt e
}
```

- Sposób ponownego zgłaszania wyjątków ma znaczenie w przypadku rozbudowanych hierarchii klas rzucanych obiektów.

Zgłaszanie i przechwytywanie wyjątków w konstruktorze

Konstruktor klasy, tak jak każda inna metoda może zgłaszać swoje własne wyjątki.

Konstruktor często odpowiedzialny jest za uzyskanie od systemu odpowiedniego bloku pamięci. Operacja ta, może zakończyć się zgłoszeniem wyjątku. Niezależnie od przyczyny należy obsłużyć sytuację wyjątkową.

Uwaga

Destruktory klas nie powinny zgłaszać wyjątków.

Zgłaszanie i przechwytywanie wyjątków w konstruktorze

```
struct exception{
  exception(const char * what) : what_(what) {}
  const char * what(void) const { return what_; }
private:
  const char* what_;
};

struct Base{
  Base(int i) { if (0 == i) throw exception("blad i == 0"); }
};

void main(void){
  try     {
    Base b(0);
  } catch(const exception& e) {
    std::cout << "wyjatek:" << e.what() << std::endl;
  } catch (...) {
    std::cout << "nieznany wyjatek" << std::endl;
  }
}
```

Zgłaszanie i przechwytywanie wyjątków w konstruktorze

```
struct Base{
  Base(int i) { if (0 == i) throw exception("blad i == 0"); }
};

struct Derived : public Base
{
  Derived(int i) : Base(i) {}
};

void main(void){
  try     {
    Derived b(0);
  } catch(const exception& e) {
    std::cout << "wyjatek:" << e.what() << std::endl;
  } catch (...) {
    std::cout << "nieznany wyjatek" << std::endl;
  }
}
```

Co w przypadku gdy klasa pochodna musi w jakiś sposób zareagować na zaistniałą sytuację wyjątkową?

Zgłaszanie i przechwytywanie wyjątków w konstruktorze

Język C++ pozwala na opakowanie w blok `try` - `catch` całego ciała metody lub funkcji.

W przypadku konstruktora danej klasy składnia jest następująca:

```
class nazwa_klasy
{
public:
    nazwa_klasy(/* argumenty konstruktora */) try
    /* : lista_inicjalizacyjna_konstruktora */
    {
        /* implementacja ciala konstruktora*/
    }
    catch( typ_przechwytywanego_wyjatku& )
    {
    }
    catch(...)
    {
    } // <-- w tym miejscu zlapany wyjatek
      // automatycznie rzucany jest ponownie. (throw;)
};
```

Zgłaszanie i przechwytywanie wyjątków w konstruktorze

W przypadku metody danej klasy lub funkcji pomijany jest blok listy inicjalizacyjnej konstruktora

```
struct S{
    void method(int i) try {
    /*...*/} // <-- cialo metody
    catch (const std::exception& /*e*/) {/*...*/}
    catch (...) {/*...*/ }
};

void fun(void) try {
/*...*/} // <-- cialo funkcji
catch (const std::exception& /*e*/) {/*...*/ }
catch (...) {/*...*/ }
```

Uwaga

W przypadku zwykłych metod oraz funkcji po dotarciu do ostatniego bloku `catch` w przeciwieństwie do konstruktora wyjątek nie jest ponownie zgłaszany.

Zgłaszanie i przechwytywanie wyjątków w konstruktorze

```
struct Base{
  Base(int i) { if (0 == i) throw exception("blad i == 0"); }
};

struct Derived : public Base
{
  Derived(int i) try : Base(i)
  {
  }
  catch (const exception& e)
  {
    std::cout << "wyjatek:" << e.what() << std::endl;
  }
  catch (const std::exception& e)
  {
    std::cout << "wyjatek:" << e.what() << std::endl;
  }
  catch (...)
  {
    std::cout << "nieznany wyjatek" << std::endl;
  }
};
```

Specyfikacja zgłaszanych wyjątków

W języku C++ **istniał** mechanizm informowania kompilatora / użytkownika o tym jakie wyjątki może zgłosić dana funkcja / metoda.

```
void fun1(void) throw();
void fun2(void) throw(typ_1, typ_2 /* ,...*/);

struct S
{
    void method1(void) throw();
    void method2(void) throw(typ_1, typ_2 /* ,...*/);
};
```

Uwaga

Obecnie mechanizm ten jest niezalecany. Część tej funkcjonalności została już usunięta z najnowszego standardu (C++17). Reszta będzie usunięta w następnej jego wersji (C++20).

Specyfikacja zgłaszanych wyjątków

Twórcy standardu języka C++ zalecają wykorzystanie słowa kluczowego `noexcept` w następujących postaciach:

- funkcja zadeklarowana jako niezgłaszająca wyjątków

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

- funkcja zadeklarowana jako niezgłaszająca wyjątków

```
void f() noexcept(true);
```

- funkcja zadeklarowana jako mogąca zgłaszać wyjątki

```
void f() noexcept(false);
```

```
struct S
{
    void method1(void) noexcept;
    void method2(void) noexcept(false);
};
```

Specyfikacja zgłaszanych wyjątków

Twórcy standardu języka C++ zalecają wykorzystanie słowa kluczowego `noexcept` w następujących postaciach:

- funkcja zadeklarowana jako niezgłaszająca wyjątków

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

- funkcja zadeklarowana jako niezgłaszająca wyjątków

```
void f() noexcept(true);
```

- funkcja zadeklarowana jako mogąca zgłaszać wyjątki

```
void f() noexcept(false);
```

```
struct S
{
    void method1(void) noexcept;
    void method2(void) noexcept(false);
};
```

std::exception

Biblioteka standardowa dostarcza wygodny interfejs w postaci klasy `std::exception`, który może być wykorzystany przez użytkownika w celu tworzenia własnych typów reprezentujących sytuacje wyjątkowe. Funkcje oraz obiekty klas biblioteki standardowej zgłaszają wyjątki dziedziczące po klasie `std::exception`.

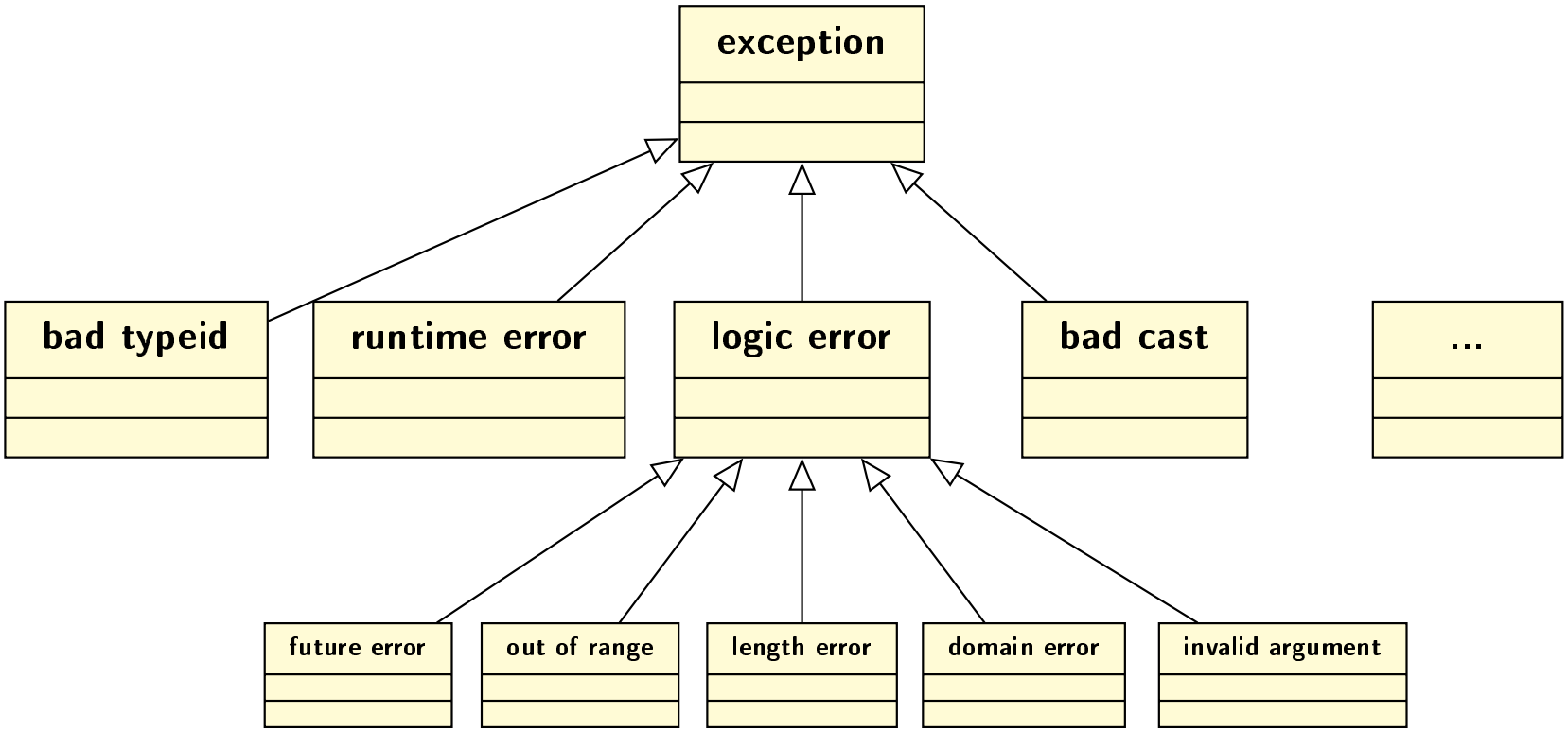
Interfejs klasy `std::exception` składa się z:

- konstruktora domyślnego oraz konstruktora kopiującego
- wirtualnego destruktora, zapewniającego poprawne niszczenie obiektów
- operatora przypisania
- metody wirtualnej zwracającej informację o przyczynie wystąpienia wyjątku.

```
virtual const char* what() const noexcept;
```

std::exception

Funkcje oraz obiekty klas biblioteki standardowej zgłaszają wyjątki dziedziczące po klasie `std::exception`.



https://en.cppreference.com/w/cpp/error/exception

W przypadku rozbudowanych hierarchii wyjątków, gdy przechwytywane są przez referencje w pierwszej kolejności należy próbować przechwycić wyjątki typu najbardziej pochodnego, przechodzą w kierunku obiektów typu podstawowego:

```
void fun6(void)
{
    try
    {
        /*...*/
    }
    catch (const std::out_of_range& /*e*/) {/*...*/}
    catch (const std::logic_error& /*e*/) {/*...*/ }
    catch (const std::exception& /*e*/) {/*...*/ }
    catch (...) {/*...*/ }
}
```

definicja klasy

```
class matrix
{
public:
    using size = std::array<unsigned int, 2>;

    matrix(unsigned int M, unsigned int N)  noexcept(false);
    matrix(const size& size) noexcept(false);
    matrix(const matrix&  rhs) noexcept(false);
    matrix(       matrix&& rhs) noexcept;

    matrix& operator=(const matrix&  rhs) noexcept(false);
    matrix& operator=(       matrix&& rhs) noexcept;
    matrix& operator+(const matrix&  rhs) noexcept(false);
    matrix& operator*(const matrix&  rhs) noexcept(false);

    double& at(unsigned int m, unsigned int n) noexcept(false);
    double& at(const std::array<unsigned int, 2>& index) noexcept(false);

    ~matrix();

private:
    size size_ = { 0,0 };
    double* data_   = nullptr;
    static const unsigned int max_size_ = static_cast<unsigned int>(1e6);
};
```

Przykład

definicja zgłaszanych wyjątków

```
class matrix_exception : public std::exception
{
public:
    matrix_exception(void)
        : matrix_exception("matrix exception") {}
    matrix_exception(const std::string& why)
        : why_(why) {}

    /*virtual */ const char* what(void) const override
    {
        return why_.c_str();
    }
private:
    std::string why_;
};
```

definicja zgłaszanych wyjątków, c.d.

```
struct matrix_max_size_exception : public matrix_exception
{
    matrix_max_size_exception(const std::string& why)
        : matrix_exception(why) {}
};

struct matrix_size_exception : public matrix_exception
{
    matrix_size_exception(const std::string& why)
        : matrix_exception(why) {}
};

struct matrix_range_exception : public matrix_exception
{
    matrix_range_exception(const std::string& why)
        : matrix_exception(why) {}
};
```

Przykład

definicja metod klasy `matrix`

```
matrix::matrix(const size& size) noexcept(false) try
    : size_(size), data_(new double[size_[0]*size_[1]])
{
    if (0 == size_[0] || 0 == size_[1])
        throw matrix_size_exception("matrix size equal to zero");

    if (size_[0] * size_[1] > max_size_)
        throw matrix_size_exception("matrix max size exceed");
}
catch (const matrix_size_exception& /*e*/)
{
    if (nullptr != data_)
        delete[] data_;
}
catch (const std::bad_alloc& /*e*/) { /*...*/ }
catch ( ... ) {/*...*/ }
```

Przykład

definicja metod klasy matrix

```
matrix & matrix::operator+(const matrix & rhs) noexcept(false)
{
    if (size_ != rhs.size_)
        throw matrix_mismatch_exception("matrix size mismatch");

    matrix result(*this);
    /*...*/
    return result;
}

matrix & matrix::operator*(const matrix & rhs) noexcept(false)
{
    if(size_[1] != rhs.size_[0])
        throw matrix_mismatch_exception("matrix size mismatch");

    matrix result(size_[0], rhs.size_[1]);
    /*...*/
    return result;
}
```

Przykład

definicja metod klasy matrix

```
double & matrix::at(unsigned int m, unsigned int n)
    noexcept(false)
{
    return at(size{ m,n });
}

double & matrix::at(const std::array<unsigned int, 2>& index)
    noexcept(false)
{
    if (index[0] >= size_[0] || index[1] >= size_[1])
        throw matrix_range_exception("matrix size exceed");

    return data_[index[0] * size_[1] + index[1]];
}
```

Przykład

```
int main(void){
    try{
        matrix m1(1e6, 1e6);   // except
        matrix m2(0, 10);      // except
        matrix m3(1024, 1024);// except
        matrix m4(100, 100);   // noexcept
        m4.at(100, 100) = 1;   // except
        matrix m5(1, 100);     // noexcept
        auto m6 = m4 + m5;     // except
        auto m7 = m4 * m5;     // except
        auto m8 = m5 * m4;     // noexcept
    }
    //catch (const matrix_size_exception& /*e*/) { /*...*/ }
    //catch (const matrix_mismatch_exception& /*e*/) { /*...*/ }
    //catch (const matrix_range_exception& /*e*/) { /*...*/ }
    //catch (const matrix_exception& /*e*/) { /*...*/ }
    catch (const std::exception& e) {
        std::cout << "exception thrown, reason: " << e.what();
    }
    catch (...) {/*...*/ }
    return 0;
}
```

Podsumowanie

Najczęściej popełniane błędy przy obsłudze sytuacji wyjątkowych:

- Przedkładanie mechanizmu zwracania kodu błędu nad zgłaszanie i przechwytywanie wyjątków.
- Niezrozumienie procesu odwikłowywania stosu (ang. *stack unwinding*.
- Wykorzystywanie wyjątków w celu kontroli przepływu sterowania.
- Niezgłaszanie wyjątków w konstruktorze, gdy tworzenie obiektu się nie powiedzie.
- Zgłaszanie wyjątków w destruktorze lub w przeciążonych operatorach `delete` lub `delete[]`.
- Zgłaszanie wyjątków nie przez wartość.
- Nieprzechwytywanie wyjątków przez referencję bądź stałą referencję.

Podsumowanie

Najczęściej popełniane błędy przy obsłudze sytuacji wyjątkowych:

- Specyfikowanie typów wyjątków zgłaszanych w funkcji.
- Nieuważane stosowanie dyrektywy `noexcept`
- Mieszanie mechanizmu obsługi wyjątków z mechanizmem zwracania kodów błędu (Umieszczanie kodu błędu wewnątrz obiektu wyjątku).
- Niedziedziczenie po wspólnej klasie bazowej dla wszystkich wyjątków w systemie (np. `std::exception`).
- Zgłaszanie wyjątków w konstruktorze klasy wyjątku.
- Niezrozumienie różnicy pomiędzy wyrażeniami `throw` oraz `throw e`
- Przepuszczanie błędów krytycznych.

https://www.acodersjourney.com/top-15-c-exception-handling-mistakes-avoid/

Dziękuję za uwagę