Bezpieczeństwo wyjątków w C++: OpenGL

Krzysztof Czaiński

Instytut Informatyki Wydziału Elektroniki i Technik Informacyjnych,
Politechnika Warszawska
K.Czainski@stud.elka.pw.edu.pl

Streszczenie Artykuł przedstawia różne sposoby obsługi błędów, aby pokazać, dlaczego potrzebny jest mechanizm wyjątków. Przybliżone zostaje pojęcie bezpieczeństwa wyjątków oraz przedstawione zostają techniki zapewnienia tego bezpieczeństwa - w kontekście *OpenGL*. Opisane są także kryteria zapewnionego bezpieczeństwa wyjątków - kryterium Dave'a Abrahamsa oraz szczegółowa klasyfikacja.

Key words: wyjątek, bezpieczeństwo wyjątków, C++, Dave Abrahams, OpenGL, RAII

1 Wstęp

Wyjątki służą do obsługi błędów. Jednak czym jest bezpieczeństwo wyjątków? Wielu programistów sądzi, że polega ono na tym, gdzie wstawić odpowiednie bloki `try` i `catch`.

W tym artykule postaram się przybliżyć pojęcie bezpieczeństwa wyjątków oraz zademonstrować sposób zapewnienia bezpieczeństwa w programach korzystających z *OpenGL*.

2 C++ i wyjątki

2.1 Obsługa błędów

Obsługa błędów to problem, a może raczej kłopot, z którym programista powinien sobie poradzić. Ten kłopot polega przede wszystkim na trudności w komunikacji [8]. Z jednej strony jest autor klasy lub biblioteki. On może wykryć błąd, ale nie posiada wiedzy, jak należy na taki błąd zareagować. Nie wie przecież, w jaki sposób i do czego jego klasa jest w danym przypadku używana. Z drugiej strony jest użytkownik owej klasy lub biblioteki (nie mylić z użytkownikiem aplikacji). Użytkownik ten jest programistą, który w aplikacji używa omawianej klasy lub biblioteki. W razie wystąpienia błędu on wie, jak na taki błąd należy zareagować, ale nie umie tego błędu wykryć. Zatem problem w obsłudze błędów polega na tym, aby autor przekazał informacje użytkownikowi.

Jako przykład posłużą:

- klasa `std::vector<int>`
- operacja `int& at(size_t i)`, zwracająca referencję do `i`-tego elementu wektora

Sytuacją wyjątkową (czyli taką, że należy sygnalizować błąd) jest odwołanie do nieistniejącego elementu wektora. Niech dla przykładu w wektorze `v` znajduje się 5 elementów; niech użytkownik wykona próbę odwołania się do elementu na pozycji 10.

```
{
  std::vector<int> v;
  for (size_t i = 0; i < 5; ++i)
    v.push_back(2*i+3);
  v.at(10) = 1; // sytuacja wyjatkowa
}
```

W jaki sposób można taką sytuację wyjątkową potraktować? Prócz zastosowania mechanizmu wyjątków, możliwości jest kilka:

1. ignorowanie
2. zakończenie działania programu
3. wyświetlenie komunikatu dla użytkownika aplikacji
4. kod powrotu
5. zmienna globalna
6. specjalny stan obiektu

Zastosowanie jednego z powyższych sposobów ma bardziej lub mniej poważne wady.

Najmniej pożądaną konsekwencją wybranego sposobu traktowania błędów jest *nieprawidłowe działanie aplikacji*. Dotyczy ona pierwszych trzech sposobów. Ignorowanie błędów prowadzi zwykle do tego, że chwilę później pojawia się sympatyczny komunikat „Program wykonał nieprawidłową operację i zostanie zamknięty". Niewiele się zmienia, gdy program zakończy swoje działanie sam. Sposób (3) spowoduje, że użytkownik aplikacji zobaczy komunikat, który będzie dla niego zupełnie niezrozumiały.

Wadą sposobu (4) jest *rezerwacja wartości zwracanej*. W przytoczonym powyżej przykładzie, metoda `at` musiałaby zwracać informację o powodzeniu, np. typu `bool`. W konsekwencji musiałaby przyjmować jako argument np. referencję do wskaźnika, który w przypadku powodzenia byłby ustawiany, aby pokazywał na żądany element wektora. Jest to komplikacja zupełnie niepożądna.

Powyższą wadę można zniwelować, stosując zmienną globalną (5) jako miejsce informacji o błędzie. Jednak stosowanie zmiennych globalnych samo w sobie prowadzi do wielu problemów, zwłaszcza w kontekście współbieżności aplikacji, co w dzisiejszych czasach jest powszechne.

W związku z powyższym, można zastosować sposób (6) - zaprojektować specjalny błędny stan obiektu. To z kolei niepotrzebnie komplikuje projekt klasy i w konsekwencji zwiększa prawdopodobieństwo pomyłki programisty.

Jednym z aspektów problemu komunikacyjnego autor-użytkownik jest to, czy wiadomość o błędzie dotrze do użytkownika. Kolejną wadą - dotyczącą sposobów (4), (5) i (6) traktowania błędów - jest brak wymuszenia na użytkowniku biblioteki obsługi błędu. Przy każdym użyciu metody `at` w naszym przykładzie użytkownik powinien sprawdzać czy błąd nie wystąpił. Jednak żaden mechanizm go do tego nie zmusza, więc istnieje szansa, że błąd zostanie *niechcący* zignorowany.

Nawet jeśli użytkownik będzie na tyle solidny, że nie zapomni obsłużyć wszystkich błędów, to pozostanie kolejna wada: Kod obsługi sytuacji wyjątkowych będzie pomieszany z pozostałym kodem programu. W konsekwencji program będzie mniej czytelny i bardziej skomplikowany. Taka sytuacja oznacza większe prawdopodobieństwo pomyłki programisty i w konsekwencji błędnie działającą aplikację.

2.2 Mechanizm wyjątków

Mechanizm wyjątków został wymyślony do obsługi błędów, pozbawionej wymienionych wad. Jednak stosowanie mechanizmu wyjątków pociąga za sobą dalsze konsekwencje.

Często w programie istnieje potrzeba przydzielenia i zwolnienia zasobów. Dla ustalenia uwagi, niech to będzie pamięć dynamiczna.

```
{
    A* a = new A;
    // (1) Kod, ktory uzywa a i moze rzucac wyjatek
    delete a;
}
```

Jeśli w miejscu oznaczonym (1) zostanie rzucony wyjątek, pamięć nie zostanie zwolniona.

Oto możliwe rozwiązanie powyższego problemu:

```
{
    A* a = NULL;
    try{a = new A;
    // (1) Kod, ktory uzywa a i moze rzucac wyjatek
        delete a;
    }
    catch ( ... )
    { delete a; throw; }
}
```

Ewentualny wyjątek jest łapany, zasób jest zwalniany, a następnie wyjątek jest rzucany dalej. To rozwiązanie posiada szereg wad. Po pierwsze kod częściowej obsługi błędu jest wymieszany z innym kodem, pociągając za sobą omówione wady takiej sytuacji. Prócz tego, bloki `try` i `catch` mogą wprowadzać nawet poważne spowolnienia.

2.3 Wzorzec projektowy *RAII*

Poprawnym i powszechnie stosowanym rozwiązaniem powyższego problemu jest użycie wzorca projektowego *RAII* (ang. *Resource Acquisition Is Initialization - zdobywanie zasobów jest inicjalizacją*). Zgodnie z tą koncepcją, konstrukcja obiektu jest automatycznie połączona ze zdobyciem zasobu, zaś usunięcie obiektu automatycznie (za pomocą destruktora) powoduje zwolnienie zasobu. Gdy taki obiekt jest zmienną lokalną, jego „czas życia" jest automatycznie kontrolowany przez kompilator. I gdy na przykład następuje wyjście z funkcji, w której obiekt został stworzony, jest on automatycznie niszczony, co powoduje zwolnienie zasobu. Wyjściem z funkcji jest zarówno naturalne jej zakończenie, jak i wyjście przedwczesne (instrukcją `return`), a także rzucenie wyjątku - we wszystkich tych przypadkach zasób zostanie automatycznie zwolniony.

W przypadku zarządzania dynamiczną pamięcią wzorzec *RAII* jest uszczegółowiony przez wzorzec *sprytnego wskaźnika* (ang. *smart pointer*). Oto przykładowa implementacja:

```
template< typename X >
class AutoPtr : boost::noncopyable
{
public:

    explicit AutoPtr( X* p = NULL ) : p_( p ) {}

    ~AutoPtr() { delete p_; }

    X& operator*() const { return *p_; }
    X* operator->() const { return p_; }

private:

    X* p_;
};
```

Prosty sprytny wskaźnik, zaprojektowany na wzór `std::auto_ptr` [9]. Pamięć jest automatycznie zwalniana, a klasa naśladuje zachowanie zwykłego wskaźnika. Kopiowanie tego typu obiektów mogłoby prowadzić do niepożądanych efektów, więc zostało zabronione.

```
{
    AutoPtr<A> a( new A);
```

```
    // (1) Kod, ktory uzywa a i moze rzucac wyjatek
}
```

Pamięć zostanie zwolniona, niezależnie czy w miejscu (1) zostanie rzucony wyjątek lub nastąpi przedwczesne wyjście, czy nie. Warto zauważyć, że powyższy kod jest krótszy, przez co prostszy, od wersji pierwotnej.

Stosowanie *RAII* nie tylko umożliwia bezpieczne rzucanie wyjątków, ale także upraszcza kod programu. Zasoby są zwalniane automatycznie, więc nie trzeba na końcu funkcji pisać kodu zwalniającego zasoby. Fakt ten zapobiega także nie zwolnieniu zasobów przez pomyłkę.

2.4 Czym jest bezpieczeństwo wyjątków?

Jak widać, bezpieczeństwo wyjątków jest czymś znacznie więcej, niż „gdzie wstawić bloki `try`/`catch`". Można nawet zaryzykować stwierdzenie, że polega na mądrym unikaniu takich bloków. [4]

3 OpenGL i wyjątki

3.1 Czym jest OpenGL?

„*OpenGL* (ang. *Open Graphics Library*) - specyfikacja uniwersalnego API do generowania grafiki. Zestaw funkcji składa się z 250 podstawowych wywołań, umożliwiających budowanie złożonych trójwymiarowych scen z podstawowych figur geometrycznych." [10] Funkcje *OpenGL* są przygotowane dla języka `C`, a co za tym idzie, `C++` je odziedziczył. Nie rzucają one wyjątków, ale i nie zapewniają bezpieczeństwa, gdy ich używać wraz z kodem, który może rzucać wyjątki.

3.2 `glBegin` i `glEnd`

Jako przykład posłuży typowa prosta funkcja rysująca punkty za pomocą *OpenGL*.

```
void render()
{
    double v[3];
    glBegin( GL_POINTS );
    for ( int i = 0 ; i < n ; ++i )
    {
        getVertex( i, v ); // rzuca?
        glVertex3dv( v );
    }
    glEnd();
}
```

Najpierw wywołanie `glBegin` ustawia *OpenGL* w stan rysowania punktów. Następnie w pętli jest wywoływana funkcja użytkownika `getVertex`, która ma za zadanie obliczyć współrzędne kolejnego punktu, który jest następnie wyświetlany za pomocą funkcji *OpenGL* `glVertex3dv`. Na koniec wywołanie `glEnd` przywraca *OpenGL*-owi pierwotny stan. `getVertex` może rzucić wyjątek.

Gdy w powyższym przykładzie funkcja `getVertex` rzuci wyjątek, `glEnd` nie zostanie wywołane i *OpenGL* pozostanie w nieprawidłowym stanie. Analogicznie jak w przykładzie z pamięcią dynamiczną, z pomocą przyjdzie *RAII*.

```
struct AutoBegin : boost::noncopyable
{
    explicit AutoBegin( GLenum mode )
    {
        glBegin( mode );
    }

    ~AutoBegin()
    {
        glEnd();
    }
};
```

RAII: konstruktor woła `glBegin`, a destruktor `glEnd`.

Zastosowanie prezentowanej klasy uodparnia przykładowy kod korzystający z *OpenGL* na wyjątki rzucane przez użytkownika. Poza tym podobnie, jak w przypadku zarządzania pamięcią dynamiczną, uniemożliwia zapomnienia wywołania `glEnd` oraz skraca zapis o jedną linijkę.

```
void render()
{
    double v[3];
    AutoBegin begin( GL_POINTS ); // (1)
    for ( int i = 0 ; i < n ; ++i )
    {
        getVertex( i, v ); // rzuca? OK.
        glVertex3dv( v );
    }
}
```

Przedstawiony wcześniej przykład, korzystający z AutoBegin.

Warto zauważyć, że w linijce oznaczonej (1) dość łatwo przez pomyłkę pominąć nazwę zmiennej:

```
    AutoBegin( GL_POINTS ); // (1)
```

Ten błąd nie zostanie nawet zasygnalizowany ostrzeżeniem kompilatora, a skutki będą zaskakujące. Spowoduje to utworzenie obiektu tymczasowego, którego czas życia wynosi 1 linijkę. `glEnd` zostanie wywołane tuż po `glBegin`, przed wejściem do pętli.

3.3 `glRenderMode`

Aby zapewnić bezpieczeństwo wyjątków, wzorzec *RAII* należy zastosować przy używaniu wielu funkcji *OpenGL*. Na przykład, gdy używamy funkcji `glRenderMode`, służącej do zmiany trybu renderowania, możemy chcieć zagwarantować, aby po wyjściu (normalnym lub wyjątkowym) tryb renderowania powrócił do domyślnego trybu `GL_RENDER`. Oto przykład:

```
void select()
{
    glRenderMode( GL_SELECT );
    // (1) kod, ktory moze rzucac wyjatek
    int hits = glRenderMode( GL_RENDER );
    //...
}
```

Gdy w miejscu (1) zostanie rzucony wyjątek, *OpenGL* pozostanie w trybie renderowanie `GL_SELECT`, zamiast powrócić do trybu `GL_RENDER`.

Rozwiązanie jest analogiczne, aczkolwiek trochę bardziej skomplikowane, dlatego warto je zaprezentować.

```
class AutoRenderMode : boost::noncopyable
{
public:

    AutoRenderMode( GLenum mode, GLenum restoreMode )
        : restoreMode_(restoreMode), restored_(false)
    {
        glRenderMode( mode );
    }

    ~AutoRenderMode()
    {
        restoreRenderMode();
    }

    GLint restoreRenderMode()
    {
        GLint result = 0;
        if ( ! restored_ )
        {
```

```
            restored_ = true;
            result = glRenderMode( restoreMode_ );
        }
        return result;
    }

private:

    GLenum restoreMode_;
    bool restored_;
};
```

Funkcja `glRenderMode` zwraca informację, której program używa, więc potrzebna jest dodatkowa metoda `restoreRenderMode`, zwracająca tę informację.

```
void select()
{
    AutoRenderMode x( GL_SELECT, GL_RENDER );
    // (1) kod, ktory moze rzucac wyjatek
    int hits = x.restoreRenderMode();
    //...
}
```

Teraz, gdy w miejscu (1) zostanie rzucony wyjątek, *OpenGL* powróci automatycznie do trybu `GL_RENDER`.

4 Bezpieczeństwo wyjątków

4.1 Gwarancje bezpieczeństwa wyjątków wg Dave'a Abrahamsa

Do określenia bezpieczeństwa wyjątków potrzebne jest kryterium. Najbardziej popularną klasyfikację zdefiniował Dave Abrahams [1]. Określił on trzy rodzaje gwarancji:

- *Podstawowa* - gwarantuje brak wycieku zasobów, a wszystkie obiekty pozostają w stanie używalnym, ale nie koniecznie przewidywalnym.
- *Silna* - gdy wystąpi wyjątek podczas operacji, stan całego programu powraca do takiego jak przed wywołaniem operacji. Oznacza to, iż zarówno obiekt powraca do pierwotnego stanu, jak i wszelkie skutki uboczne są cofane.
- *Nothrow* - nigdy nie rzuca wyjątku. Oznacza to gwarancję, że operacja zawsze się powiedzie. Warto zauważyć, że niektóre operacje (np. *destruktory*, `operator delete`) muszą zapewniać właśnie tę najsilniejszą gwarancję. W przeciwnym razie pisanie poprawnych przewidywalnych programów nie byłoby możliwe. Bez operacji gwarantujących *nothrow* nie dałoby się także zapewnić innym operacjom pozostałych gwarancji.

4.2 Szczegółowa klasyfikacja bezpieczeństwa wyjątków

Czy pisać specyfikacje `throw()`? To pytanie zadawało sobie wielu ekspertów. Oprę się na wnioskach Herba Suttera [6]:

- *Moral #1: Never write an exception specification.*
- *Moral #2: Except possibly an empty one, but if I were you I'd avoid even that.*

Wniosek jest taki, że nie należy pisać specyfikacji `throw()`... Ale czy to oznacza, że należy ignorować, co funkcje rzucają? Lepiej nie. Sensowne wydaje się pisanie tego w komentarzu. Oto informacje o ewentualnie rzucanych wyjątkach oraz o ich bezpieczeństwie, jakie warto w komentarzu zamieścić:

- Co funkcja może rzucać i w jakich okolicznościach: `/** @throws co kiedy */`.
- Jak się funkcja zachowa w razie wyjątku, czyli bezpieczeństwo wyjątków: `/** @safety - - - */`. Zamiast każdego z '`-`' może wystąpić odpowiednia mała lub wielka litera - szczegółowo wyjaśniona poniżej:

Ponownie powołam się na Herba Suttera, który trafnie opisał tę sprawę [2]. Niech opis bezpieczeństwa w razie wyjątku składa się z 3 części, którym odpowiadają '`- - -`':

1. *Atomicity* - niepodzielność. Trzy możliwości:
   - '`-`': brak gwarancji. W razie rzuconego wyjątku częściowe skutki rozpoczętych operacji mogą pozostać.
   - '`a`': niepodzielność - czyli funkcja zostanie wykonana w całości, albo wcale. Oznacza to, że w razie wystąpienia wyjątku skutki wszystkich rozpoczętych operacji zostaną cofnięte do stanu sprzed wywołania funkcji.
   - '`A`': na pewno całość - czyli gwarancja powodzenia operacji. Oznacza to, że wyjątek nie zostanie rzucony pod żadnym pozorem.
2. *Consistency* - spójność. Trzy możliwości:
   - '`-`': brak gwarancji spójności, co oznacza, że operacja może pozostawić system w niepoprawnym stanie, może nie zwolnić zasobów itp.
   - '`c`': tylko w razie rzuconego wyjątku istnieje gwarancja, że system pozostanie w stanie spójnym.
   - '`C`': niezależnie od powodzenia lub nie, system pozostaje w stanie spójnym. Z punktu widzenia programisty, wszystkie funkcje powinny zapewniać ten rodzaj gwarancji, gdyż w przeciwnym razie nie można by ich bezpiecznie wywołać. Każda funkcja wywołana w programie musi po sobie pozostawiać program w stanie spójnym, niezależnie od tego czy wystąpił wyjątek, czy nie.
3. *Isolation* - brak skutków ubocznych. Trzy możliwości:
   - '`-`': skutki uboczne występują niezależnie od powodzenia operacji.
   - '`i`': skutki uboczne występują tylko w przypadku powodzenia operacji.
   - '`I`': Skutki uboczne nigdy nie występują.

Sutter pisze jeszcze o *Durability* [2] - jak w modelu baz danych, ale nie będę tego analizować. W programach wykorzystujących *OpenGL* rzadko potrzeba pilnować, czy wynik działania operacji przetrwa w razie takich kataklizmów jak upadek systemu...

Poniżej przedstawiam przykłady specyfikacji bezpieczeństwa wyjątków dla funkcji, zgodne z prezentowaną konwencją:

- `/** @safety AC- */` oznacza, że operacja zawsze się powiedzie (ekwiwalent `throw()`); może emitować skutki uboczne.
- `/** @safety -CI */` oznacza, że w razie niepowodzenia, obiekty pozostaną w stanie spójnym (`C`), ale nie wiadomo jakim; niezależnie od wyjątków, nie wystąpią żadne skutki uboczne.
- `/** @safety aCi */` oznacza silną gwarancję Dave'a Abrahamsa, czyli w razie niepowodzenia system będzie w takim stanie, jakby wywołanie funkcji nie nastąpiło; dotyczy to także skutków ubocznych.

5 Podsumowanie

Bezpieczeństwo wyjątków można zapewnić małym kosztem, jeśli zostanie ono uwzględnione już na etapie projektowania. Warto je zagwarantować, gdyż pisanie programów, uwzględniając bezpieczeństwo wyjątków, ma istotne zalety.

Pierwszą zaletą jest to, że użytkownik klasy może używać wyjątków. Gdyby klasa nie zapewniała bezpieczeństwa wyjątków, jej użytkownik musiałby zrezygnować z ich stosowania i obsługiwać błędy na jeden z omówionych wcześniej sposobów, dziedzicząc odpowiednie wady.

Po wtóre, wyjątki się zdarzają nawet gdy sam użytkownik zrezygnuje z ich stosowania. Np. gdy zabraknie pamięci, automatycznie generowany jest wyjątek. W takim przypadku program mógłby bezpiecznie kontynuować pracę tylko jeśli jest napisany uwzględniając bezpieczeństwo wyjątków. Warto zauważyć, że w języku *Java* bardzo trudno byłoby zapewnić bezpieczeństwo wyjątków i np. *Eclipse* po napotkaniu wyjątku *OutOfMemoryException* wyświetla komunikat, że należy zapisać zmiany i zamknąć aplikację, ponieważ dalsza praca nie jest bezpieczna. Jest to związane z faktem, że nie ma zapewnionego bezpieczeństwa wyjątków.

Wreszcie zapewnianie bezpieczeństwa wyjątków ma także skutek uboczny: o zwolnieniu zasobów trudno zapomnieć. Dzięki temu, że korzystamy z mechanizmu automatycznego zwalniania zasobów, stosując wzorzec *RAII*, unikamy niezwolnienia zasobów przez pomyłkę.

Z powyższych przyczyn warto zapewnić bezpieczeństwo wyjątków także programów korzystających z *OpenGL*. Można to zrobić małym kosztem, a długofalowe zyski z uniknięcia błędów i kłopotów znacznie przewyższają koszty.

Literatura

1. Abrahams, D.: Exception-Safety in Generic Components. In: Generic Programming: Proceedings of a Dagstuhl Seminar, M. Jazayeri, R. Loos, and D. Musser, eds. Springer Verlag (1999).
2. Sutter, H.: Guru of the Week: #61 CHALLENGE EDITION: ACID Programming. (1999) http://www.gotw.ca/gotw/061.htm
3. Sutter, H.: Exceptional C++. Addison Wesley (1999)
4. Sutter, H.: Guru of the Week: #65 Try and Catch Me. (2000) http://www.gotw.ca/gotw/065.htm
5. Sutter, H.: More Exceptional C++. Addison Wesley (2001)
6. Sutter, H.: A Pragmatic Look at Exception Specifications. C/C++ Users Journal, 20(7) (2002)
7. Sutter, H.: Exceptional C++ Style. Addison Wesley (2004)
8. Nowak, R.: Śrenio zaawansowane programowanie w C++ (ZPR): Wykład 4 - obsługa błędów, sprytne wskaźniki (2007)
9. std::auto_ptr reference, http://www.cppreference.com/cppmisc/auto_ptr.html
10. OpenGL, http://pl.wikipedia.org/wiki/OpenGL
11. C++ Boost, http://www.boost.org/