Zaawansowane CPP/Wykład 15: Wyjątkowo odporny kod

From Studia Informatyczne

< Zaawansowane CPP

Spis treści

Wstęp

W poprzednim wykładzie opisałem mechanizm obsługi błędów za pomocą wyjątków. Jest to bardzo silny mechanizm: rzucony wyjątek powoduje natychmiastowe przekazanie sterowania do najbliższej klauzuli `catch`, niejako "tnąc" w poprzek dowolnie głęboko zagnieżdżone funkcje. To oczywiście jest jedną z jego podstawowych zalet, ale musimy podchodzić do tej własności bardzo ostrożnie.

W tym wykładzie zwrócę uwagę na kilka niebezpieczeństw wynikających z obsługi wyjątków i na sposoby zapobiegania im.

Wyjątkowe niebezpieczeństwa

W zasadzie korzystanie z wyjątków jest proste: funkcja, która stwierdza wystąpienie błędu, a nie umie go sama obsłużyć, przekazuje odpowiedzialność swoim przełożonym, rzucając wyjątek. Jej przełożeni mogą zrobić to samo (wystarczy, że nie przechwycą wyjątku). Zakładamy jednak, że gdzieś w tej hierarchii wyjątek zostanie złapany przez kogoś, kto wie jak go obsłużyć. W praktyce sprawa może być bardziej skomplikowana. Rzucony wyjątek powoduje natychmiastowe przerwanie nie tylko funkcji, która go rzuciła, ale również wszystkich funkcji, przez które "przelatuje". Jeśli te funkcje nie są na to przygotowane, to wyjątek może narobić dodatkowych szkód. Typowy przykład to niezwolnione zasoby:

```
void f() {
przydziel_zasob();
g(); /*może rzucić wyjątek*/
zwolnij_zasob();
}
```

Rzucenie wyjątku z `g()` spowoduje wyciek zasobu (zwykle pamięci). Taki przykład był już rozważany w Wykładzie 10 Podane tam rozwiązanie to technika "przydział zasobu jest inicjalizacją", czyli oddelegowanie zarządzania zasobem do osobnej klasy, której konstruktor przydziela zasób, a destruktor zwalnia:

```
void f() {
Zasob x;
```

```
g(); /*może rzucić wyjątek*/
} /* niejawnie wywoływany  destruktor x. Zasob() */
```

Wtedy podczas zwijania stosu zasób zostanie zwolniony automatycznie. Proszę zauważyć jednak, że jeśli nie przechwycimy wyjątku, to zasób może dalej pozostać niezwolniony. Rozwiązaniem może być kod:

```
void f() {
przydziel_zasob();
try {
g(); /*może rzucić wyjątek*/
}
cath(...) {zwolnij_zasob();throw;}

zwolnij_zasob();
}
```

Po zwolnieniu zasobu rzucamy (podrzucamy?) ponownie ten sam wyjątek. W ten sposób funkcja `f()` staje się "przeźroczysta dla wyjątków" (exception-neutral).

Konstruktory

Szczególnym przypadkiem mogącym prowadzić do wycieku pamięci są wyjątki rzucane z konstruktora. Rozważmy następujący kod:

```
struct BigRsource {
char c[10000000];
};
struct BadBoy {
 BadBoy() {throw 0;};
};

struct X {
  BigResource *p1;
  BadBoy      p2;
 X: p1(new BigResource) {}

 X() {
  delete p1;
  }
}
```

Na pierwszy rzut oka jest to pierwszorzędny przykład programowania obiektowego: pamięć jest przydzielana w konstruktorze i zwalniana w destruktorze, nie ma więc możliwości wycieku. Prześledźmy jednak, co się stanie, gdy napiszemy:

```
try {
X x;
} catch(...) {};
```

Konstruktor najpierw przydzieli pamięć dla wskaźnika `p1`. Załóżmy, że ta alokacja się powiedzie. Następnie zostanie wywołany konstruktor `BadBoy`, który rzuci wyjątek. Wyjątek nie zostanie złapany w konstruktorze `X`, więc sterowanie zostane przekazane do klauzuli `catch`. Nastąpi zwinięcie stosu, ale destruktor obiektu `x` **nie** zostanie wywołany! Dzieje się tak dlatego, że w C++ destruktory nie są wołane dla obiektów, których konstrukcja się nie powiodła. W taki sposób tracimy 10MB. Możliwe rozwiązania są podobne jak w poprzednim wypadku: korzystamy z `auto_ptr`:

```
struct X {
 std::auto_ptr<BigResource> p1;
 BadBoy      p2;
 X: p1(new  BigResource) {}
```

```
 X() {
 delete p1;
 }
};
```

lub sami łapiemy wyjątek:

```
struct X {
 BigResource *p1;
 BadBoy      p2;
 X: try {p1(new BigResource) {}}
 catch(...){delete p1;};

 ~X() {
 delete p1;
 }
}
```

Proszę zwrócić uwagę na blok `try`, który otacza cały konstruktor łącznie z listą inicjalizatorów.

The bad, the good and the ugly

Jeżeli wyjątek został rzucony przez metodę jakiegoś obiektu, to dla dalszego działania programu ważne jest, w jakim stanie go pozostawił. Wyróżnimy trzy możliwości:

The bad. Obiekt jest w stanie niekonsystentnym, nie są zachowane niezmienniki jego typu, być może nastąpił wyciek zasobów. Nieokreślone jest zachowanie wywoływanych metod, w szczególności może nie powieść się destrukcja obiektu.

The ugly. Obiekt jest w stanie konsystentnym, ale niezdefiniowanym.

The good. Obiekt pozostaje w stanie, w jakim był przed rzuceniem wyjątku. Jest to semantyka transakcji: commit--rollback.

Ewidentnie najbardziej pożądanym zachowaniem jest stan ostatni. Nie zawsze da się jednak zapewnić takie zachowanie bez ponoszenia dużych kosztów. Wtedy możemy zadowolić się stanem drugim. Stan pierwszy to oczywista katastrofa.

Przykład: stos

Rozważmy stos z dynamiczną obsługą pamięci. Przykład takiego stosu był podany w Wykładzie 7. Żeby nie wprowadzać komplikacji, nie będziemy tu korzystać z klas wytycznych:

```
template <class T,size_t N = 10> class Stack {
  size_t nelems;
  size_t top;
  T* v;

public:
  bool is_empty() const;
  void push(const T&);
  T pop();

  Stack(size_t n = N);
   Stack();
  Stack(const Stack&);
  Stack& operator=(const Stack&);
};
```

W powyższym konstruktorze może nie powieść się tylko operacja tworzenia tablicy `v`. Ale wtedy, zgodnie z tym co już omawialiśmy w poprzednim wykładzie, wyrażenie `new` samo po sobie posprząta. Nie musimy się martwić stanem pozostawionego obiektu, bo jeśli konstrukcja się nie powiedzie, to obiektu po prostu nie ma.

Z konstruktorem kopiującym jest już trochę gorzej:

```
template <class T,size_t N>  Stack<T,N>::Stack(const Stack<T,N>& s):
v(new T[nelems = s.nelems]) {
 if( s.top > 0 )
 for(top = 0; top < s.top; top++)
 v[top] = s.v[top]; /* tu może zostać rzucony wyjatek */
}
```

Podobnie jak poprzednio, w wypadku niepowodzenia wyrażenie `new` posprząta po sobie. Ale wyjątek może zostać rzucony również przez operator przypisania klasy `T`. Wtedy będziemy mieli do czynienia z wyciekiem pamięci, ponieważ nie zostanie wywołany destruktor stosu, który zwalnia pamięć `v`. Taki przykład już omawialiśmy na początku wykładu. Rozwiązaniem jest użycie `auto_ptr` lub przechwycenie wyjątku:

```
template <class T,size_t N>  Stack<T,N>::Stack(const Stack<T,N>& s):
v(new T[nelems = s.nelems]) {
 try {
   if( s.top > 0 )
   for(top = 0; top < s.top; top++)
   v[top] = s.v[top]; /* tu może zostać rzucony wyjatek */
 }
 catch(...) {
   delete [] v; throw ;
 }
}
```

To rozwiązanie zakłada, że destrukcja `v` powiedzie się, tzn. że operator przypisania:

```
v[top] = s.v[top];
```

pozostawił lewą stronę w stanie umożliwiającym jej destrukcję.

Sytuacja jest groźniejsza w przypadku operatora przypisania:

```
template <class T,size_t N> Stack<T,N>&
Stack<T,N>::operator=(const Stack<T,N>& s) {
 delete [ ] v;
 v = new T[nelems=s.nelems];
 if( s.top > 0 )
 for(top = 0; top < s.top; top++)
    v[top] = s.v[top];
 return *this;
}
```

Wyjątek rzucony przez wyrażenie `new` zostawia stos w stanie złym, z wiszącym luźno wskaźnikiem `v`. Wyjątek rzucony przez operator przypisania elementów tablicy `v` w najlepszym przypadku zostawia stos w stanie niezdefiniowanym. Implementacja, która w wypadku wystąpienia wyjątku zostawia stos w takim stanie, w jakim go zastała, jest podana poniżej:

```
template <class T,size_t N> Stack<T,N>&
Stack<T,N>::operator=(const Stack<T,N>& s) {
 T *tmp;
 try {
   tmp = new T[nelems=s.nelems];
   if( s.top > 0 )
     for(size_t i = 0; i < s.top; i++)
```

```
        tmp[i] = s.v[i];
  }
  catch(...) {delete [] tmp,throw;}
  swap(v,tmp);
  delete [] tmp;
  top=s.top;
  return *this;
}
```

Przejdźmy teraz do podstawowych funkcji stosu, zaczynając od funkcji `push`:

```
template <class T,size_t N>
void Stack<T,N>::push(const T &element) {
 if( top == nelems ) {
   T* new_buffer = new T[nelems += N];
   for(int i = 0; i < top; i++)
     new_buffer[i] = v[i];
   delete [] v;
   v = new_buffer;
 }
 v[top++] = element;
}
```

Załóżmy na początek, że nie ma potrzeby zwiększania pamięci, wykonywane jest więc tylko polecenie:

```
v[top++] = element;
```

Jak już zauważyliśmy, przypisanie może się nie powieść, wtedy stos zostanie w stanie złym lub niezdefiniowanym, ponieważ `top` zostanie zwiększone. Lepiej jest więc napisać:

```
v[top] = element;
 ++top;
```

Zobaczmy, co się dzieje, jeśli zażądamy zwiększenia pamięci. Niepowodzenie wyrażenia `new` zostawi nas ze zwiększonym polem `nelems`, pomimo że pamięć się nie zwiększyła. Wyjątek z operatora przypisania zostawi nas z wyciekiem pamięci, ponieważ pamięć przydzielona do `new_buffer` nigdy nie zostanie zwolniona. Uwzględaniając te uwagi, poprawimy funkcję `push` następująco:

```
template <class T,size_t N>
void Stack<T,N>::push(T element) {
 if( top == nelems ) {
   T* new_buffer;
   size_t new_nelems;
   try {
     new_nelems=nelems+N;
     new_buffer = new T[new_nelems];
     for(int i = 0; i < top; i++)
       new_buffer[i] = v[i];
   }
   catch(...) { delete [] new_buffer;}
     swap(v,new_buffer);
     delete [] new_buffer;
     nelems = new_nelems;
   }

 v[top] = element;
 ++top;
 }
```

Na koniec została nam jeszcze funkcja `pop`:

```
template <class T,size_t N> T Stack<T,N>::pop() {
 if( top == 0 )
   throw std::domain_error("pop on empty stack");
 return v[--top]; /* tu może nastąpić kopiowanie */
}
```

Jak widać funkcja `pop` może rzucić jawnie wyjątek `std::domain_error`. Z tym wyjątkiem nie ma problemów. Potencjalny problem stwarza za to wyrażenie:

```
return v[--top]; /* tu może nastąpić kopiowanie */
```

Ponieważ zwracamy `v[--top]` przez wartość, to może nastąpić kopiowanie elementu typu `T`. Nie musi, ponieważ kompilator ma prawo wyoptymalizować powstały obiekt tymczasowy. Jeżeli jednak zostanie wywołany konstruktor kopiujący, to może rzucić wyjątek. Wtedy stos pozostanie w zmienionym stanie, bo wartość `top` zostanie zmniejszona. Rozważmy też wyrażenie:

```
x = s.pop();
```

Jeżeli operacja przypisania się nie powiedzie, to stracimy bezpowrotnie jeden element stosu. Można by powiedzieć, że to już nie jest sprawa stosu, ale lepiej po prostu rozdzielić operacje modyfikujące stan stosu od operacji tylko ten stan odczytujących:

```
template <class T,size_t N> void Stack<T,N>::pop() {
 if( top == 0 )
   throw std::domain_error("pop on empty stack");
   --top;
 }
template<class T,size_t N> T &Stack<T,N>::top() {
 if( top == 0 )
   throw std::domain_error("pop on empty stack");

 return v[top-1];
}
template<class T,size_t N> const T &Stack<T,N>::top() const {
 if( top == 0 )
   throw std::domain_error("pop on empty stack");

 return v[top-1];
}
```

W przeciwieństwie do `pop()` operacja `top()` może zwracać wartość przez referencje. Funkcja `pop()` robić tego w ogólności nie mogła, bo potencjalnie niszczyła obiekt zdejmowany ze stosu.

Kolejny stos

Zaprezentowana w poprzedniej części implementacja stosu wymagała, aby parametr szablonu `T` posiadał:

- konstruktor domyślny.
- bezpieczny (względem wyjątków) operator przypisania.
- destruktor nie rzucający wyjątków.

Proszę zauważyć, że konstruktor domyślny właściwie niczemu nie służy. Jest potrzebny tylko po to, aby stworzyć tablicę obiektów, które potem będą tak naprawdę nadpisywane za pomocą operatora przypisania. Taka inicjalizacja i przypisanie jest w C++ dokonywana za pomocą konstruktora kopiującego. Na zakończenie przedstawię implementację klasy `Stack`, która od typu `T` potrzebuje tylko destruktora i konstruktora kopiującego. W tym celu będziemy przydzielać "gołą" pamięć oraz tworzyć i niszczyć w niej obiekty

bezpośrednio. Do tego celu wykorzystamy alokator opisany w poprzednim wykładzie. Zaczniemy od zdefiniowania pomocniczej klasy do zarządzania pamięcią:

```
template<typename T,typename Allocator = std::allocator<T> >
struct  Stack_impl : public Allocator{
 size_t _top;
 size_t _size;
 T* _buffer;

 Stack_impl(size_t n):
   _top(0),
   _size(n),
   _buffer(Allocator::allocate(_size)) {};

  Stack_impl() {
   for(size_t i=0;i<_top;++i)
     destroy(_buffer++);

   deallocate(_buffer,_size);
 }

 void swap(Stack_impl& rhs) throw() {
   std::swap(_buffer,rhs._buffer);
   std::swap(_size,rhs._size);
   std::swap(_top,rhs._top);
 }
};
```

Jedyne miejsce, gdzie może zostać rzucony wyjątek to funkcja `allocate()`, ale wtedy żadna pamięć nie zostanie przydzielona ani żaden obiekt nie zostanie stworzony. Korzystamy tu też z żądania, aby alokator był bezstanowy, inaczej funkcja `swap` musiałaby też zamieniać składowe alokatorów.

Klasa `Stack` korzysta z klasy `Stack_impl`:

```
template<typename T,size_t N = 10,
        typename Allocator = std::allocator<T> >
class Stack {
private:
  Stack_impl<T,Allocator> _impl;
```

(Źródło: stack_sutter.h)

Konstruktory:

```
public:
  Stack(size_t n = N):_impl(n) {};

 Stack(const Stack& rhs):_impl(rhs._impl) {
   while(_impl._top < rhs._impl._top) {
     _impl.construct(_impl._buffer+_impl._top, rhs._impl._buffer[_impl._top]);
     ++_impl._top;
   }
 }
```

(Źródło: stack_sutter.h)

robią się teraz prostsze. Nie ma potrzeby definiowania destruktora. Destruktor domyślny sam wywoła destruktor pola `_impl`. Jeżeli w konstruktorze kopiującym zostanie rzucony wyjątek w funkcji `construct`, to wywołany podczas zwijania stosu destruktor `Stack_impl` wywoła destruktory stworzonych obiektów i zwolni pamięć.

Operator przypisania korzysta z "triku":

```
Stack &operator=(const Stack& rhs) {
   Stack tmp(rhs);
   _impl.swap(tmp._impl);
   return *this;
 }
```

(Źródło: stack_sutter.h)

Tworzymy kopię prawej strony i zamieniamy z lewą stroną. Obiekt `tmp` jest obiektem lokalnym, więc zostanie zniszczony. Jeśli nie powiedzie się kopiowanie, to stos pozostaje w stanie niezmienionym. Proszę zauważyć, że jest to bezpieczne nawet w przypadku samopodstawienia `s=s`.

Funkcja `push` stosuje podobną technikę:

```
void push(const T &elem) {
   if(_impl._top==_impl._size) {

     Stack tmp(_impl._size+N);
     while(tmp._impl._top < _impl._top) {
        _impl.construct(tmp._impl._buffer+tmp._impl._top,
              _impl._buffer[tmp._impl._top]);
        ++tmp._impl._top;
    }
     _impl.swap(tmp._impl);
   }

   _impl.construct(_impl._buffer+_impl._top,elem);
   ++_impl._top;
 }
```

(Źródło: stack_sutter.h)

Funkcje `top()` i `pop()` pozostają praktycznie niezmienione, z tym, że funkcja `pop()` niszczy obiekt na wierzchołku stosu:

```
T &top() {
   if(_impl._top==0)
     throw std::domain_error("empty stack");
   return _impl._buffer[_impl._top-1];
 }

 void pop() {
   if(_impl._top==0)
     throw std::domain_error("empty stack");

   --_impl._top;
   _impl.destroy(_impl._buffer+_impl._top);
 }

 bool is_empty() {
   return _impl._top==0;
 }
};
```

(Źródło: stack_sutter.h)