Zaawansowane CPP/Wykład 13: Wyjątki

From Studia Informatyczne

< Zaawansowane CPP

Spis treści

Wstęp

Jesteśmy istotami omylnymi, więc niezależnie od naszych starań, pisane przez nas programy bedą zawierały usterki, dostarczane do nich dane nie zawsze będą poprawne, a i sprzęt może nie działać tak jak trzeba.

Nie oznacza to, że należy zaniechać dążenia do pisania bezbłędnego kodu, wprost przeciwnie, jakość kodu powinna być jednym z naszych priorytetów, ale należy się też pogodzić z faktem, że błędy będą występować i powinniśmy być na takie sytuacje przygotowani. Jak to mówią "najwyższą formą zaufania jest kontrola".

Na potrzeby tego wykładu zdefiniujemy bardzo luźno błąd jako wystąpienie sytuacji, która wystąpić nie powinna. Nie będziemy też interesować się bardzo tym, jak błedy wykrywać, ale raczej co zrobić, kiedy takowy wykryjemy. W następnym podrozdziale omówię bardzo pobieżnie różne możliwości reakcji na wystąpienie błędu i wprowadzę pojęcie wyjątku. Reszta wykładu będzie poświęcona zagadnieniom związanym z pisaniem kodu używającego wyjątków.

Wykrywanie błędów

Zanim przejdziemy do sytuacji, w której wiemy, że wystąpił bład, musimy poświęcić kilka akapitów na zastanowienie się czy w ogóle należy błedy wykrywać i obsługiwać. Nawet jeśli większość z Państwa krzyknie "oczywiście, że tak" (choć podejrzewam, że większość tego sama nie robi: kto ostatnio sprawdzał wartość zwróconą przez funkcję `printf`?), to i tak pozostaje pytanie, jakie błędy będziemy starali się wykrywać.

Na to pytanie nie ma jednoznacznej odpowiedzi, jak zresztą na większość pytań dotyczących decyzji projektowych. Różne projekty wymagają różnego poziomu niezawodności, a więc i różnych zabezpieczeń.

Na jednym końcu są programy, które po prostu nie mogą "paść", na drugim np. niektóre progamy symulacyjne, które wykonują się w godzinę lub mniej. Nawet jednak w tej ostatniej sytuacji, różne formy obsługi błędów mogą nam bardzo pomóc w debugowaniu. Linijki typu:

```
if( NULL==(fin=fopen(input_file_name,"r"))) {
    fprintf(stderr,"cannot open file input_file_name);
```

```
    exit(1);
  }
}
```

lub

```
if( NULL==(p=malloc(n_bytes))) {
    fprintf(stderr,"cannot allocate memory for ...");
    exit(1);
  }
}
```

mogą nam oszczędzić jeśli nie godzin, to wielu minut frustracji. Proszę zwrócić uwagę, że oba przykłady dotyczą zasobów zewnętrznych, nie do końca pod naszą kontrolą i tym bardziej powinny być sprawdzane, zwłaszcza, że będzie nas to kosztować minutę pisania i prawie tyle co nic w trakcie wykonania.

Kontrola zakresu

A co z bardziej kosztownymi testami? Typowym przykładem jest sprawdzanie zakresu. Czy np. nasz stos `Stack` powinien sprawdzać, czy wykonujemy `pop` lub `top` na stosie pustym albo `push` na stosie pełnym? Czy powinniśmy sprawdzać poprawność podanego indeksu w wyrażeniach `v[i]`?

Mam nadzieję, że Państwo nie oczekują jednoznacznej odpowiedzi na te pytania, bo jej po prostu nie ma. Niestety, tego rodzaju testy mogą być bardzo kosztowne. Operacje dostępu do elementów są bardzo proste, koszt testu będzie pewnie dominujący, a te operacje mogą być bardzo często wykonywane. Z drugiej strony błędy przekroczenia zakresu są bardzo "wredne". Rozważmy np. taki prosty kod:

```
Stack<int,5> s;

  for(int i=0;i<1000;i++)
    s.push(i);

  int i=0;
  while(1)
    std::cerr<<++i<<" "<<s.pop()<<"";
```

(Źródło: overflow.cpp)

Na moim komputerze powyższy program wykonał 20981 operacji `pop()`, zanim padł z komunikatem `Naruszenie ochrony pamięci`. Proszę zauważyć, że najpierw zapisał 995 liczb w pamięci należącej nie wiadomo do kogo! Skutki takiego błędu mogą więc wystąpić w zupełnie innym miejscu programu.

Nie ma dobrego rozwiązania tego dylematu, ale zawsze może pomóc zdrowy rozsądek. Użycie sprawdzenia zakresu w operacji mnożenia macierzy miałoby katastrofalne skutki dla wydajności kodu. Z drugiej strony jest to prosty kod, w którym łatwo zapewnić aby indeksy nie wychodziły poza zakres. Tutaj więc kontrola zakresu jest niewskazana.

Częstym rozwiązaniem jest włączanie kontroli zakresu podczas debugowania i wyłączanie jej w "produkcyjnej" wersji programu. Moim zdaniem może to być bardzo pożyteczne, zwłaszcza w językach, które dopuszczają włączanie i wyłączanie sprawdzania zakresów za pomocą opcji kompilacji (oczywiście może to dotyczyć tylko typów wbudowanych). W przypadku stosu, mogą nam się przydać w tym celu klasy wytyczne, opracowane w wykładzie 7 (http://osilek.mimuw.edu.pl/index.php?title=Zaawansowane_CPP/Wyk%C5%82ad_7:_Klasy_wytycznych). Może warto dodać, że kontenery STL, dostarczające operację indeksowania, dostarczają również metodę dostępu ze sprawdzaniem: `at(int)`.

Obsługa błędów

Załóżmy więc, że w pogramie mamy przynajmniej kilka linijek wykrywających potencjalne błędy. No i stało się. Wiemy już, że w programie wystąpił błąd, co teraz? Mamy wiele możliwości, wymienię tylko kilka z nich:

1. Kończymy program z ewentualnym komunikatem o błędzie.
2. Kończymy program, ale najpierw sprzątamy po sobie: zwalniamy zasoby, których nie zwolni system operacyjny, zapisujemy dane, itp.
3. Staramy się kontynuować program pomimo błędu, próbując go poprawić, obejść lub zrezygnować z części funkcjonalności.

W tym wykładzie nie będzie interesować nas sama konkretna strategia, ale sposób rozdzielenia procesu wykrycia błędu od wyboru strategii. Jest to problem, który dotyczy każdego kodu, ale głównie funkcji bibliotecznych. Ich projektant/programista może wykryć, że w trakcie ich wykonywania wystąpiła nieprawidłowość, ale nie może jednak wiedzieć jak z takim błędem postąpić. To jest decyzja osoby korzystającej z tej funkcji. Musi więc istnieć jakiś mechanizm przekazywania tej informacji z wywoływanej funkcji na zewnątrz do funkcji wywołującej.

Najprostyszym sposobem jest zwrócenie jakiejś wartości sygnalizującej błąd. Jeśli funkcja nie zwraca żadnego wyniku, to jest to proste; jeśli jednak funkcja ma zwracać jakiś wynik, to nie zawsze da się znaleźć taką wartość, która by jednoznacznie mogła definiować błąd. Rozszerzenia tej metody, to zwrócenie informacji o przebiegu funkcji poprzez dodatkowy argument przekazywany przez referencje. Można też ustawiać i odczytywać jakieś zmienne stanu. Największą wadą tego podejścia jest konieczność każdorazowego sprawdzania tych wartości, co wymaga pisania dużej ilości trywialnego kodu. Z tego powodu sprawdzanie poprawności wywołania takich funkcji jest często opuszczane. W C++ dochodzi jeszcze niemożność zwrócenia wartości z konstruktora (choć oczywiście możemy ustawić w nim zmienną stanu informującą o powodzeniu konstrukcji).

Wyjątki

C++ dostarcza nowego mechanizmu, jakim są wyjątki. Polega on na tym, że funkcja która błąd wykryje i nie chce lub nie może go obsłużyć sama, rzuca wyjątek, który może być dowolnym obiektem. Rzucenie wyjątku powoduje natychmiastowe przerwanie wykonywania funkcji. Procedura wywołująca może ten wyjątek złapać. Wyjątek niezłapany prowadzi do zatrzymania programu, a więc wyjątki nie mogą zostać zignorowane. Zilustruję to na przykładzie naszego stosu, do którego dodam instrukcje rzucające wyjątki w przypadku przekroczenia zakresu (dla prostoty nie będę korzystał z klas wytycznych):

```
template<typename T = int , size_t N = 100> class Stack {
private:
  T _rep[N];
  size_t _top;
public:
  Stack():_top(0) {};
  void push(T val) {
    if(_top == N) {
      throw "pushing on top of the full stack";
    }
    _rep[_top++]=val;
  }
  T pop() {
    if(is_empty()) {
      throw "poping form an empty stack";
    }
    return _rep[--_top];
  }
  bool is_empty() const     {return (_top==0);}
};
```

(Źródło: stack_except.h)

Polecenie `throw` służy właśnie do rzucania wyjątków. W tym wypadku rzucane są stałe napisowe, które będą automatycznie konwertowane na typ `const char *`. Wykonanie programu:

```
main() {
  Stack<int,5> s;
  s.push(1);
  s.pop();
  s.pop(); /* tu będzie rzucony wyjątek */

  for(int i=0;i<10;i++)
  s.push(i); /* tu też gdyby udało się tu dojść */
}
```

(Źródło: overflow.cpp)

spowoduje przerwanie programu w trakcie wykonywania drugiego polecenia `pop`. Komunikat, który się przy tym pojawia jest zależny od implementacji.

Wyjątki można łapać, korzystając z bloku `try`:

```
Stack<int,5> s;
  s.push(1);
  try {
    s.pop();
    s.pop();
  }
  catch(const char *msg) {
    std::cerr<<msg<<std::endl;
  }
  try {
      for(int i=0;i<10;i++)
    s.push(i);
  }
  catch(const char *msg) {
    std::cerr<<msg<<std::endl;
  }
```

(Źródło: stack_except.cpp)

W bloku `try` umieszczamy instrukcje, które mogą potencjalnie rzucić wyjątek. Za blokiem `try` umieszczamy jedną lub więcej klauzul `catch`, które te wyjątki łapią. Wyjątek rzucony w bloku `try` powoduje przekazanie sterowania do pierwszej pasującej klauzuli `catch`.

Wyjątki złapane

Przyjrzyjmy się teraz dokładniej mechanizmowi rzucania i łapania wyjątków. Rozważmy prosty przykład:

```
struct X {
  int val;
  X(int i=0):val(i) {cerr<<"constructing "<<val<<"";}
   X() {cerr<<"destructing "<<val<<endl;}
};

void f()  {
  X x(1);
  throw 0;
  cout<<"f";
};

main(){
  X  y(2);
  try {
    X z(3);
    f();
    cout<<"try";
  }
  catch(double){cout<<"zlapalem double-a";}
```

```
  catch(int){cout<<"zlapalem int-a";}
  catch(...){cout<<"zlapalem cos ";}

 cout<<"main";
}
```

(Źródło: caught.cpp)

Oto wynik wykonania tego programu:

```
constructing 2
constructing 3
constructing 1
destructing 1
destructing 3
zlapalem int-a
main
destructing 2
```

Co możemy zauważyć?

1. Wyjątek przerwał wykonywanie funkcji `f()` i bloku `try`, sterowanie zostało przekazana do klauzuli `catch(int)`.
2. Przedtem wywołane zostały destruktory obiektów `x` i `z`, czyli lokalnych obiektów w zasięgu bloku `try`. Ten proces nazywamy "zwijaniem stosu".
3. Po wykonaniu klauzuli `catch` sterowanie zostało przekazane do następnego wyrażenia.

Klauzula `catch(...)` wyłapuje każdy wyjątek. Jeśli np. pominiemy klauzulę `catch(int)`:

```
catch(double){cout<<"zlapalem double-a";}
//catch(int){cout<<"zlapalem int-a";}
catch(...){cout<<"zlapalem cos ";}
```

to wynikiem wywołania programu będzie:

```
constructing 2
constructing 3
constructing 1
destructing 1
destructing 3
zlapalem cos
main
destructing 2
```

Z tego przykładu widać też, że w przypadku dopasowywania klauzul `catch` nie następuje niejawna konwersja argumentów.

Niezłapane wyjątki

A co się stanie, jeśli wyjątku nie złapiemy? Żeby się o tym przekonać usuniemy kolejną klauzulę `catch`:

```
catch(double){cout<<"zlapalem double-a";}
//catch(int){cout<<"zlapalem int-a";}
//catch(...){cout<<"zlapalem cos ";}
```

Wynik programu jest teraz zupełnie inny:

```
constructing 2
constructing 3
constructing 1
terminate called after throwing an instance of 'int'
Abort
```

Niezłapany wyjątek spowodował wywołanie funkcji `abort()`, która zakończyła program bez wywołania destruktorów. Ściśle rzecz biorąc, niezłapany wyjątek wywołuje funkcję `terminate()`, która z kolei domyślnie wywołuje funkcję `abort()`. Co do tego, czy wywoływane są destruktory lokalnych obiektów (zwijanie stosu), to jest to zachowanie zależne od implementacji. Jak widać, w implementacji `g++` w przypadku niezłapania wyjątku destruktory obiektów nie są wywoływane.

Domyślne zachowanie funkcji `terminate()` można zmienić, ustawiając własną funkcję, za pomocą:

```
namespace std {
typedef void (*terminate_handler)(void);
terminate_handler set_terminate(terminate_handler new_terminate);
}
```

Funkcja ustawiana w tym poleceniu nie może zwrócić sterowania, taka próba kończy sie wywołaniem funkcji `abort()`. Oznacza to, że funkcja `new_terminate()` musi kończyć się wywołaniem `abort()` lub `exit()`.

```
void my_terminate() {std::cerr<<"terminating "<<std::endl;exit(1);}

main() {
  std::set_terminate(my_terminate);
  throw 0;
}
```

(Źródło: terminate.cpp)

Wyjątki w destruktorach

Jeśli podczas opisanego powyżej procesu obsługi wyjątku, wywołana zostanie funkcja, która sama wywoła wyjątek, to program zostanie natychmiast przerwany wykonaniem funkcji `terminate()` (nie dotyczy to już funkcji wywoływanych wewnątrz klauzuli `catch`). W szczególności stanie się to, jeśli któryś z destruktorów wywoływanych w trakcie zwijania stosu rzuci wyjątek:

```
struct X {
  ~X() {
    std::cerr<<std::uncaught_exception()<<"";
    throw 0;};
} ;

main() {
  try
    {
      X x;
    }
  catch(int) {};

  try {
    X x;
    throw 0;
  }
  catch(int) {};
}
```

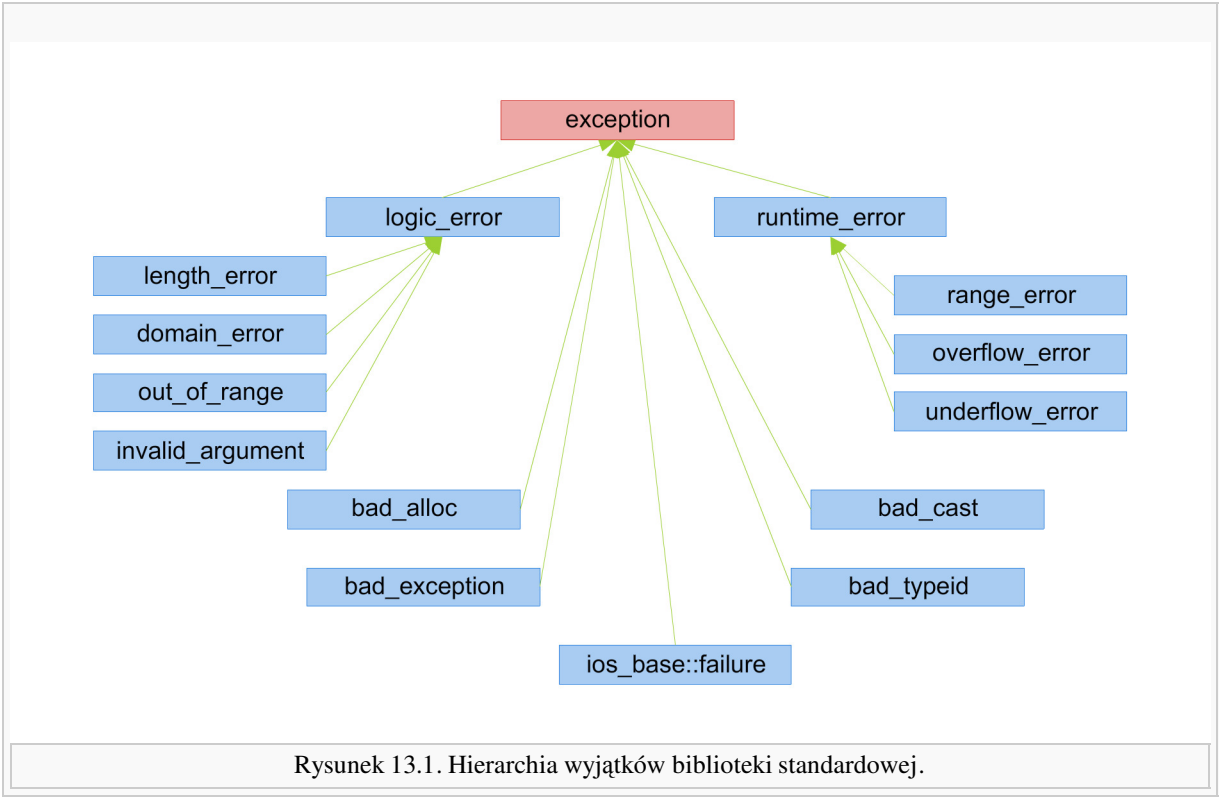
(Źródło: destruktor.cpp)

W powyższym kodzie destruktor klasy `X` jest wołany dwa razy. Po raz pierwszy, podczas wychodzenia z pierwszego bloku `try`. Jest to normalne wywołanie spowodowane wyjściem poza zakres. Destruktor rzuca wyjątek, który zostaje wyłapany przez klauzulę `catch(int)` na końcu bloku. Drugi raz destruktor jest wołany jako część zwijania stosu po wyjątku rzuconym jawnie w drugim bloku `try`. Mimo że łapiemy wyjątki `int`, i tak w tej sytuacji wywoływana jest funkcja `terminate()`, a w konsekwencji i `abort()`. Jest to jeden z powodów, dla których destruktory nie powinny rzucać wyjątków. Funkcja `uncaught_exception()` umożliwia rozróżnienie tych dwu kontekstów wywołania destruktora. Zwraca ona prawdę, jeśli jakiś wyjątek jest właśnie obsługiwany.

Inne powody nie rzucania wyjątków z destruktorow wiążą się z dynamiczną alokacją pamięci i zostaną omówione w kolejnym wykładzie.

Hierachie wyjątków

Jako wyjątek może zostać wyrzucony dowolny obiekt. Umożliwia nam to grupowanie wyjątków w hierarchie za pomocą dziedziczenia. Zilustrujemy to za pomocą hierachii wyjątków z biblioteki standardowej, przedstawionej na rysunku 13.1.



Rysunek 13.1. Hierarchia wyjątków biblioteki standardowej.

Można z niej korzystać np. następująco:

```
main() {
  try {
        throw domain_error() ;
  }
  catch(invalid_argument &e) {
    cerr<<e.what()<<"";
  }
  catch(logic_error &e) {
    cerr<<"logic "<<e.what()<<"";
  }
  catch(exception &e) {
    cerr<<"some exception "<<e.what()<<"";
```

```
  }
}
```

(Źródło: hierarchy.cpp)

Kolejność klauzul `catch` jest ważna, ponieważ klauzule są sprawdzane po kolei i pierwsza, która pasuje, zostanie wykonana. Gdybyśmy więc podali kaluzulę `catch(Exception &e)` jako pierwszą, przechwyciła by ona wszystkie standardowe wyjątki. Ważne jest też, aby korzystając z hierarchii dziedziczenia, przechwytywać wyjątki przez referencję. Inaczej nie zostaną wywołane poprawne funkcje wirtualne. Zachęcam do eksperymentów z powyższym kodem.

Deklaracje wyjątków

C++ pozwala na deklarowanie listy możliwych wyjątków rzucanych z funkcji. Służy do tego deklaracja `throw (...)` umieszczana za deklaracją listy argumentów funkcji, np.:

```
void no_throw(int) throw();
```

deklaruje, że funkcja `no_throw` nie rzuci żadnego wyjątku, natomiast

```
void throw_std(int) throw(exception);
```

deklaruje, że funkcja `throw_std` będzie rzucać tylko wyjątki ze standardowej hierarchii. Brak deklaracji oznacza, że funkcja może rzucać co chce.

Niestety, C++ nie dostarcza nam praktycznie żadnych mechanizmów, które by mogły wymusić konsystencję tych deklaracji. Rozważmy implementację funkcji:

```
void f(int i) {throw i;}
void g() throw(int) {throw 0;};
void no_throw(int i) {f(i;};
```

Funkcja `f` proklamuje całemu światu, że może rzucać wyjątek (poprzez brak deklaracji, że nie może), podobnie funkcja `g()`, a pomimo to kompilator nie pozwala na to, aby wywołać ją wewnątrz funkcji, która jawnie deklaruje, że wyjątku nie rzuci. Proszę to porównać np. z zastosowaniem kwalifikatora `const`: funkcja zadeklarowana jako `const` nie może wywołać funkcji, które `const` nie są. W przypadku wyjątków narzucenie takiej konsystencji spowodowałoby, że potrzeba by przerabiać ogromne ilości kodu napisanego, zanim mechanizm wyjątków stał się używany. Sprawia to niestety, że deklaracje wyjątków nie są zbyt użyteczne, łatwo bowiem niechcący napisać kod, który je złamie. A konsekwencje tego są poważne. Sprawdzania konsystencji w czasie kompilacji wprawdzie nie ma, ale jest sprawdzanie w czasie wykonania. Jeżeli funkcja rzuci wyjątek, który nie znajduje się na jej liście zadeklarowanych wyjątków, to następuje wywołanie funkcji `unexpected()`, która domyślnie wywołuje `abort()`. Nawet złapanie tego wyjątku nic nie pomoże.

Kolejnym problemem są szablony funkcji i metody szablonów klas. Ich projektant nie może przewidzieć, z jakimi argumentami zostaną one konkretyzowane, a więc jakie wyjątki mogą zostać rzucone. Dlatego w szablonach lepiej deklaracje wyjątków pomijać. W ogóle, deklaracje wyjątków należy umieszczać tam, gdzie uważamy, że rzucenie każdego innego wyjątku jest przejawem poważnego błędu. Najczęściej jest to sytuacja, kiedy chcemy zadeklarować, że dana funkcja w ogóle nie rzuca wyjątków. Jak już pisałem taką własność powinny posiadać destruktory.

Niespodziewane wyjątki

Podobnie jak w przypadku funkcji `terminate()`, możemy podstawić własną funkcję `unexpected()`. Podobnie jak `terminate()` funkcja `unexpected()` nie zwraca sterowania, może za to sama rzucić

wyjątek. W ten sposób możemy jej użyć do "podmiany" niespodziewanego wyjątku na inny. Zobaczmy jak to działa:

```
void f() throw(int) {
  throw "niespodzianka!";
};

main() {
  try {
  f();
  }
  catch(...) {};
}
```

(Źródło: unexpected.cpp)

Ponieważ `f()` rzuca `const char *`, a deklaruje tylko `int`-a, powyższy program wywoła funkcję `unexpected()`, która przerwie program. Jeżeli podmienimy wyjątek poprzez ustawienie odpowiedniej funkcji:

```
void unexpected_handler() {throw 0;};
std::set_unexpected(unexpected_handler);
```

to zamiast `const char *` zostanie rzucony `int` i złapany przez klauzulę `catch(...)`. Tak się stanie ponieważ `int` jest na liście wyjątków funkcji `f()`. Gdyby nie był, to zostałaby wywołana funkcja `terminate()`. Jeżeli jednak deklaracja wyjątków funkcji `f()` zawierać będzie wyjątek `std::bad_exception`, to każdy wyjątek rzucony przez `unexpected()` i nie znajdujący się na liście zadeklarowanych wyjątków funkcji `f()`, jest podmieniany na `std::bad_exception()`:

```
void f() throw(int,std::bad_exception) {
  throw "niespodzianka!";
};

void unexpected_handler() {throw 3.1415926;};

main() {
  std::set_unexpected(unexpected_handler);
  try {
  f();
  }
  catch(std::bad_exception ) {};
}
```

(Źródło: unexpected.cpp)

Wydajność wyjątków

Autor pozycji *"Język C++ bardziej efektywny"* S. Meyers sugerował, że użycie mechanizmu wyjątków może spowolnić program, nawet jeśli wyjątki nie będą rzucane. Ponieważ od tego czasu minęło dobrych kilka lat, postanowiłem sam sprawdzić, jak się sprawy mają. W tym celu skorzystałem z następujących programików:

```
double scin(double x,bool flag) {
  if(flag) throw 0;
  return sin(x);
}
main(){
  volatile int f=0;
  double s=0.0;
  for(int i=0;i<100000000;++i) {
    try {
```

```
    s+=scin(rand()/(double)RAND_MAX,f);
    } catch(int) {};
  }
}
```

(Źródło: exceptions.cpp)

oraz

```
double scin(double x,bool flag) {
  if(flag) return 0;
  return sin(x);
}

main(){
  volatile int f=0;
  double s=0.0;
  for(int i=0;i<100000000;++i)
    s+=scin(rand()/(double)RAND_MAX,f);
}
```

(Źródło: no_exceptions.cpp)

Jak widać drugi z nich nie ma nawet śladu wyjątków. W pierwszym podejściu ustawilem flagę `f` na zero, co powodowało, że żaden wyjątek nie był rzucany. Czas wykonania obu programów (w sekundach) podany jest w poniższej tabelce:

	nie rzucane wyjątki	bez wyjątków	rzucane wyjątki	`return`
-O0	15	15		
-O1	13	4		
-O2	13	4		
-O3	4	4	600	4

Porównując kolumny 1 i 2, widać, że dla pełnej optymalizacji nie ma żadnej różnicy. Szczegółowe badanie wykazało, że to włączenie opcji `-finline-functions` powoduje skok prędkości pomiędzy dwoma ostatnimi wierszami w pierwszej kolumnie. Ten sam efekt można uzyskać, dodając do funkcji `scin` kwalifikator `inline`.

Następnie porównałem koszt zwykłego powrotu z funkcji z kosztem rzucenia wyjątku. Wyniki są przedstawione w dwóch ostatnich kolumnach tabelki. Tu widać dramatyczną różnicę: obsługa wyjątku jest ponad 100 razy wolniejsza.
