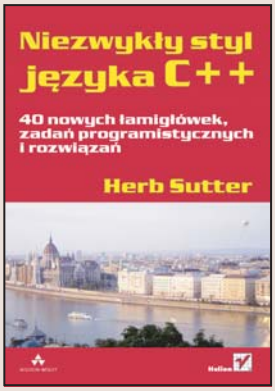
Wyjątkowy styl języka C++. 40 nowych łamigłówek,zadań programistycznych i rozwiązań



Autor: Herb Sutter
Tłumaczenie: Tomasz Walczak
ISBN: 83-246-0061-2
Tytuł oryginału: Exceptional C++ Style: 40 New Engineering Puzzles, Programming and Solutions (C++ in Depth Series)
Format: B5, stron: 304

Zaprojektuj i napisz wydajniejsze oprogramowanie

- Poznaj najlepsze metody stosowania biblioteki STL
- Zaimplementuj wydajne mechanizmy zarządzania pamięcią i zasobami
- Zoptymalizuj kod źródłowy swoich aplikacji

Projektowanie i tworzenie wydajnych aplikacji to sztuka znajdowania kompromisu pomiędzy kosztami a funkcjonalnością, elegancją i łatwością pielęgnacji oraz między elastycznością i nadmierną złożonością. Znalezienie takiego „złotego środka" jest zadaniem wymagającym znajomości najlepszych praktyk programistycznych. Guru języka C++, Herb Sutter, w książce „Wyjątkowy język C++. 40 nowych łamigłówek, zadań programistycznych i rozwiązań" przedstawił najistotniejsze zasady stosowania biblioteki standardowej, reguły inżynierii oprogramowania i wiele innych tematów związanych z tworzeniem programów w języku C++. Książka ta jest kontynuacją jego rozważań i rad dla programistów chcących pisać wydajne oprogramowanie.

W książce Herb Sutter koncentruje się na stylu pisania kodu źródłowego. Przedstawia 40 nowych przykładów, dzięki którym dowiesz się nie tylko, co się dzieje w programie, ale także w jaki sposób. Czytając ją, poznasz nowe sposoby stosowania kluczowych elementów języka C++. Każde z zagadnień przedstawione jest w formie zagadki z rozwiązaniem. Dzięki temu lepiej zapamiętujemy metodykę postępowania, co ułatwia wykorzystanie jej w codziennej pracy.

- Zasady programowania uogólnionego
- Niestandardowe zastosowania biblioteki STL
- Bezpieczna obsługa wyjątków
- Reguły projektowania klas
- Efektywne zarządzanie pamięcią
- Optymalizowanie aplikacji pod kątem wydajności
- Unikanie pułapek w kodzie

Jeśli chcesz poprawić stabilność i wydajność swoich programów, sięgnij po kolejny poradnik autorstwa Herba Suttera.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 2.

Zagadnienia i techniki związane z bezpieczną obsługą wyjątków

Obsługa wyjątków jest podstawowym mechanizmem zgłaszania błędów w języku C++ i innych współczesnych językach programowania. W książkach *Exceptional C++*[1] [Sutter00] oraz *More Exceptional C++*[2] [Sutter02] szczegółowo przedstawiam wiele zagadnień związanych z określeniem, czym jest bezpieczna obsługa wyjątków i jak pisać kod z bezpieczną obsługą wyjątków. Opisuję także właściwości języka i interakcje, o których musisz pamiętać.

W tym rozdziale kontynuuję te rozważania, skupiając się na pewnych właściwościach języka specyficznych dla obsługi wyjątków. Na początku odpowiadam na odwieczne pytanie — czy bezpieczna obsługa wyjątków to tylko wpisywanie `try` i `catch` w odpowiednich miejscach? Jeśli nie, to co jeszcze? Nad czym musisz się zastanowić, tworząc w programie schemat obsługi wyjątków?

Odchodząc nieco od tematu — warto poświęcić całe zagadnienie, aby przedstawić powody, dla których pisanie kodu z bezpieczną obsługą wyjątków to czysta korzyść. Takie postępowanie wiąże się ze stylem programowania, który prowadzi do bardziej stabilnego i łatwiejszego w pielęgnacji kodu, pomijając nawet korzyści wynikające ze stosowania wyjątków. Jest jednak pewne ograniczenie tych korzyści i myślenia na zasadzie „im więcej, tym lepiej". W przypadku specyfikacji wyjątków ograniczenie to jest szczególnie widoczne. Dlaczego wyjątki istnieją w języku? Dlaczego ich występowanie jest dobrze uzasadnione? I dlaczego mimo to nie powinieneś używać ich w programach?

---

[1] Wydanie polskie: *Wyjątkowy język C++. 47 łamigłówek, zadań programistycznych i rozwiązań*, WNT, 2002 — *przyp. tłum.*

[2] Wydanie polskie: *Wyjątkowy język C++. 40 nowych łamigłówek, zadań programistycznych i rozwiązań*, Helion, 2005 — *przyp. tłum.*

Tego i innych rzeczy dowiesz się, czerpiąc z fontanny wiedzy współczesnego „wyjątkowego" środowiska programistów.

Zagadnienie 11. Bloki try i catch **Stopień trudności: 3**

Czy bezpieczna obsługa wyjątków to tylko wpisywanie `try` *i* `catch` *w odpowiednich miejscach? Jeśli nie, to co jeszcze? Nad czym musisz się zastanowić, tworząc w programie schemat obsługi wyjątków?*

Pytanie profesora

1. Do czego służy blok `try`?

Pytania magistra

2. „Pisanie kodu z bezpieczną obsługą wyjątków polega głównie na wpisywaniu `try` i `catch` w odpowiednich miejscach". Przeprowadź analizę tego stwierdzenia.

3. Kiedy należy stosować bloki `try` i `catch`? Kiedy nie należy ich stosować? Przedstaw odpowiedź w formie wskazówki co do dobrego stylu programowania.

Rozwiązanie

Zabawa w berka

1. ***Do czego służy blok*** `try`***?***

Blok `try` to fragment kodu (złożone wyrażenie), który program próbuje wykonać. Po bloku tym znajduje się jeden lub więcej bloków `catch`, do których program przechodzi w sytuacji przechwycenia zgłoszonego w bloku `try` wyjątku odpowiedniego typu. Na przykład:

```
// Przykład 11.1. Przykładowy blok try
//
try {
   if ( pewien_warunek )
      throw string( "To jest ciąg znaków" );
   else if ( pewien_inny_warunek )
      throw 42;
}
catch ( const string& ) {
   // Zrób coś, jeśli przechwycony został ciąg znaków
}
catch(...) {
   // Zrób coś, jeśli przechwycony zostanie dowolny inny wyjątek
}
```

W przykładzie 11.1 kod w bloku `try` może zgłosić jako wyjątek ciąg znaków lub liczbę całkowitą, a może też w ogóle nie zgłosić wyjątku.

Życie to nie tylko zabawa w berka

2. ***„Pisanie kodu z bezpieczną obsługą wyjątków polega głównie na wpisywaniu w odpowiednich miejscach*** `try` ***i*** `catch`***”. Przeprowadź analizę tego stwierdzenia.***

Krótko mówiąc, takie stwierdzenie obrazuje podstawowy błąd w rozumieniu bezpieczeństwa wyjątków. Wyjątki są po prostu jednym ze sposobów zgłaszania błędów i na pewno wiesz, że pisanie kodu odpornego na błędy nie polega jedynie na sprawdzaniu zwracanych wartości i obsłudze warunków powodujących te błędy.

W rzeczywistości okazuje się, że bezpieczna obsługa wyjątków rzadko wiąże się z wpisywaniem `try` i `catch` — im rzadziej, tym lepiej. Powinieneś też zawsze pamiętać, że o bezpieczną obsługę wyjątków trzeba zadbać już na etapie projektowania kodu. Nie jest to element, który można dodać na końcu, dopisując kilka dodatkowych instrukcji `catch`.

Z pisaniem kodu z bezpieczną obsługą wyjątków wiążą się trzy główne zagadnienia:

1. *Gdzie i kiedy należy zgłaszać wyjątki?* Ta kwestia dotyczy umieszczania instrukcji `throw` w odpowiednich miejscach. W szczególności musisz rozważyć:
   - Które fragmenty kodu powinny zgłaszać wyjątki? Wiąże się to z wyborem błędów, które obsługiwane będą za pomocą zgłoszenia wyjątku, a nie za pomocą zwracania wartości informującej o błędzie lub innej techniki.
   - Które fragmenty kodu nie powinny zgłaszać wyjątków? W szczególności — które fragmenty kodu muszą być bezbłędne? (Patrz też zagadnienie 12. oraz [Sutter99]).
2. *Gdzie i kiedy należy obsłużyć wyjątek?* Jest to jedyne zagadnienie związane z wpisywaniem w odpowiednich miejscach `try` i `catch`, co jednak zwykle można zautomatyzować. Po pierwsze, zastanów się nad poniższymi pytaniami:
   - Które fragmenty kodu mogą przechwytywać błędy? Wymaga to określenia, które fragmenty kodu mają odpowiedni kontekst i informacje pozwalające na obsługę błędu zgłoszonego przez wyjątek (zwykle poprzez przekształcenie wyjątku na inną postać). Zauważ, że także przechwytujący kod musi mieć informacje niezbędne do porządkowania, na przykład do zwolnienia dynamicznie przydzielonych zasobów.
   - Które fragmenty kodu *powinny* przechwytywać błędy? W tym miejscu należy wybrać najbardziej do tego odpowiednie spośród fragmentów kodu, które mogą przechwytywać błędy.

Po udzieleniu odpowiedzi na te pytania zwróć uwagę na to, że stosowanie idiomu „alokacja zasobów jest inicjalizacją” pozwala wyeliminować wiele bloków `try` dzięki automatyzacji porządkowania. Jeśli opakujesz dynamicznie przydzielane zasoby w zarządzające nimi obiekty, zwykle destruktor będzie mógł zwolnić je automatycznie bez potrzeby używania bloków `try` i `catch`. Taka sytuacja jest oczywiście pożądana, nie wspominając, że taki kod jest zwykle łatwiejszy do napisania i zrozumienia.

Powinieneś automatycznie obsługiwać wyjątki związane z porządkowaniem za pomocą destruktorów, a nie za pomocą bloków `try` *i* `catch`*.*

3. *Jeśli wyjątek zostanie zgłoszony w dowolnym miejscu, to czy pozostała część kodu będzie bezpieczna?* To zagadnienie dotyczy poprawnego zarządzania zasobami, związanego z unikaniem wycieków, z pielęgnacją klas, z niezmiennikami i innymi elementami poprawności programu. Mówiąc inaczej, polega to na zapobieganiu błędom programu wynikającym z przechodzenia wyjątku z miejsca jego zgłoszenia przez te fragmenty kodu, które nie powinny troszczyć się o wyjątek przed jego dotarciem do miejsca przechwycenia i obsługi. Dla większości programistów, z którymi pracowałem, jest to zdecydowanie najbardziej czasochłonny i najtrudniejszy do opanowania aspekt bezpiecznej obsługi wyjątków.

Zauważ, że tylko jeden z tych trzech problemów ma coś wspólnego z pisaniem `try` oraz `catch`, a nawet ten można łatwo rozwiązać dzięki rozsądnemu zastosowaniu destruktorów do zautomatyzowania porządkowania.

3. ***Kiedy należy stosować bloki*** `try` ***i*** `catch`***? Kiedy nie należy ich stosować? Przedstaw odpowiedź w formie wskazówki co do dobrego stylu programowania.***

Poniżej przedstawiam pewne sugestie. W skrócie:

1. *Określ ogólny schemat zgłaszania i obsługi błędów dla swoich aplikacji lub podsystemów, a następnie stosuj go konsekwentnie*. W szczególności schemat ten powinien zawierać podstawowe elementy przedstawione poniżej (zwykle zawiera ich znacznie więcej):

   - *Zgłaszanie błędów*. Określ, jakie rodzaje błędów funkcje powinny zgłaszać oraz w jaki sposób. Staraj się używać wyjątków zamiast innych metod zgłaszania błędów. Zwykle dobrze jest dla każdej sytuacji określić jedną domyślną metodę, która jest najbardziej czytelna i łatwa w pielęgnacji. Na przykład wyjątki są najbardziej użyteczne dla konstruktorów i operatorów, które nie mogą zwracać wartości, a także w sytuacjach, kiedy miejsce zgłaszania błędu jest znacznie oddalone od miejsca jego obsługi.

   - *Przekazywanie błędów*. Między innymi powinieneś zdefiniować granice, których wyjątki nie powinny przekraczać. Zwykle są to granice modułów lub API.

   - *Obsługa błędów*. Między innymi tam, gdzie to możliwe, powinieneś przenieść funkcje zarządzające porządkowaniem do obiektów i ich destruktorów, zamiast do bloków `try` i `catch`.

2. *Umieszczaj instrukcje* `throw` *w tych miejscach wykrycia błędów, w których błędów nie można obsłużyć na miejscu*. Kod, w którym można natychmiast rozwiązać problem, oczywiście nie musi tego problemu zgłaszać.

   W przypadku każdej operacji opisz, jakie wyjątki może zgłaszać i dlaczego. Powinno być to częścią dokumentacji każdej funkcji i modułu. Nie musisz pisać specyfikacji wyjątków dla każdej funkcji (a nawet nie powinieneś — patrz

zagadnienie 13.), ale powinieneś jasno i dokładnie opisać, czego użytkownik może się spodziewać. Obsługa błędów jest częścią interfejsu funkcji i modułów.

3. *Umieszczaj* `try` *i* `catch` *w miejscach, w których program ma wystarczające informacje do obsługi błędu, jego translacji oraz do wymuszenia ograniczeń określanych przez schemat obsługi błędów*. Uważam, że istnieją trzy główne powody dodawania bloków `try` i `catch`:

   - *Obsługa błędu*. To prosty przypadek. Zdarzył się błąd, wiemy, co z nim zrobić, więc robimy to. Życie toczy się dalej (oprócz samego wyjątku, który odchodzi na zasłużony odpoczynek). Pamiętaj — jeśli to możliwe, użyj destruktora; jeśli nie, możesz użyć bloków `try` i `catch`.

   - *Translacja wyjątku.* Oznacza to przechwycenie wyjątku, który zgłasza problem niższego poziomu, a następnie zgłoszenie nowego wyjątku, sformułowanego na wyższym poziomie w kontekście przekształcającego go kodu. Oryginalny wyjątek może też zostać przekształcony na inną reprezentację, na przykład na kod błędu.

     Wyobraź sobie przykładową klasę sesji służącą do obsługi komunikacji, która działa dla hostów różnych typów oraz różnych protokołów przesyłania danych. Próba otwarcia sesji na innym serwerze może się nie powieść z wielu przyczyn niskiego poziomu, które może wykryć klasa obsługująca tę sesję. (Na przykład brak dostępu do sieci lub nieudzielenie dostępu ze strony hosta zdalnego). Funkcja `Open` może obsługiwać takie zdarzenia samodzielnie, dlatego błędów nie trzeba zgłaszać funkcji wywołującej, w kontekście której nie istnieją informacje o pakiecie `Foo` lub o tym, co zrobić, jeśli zwrócona zostanie nieznana wartość. Klasa sesji bezpośrednio obsługuje wewnętrzne błędy niskiego poziomu, utrzymuje poprawny stan oraz zgłasza błędy wyższego poziomu lub wyjątki, aby poinformować funkcję wywołującą, że otwarcie sesji zakończyło się niepowodzeniem.

```
void Session::Open( /*...*/ ) {
   try {
      // Cała operacja
   }
   catch ( const ip_error& err ) {
      // - Zrób coś z błędem IP
      // - porządkowanie
      throw Session::OpenFailed();
   }
   catch ( const KerberosAuthentFail& err ) {
      // - Zrób coś z błędem autoryzacji
      // - porządkowanie
      throw Session::OpenFailed();
   }
   // .. itd....
}
```

   - *Przechwytywanie za pomocą* `catch(...)` *wyjątków na granicach podsystemów lub innych jednostek czasu wykonania*. Operacja ta zwykle wiąże się z translacją błędu, zwykle na kod błędu lub inną reprezentację różną od wyjątków. Na przykład, kiedy stos rozwija się do C API, masz tylko dwie możliwości — zwrócić kod błędu do aktualnej funkcji API lub

ustawić stan błędu, co pozwala funkcji wywołującej sprawdzić go za pomocą odpowiedniej funkcji API `GetLastError`.

Określ ogólny schemat zgłaszania błędów oraz ich obsługi dla aplikacji lub podsystemu, a następnie stosuj go konsekwentnie. Pamiętaj o schemacie zgłaszania błędów, przekazywania błędów oraz ich obsługi.

Umieszczaj instrukcje `throw` w tych miejscach wykrycia błędów, w których błędów nie można obsłużyć na miejscu.

Umieszczaj bloki `try` i `catch` w miejscach, w których program ma wystarczające informacje do obsługi błędu i jego translacji oraz do wymuszenia ograniczeń określanych przez schemat obsługi błędów (na przykład przechwytuje za pomocą `catch(...)` wszystkie błędy na granicach podsystemów lub innych jednostek czasu wykonania).

Podsumowanie

Pewien mędrzec powiedział kiedyś:

prowadź, podążaj śladem albo usuń się z drogi!

W przypadku analizy bezpiecznej obsługi wyjątków można to sparafrazować tak:

zgłaszaj, przechwytuj albo usuń się z drogi!

W praktyce ostatni przypadek — „usuń się z drogi" — stanowi istotną część analizy i testów bezpieczeństwa wyjątków. Jest to podstawowy powód, dla którego pisanie kodu z bezpieczną obsługą wyjątków nie polega głównie na odpowiednim wpisywaniu `try` i `catch`. Jego istotą jest schodzenie z toru pocisku w odpowiednim momencie.

Zagadnienie 12. Bezpieczna obsługa wyjątków — czy warto? Stopień trudności: 7

Czy pisanie kodu z bezpieczną obsługą wyjątków jest warte wysiłku? Kwestia ta nie powinna budzić żadnych wątpliwości… jednak czasem nadal się tak dzieje.

Pytania do profesora

1. Powtórka — krótko zdefiniuj, jakie gwarancje, zdaniem Abrahamsa, powinna zapewniać bezpieczna obsługa wyjątków (słabą, mocną i niezawodności).

2. Kiedy warto pisać kod, który zapewnia:
   a) gwarancję słabą?
   b) gwarancję mocną?
   c) gwarancję niezawodności?

Rozwiązanie

Gwarancje Abrahamsa

1. ***Powtórka — krótko zdefiniuj, jakie gwarancje, zdaniem Abrahamsa, powinna zapewniać bezpieczna obsługa wyjątków (słabą, mocną i niezawodności).***

   Gwarancja słaba (ang. *basic guarantee*) mówi, że nieudana operacja może zmieniać stan programu, ale nie może powodować wyciekania zasobów, a zmienione obiekty i moduły wciąż mogą zostać usunięte albo użyte w stabilny (choć niekoniecznie przewidywalny) sposób.

   Gwarancja mocna (ang. *strong guarantee*) wiąże się z semantyką transakcji typu commit-rollback. Nieudana operacja nie może powodować zmiany stanu programu w zakresie zmiany obiektów, których dotyczy. Oznacza to brak efektów ubocznych, które wpływałyby na obiekty, włączając w to poprawność ich stanu, ich zawartość oraz stan obiektów pomocniczych, takich jak wskaźniki na zawartość kontenerów.

   Wreszcie *gwarancja niezawodności* (ang. *nofail guarantee*) mówi, że niepowodzenie nie może się zdarzyć. W kategoriach wyjątków oznacza to, że operacja nie zgłasza wyjątków. (Abahams i inni, włączając w to wcześniejsze książki z serii *Exceptional C++*, początkowo używali nazwy „gwarancja niezgłaszania". Zmieniłem nazwę na „gwarancja niezawodności", ponieważ gwarancja ta dotyczy każdej formy obsługi błędów, zarówno za pomocą wyjątków, jak i za pomocą innych mechanizmów, na przykład kodu błędu.

Kiedy warto stosować mocniejsze gwarancje?

2. ***Kiedy warto pisać kod, który zapewnia:***

   a) ***gwarancję słabą?***

   b) ***gwarancję mocną?***

   c) ***gwarancję niezawodności?***

   Zawsze warto pisać kod, który zapewnia choć jedną z tych gwarancji. Wynika to z kilku przyczyn:

   1. Parafrazując znane powiedzenie — *wyjątki się zdarzają*. Po prostu tak się dzieje. Biblioteka standardowe je zgłasza. Język je zgłasza. Programiści muszą uwzględniać to w kodzie. Na szczęście nie jest to zbyt skomplikowane, ponieważ wiemy, jak trzeba to robić. Wymaga to wykształcenia kilku nawyków i sumiennego ich przestrzegania — ale w końcu tego samego wymaga nauczenie się programowania z użyciem kodów błędów.

      Dużym problemem jest, jak zawsze, obsługa błędów jako taka. Techniczna realizacja sposobu zgłaszania błędów za pomocą zwracania kodu błędu lub zgłaszania wyjątku prawie całkowicie należy do składni, podczas gdy główne różnice leżą w semantyce zgłaszania, dlatego każda z tych technik wymaga specyficznego podejścia.

2. *Pisanie kodu z bezpieczną obsługą wyjątków jest korzystne*. Kod z bezpieczną obsługą wyjątków i dobry kod idą w parze. Te same techniki, które pomagają tworzyć kod z bezpieczną obsługą wyjątków, wyznaczają standardy, których i tak powinniśmy przestrzegać. Oznacza to, że techniki tworzenia kodu z bezpieczną obsługą wyjątków są korzystne same w sobie, nawet pomijając kwestię samych wyjątków.

Aby się o tym przekonać, przyjrzyj się opisanym przeze mnie i przez innych autorów podstawowym technikom, które ułatwiają bezpieczną obsługę wyjątków:

- Stosuj zasadę „alokacja zasobów jest inicjalizacją" (ang. *resource acquisition is initialization — RAII*) do zarządzania własnością zasobów. Używanie obiektów mających zasoby, jak klasy `Lock` lub wskaźniki `shared_ptr` (patrz [Boost, Sutter02a]) jest zwykle dobrym pomysłem. Nie powinno Cię zaskoczyć, że wśród wielu ich zalet możesz także znaleźć bezpieczną obsługę wyjątków. Ile razy widziałeś już funkcje (mówimy tu oczywiście o funkcjach napisanych przez innych programistów, nie o Twoim kodzie), w których jedna ze ścieżek prowadzących do szybkiego zwrócenia wyniku nie wykonuje odpowiedniego porządkowania, ponieważ nie jest ono automatycznie zarządzane za pomocą RAII?
- Stosuj zasadę: „wykonaj wszystkie działania, a następnie zatwierdź je, używając jedynie operacji, które nie zgłaszają błędów", aby uniknąć zmiany wewnętrznego stanu, dopóki nie upewnisz się, że wszystkie operacje zakończą się powodzeniem. Takie programowanie transakcyjne jest bardziej przejrzyste i bardziej bezpieczne nawet wtedy, kiedy używasz kodów błędów. Ile razy widziałeś już funkcje (oczywiście ponownie chodzi tu o cudze funkcje, nie Twoje), w których jedna ze ścieżek prowadzących do szybkiego zwrócenia wyniku powoduje zmianę stanu obiektu, ponieważ zmiana wystąpiła przed późniejszą nieudaną operacją?
- Stosuj zasadę „jedna klasa (lub funkcja), jedno zadanie". Funkcje, które wykonują wiele zadań jednocześnie, takie jak `Stack::Pop` lub `EvaluateSalaryAndReturnName` opisane w zagadnieniach 10. i 18. w książce *Exceptional C++*[3] [Sutter00], rzadko oferują w pełni bezpieczną obsługę wyjątków. Wiele problemów związanych z bezpieczną obsługą wyjątków można uprościć lub wyeliminować bez długiego zastanawiania się, stosując się do zasady „jedna funkcja, jedno zadanie". Wskazówka ta jest dużo starsza niż wiedza o możliwości zastosowania jej do bezpiecznej obsługi wyjątków. Pomysł ten jest praktyczny sam w sobie.

Stosowanie się do tych wskazówek przysporzy Ci samych korzyści.

Biorąc pod uwagę powyższe rozważania — w jakich sytuacjach należy używać poszczególnych gwarancji? Poniżej znajduje się krótka wskazówka, do której stosuje się biblioteka standardowa języka C++ i którą także Ty możesz zastosować z korzyścią dla jakości pisanego kodu.

[3] Wydanie polskie: *Wyjątkowy język C++. 47 łamigłówek, zadań programistycznych i rozwiązań*, WNT, 2002 — *przyp. tłum.*

Funkcja powinna zawsze być zgodna z najbardziej restrykcyjnymi gwarancjami, których stosowanie nie pociąga za sobą kosztów po stronie funkcji wywołującej, niewymagającej takich gwarancji.

Jeśli więc funkcja może zapewniać gwarancję niezawodności bez generowania kosztów po stronie funkcji wywołującej, która nie potrzebuje takiej gwarancji, powinieneś zastosować właśnie ją. Pamiętaj także, że niektóre kluczowe funkcje muszą być operacjami niezawodnymi.

Nigdy nie umożliwiaj zgłaszania wyjątku przez destruktory, funkcje zwalniające zasoby oraz funkcje zamieniające obiekty. W przeciwnym razie często niemożliwe jest wykonanie porządkowania w sposób niezawodny i bezpieczny.

Jeśli funkcja może zapewniać tylko gwarancję mocną bez generowania kosztów po stronie niektórych użytkowników, powinieneś ją zastosować. Zauważ, że funkcja `vector::insert` jest przykładem funkcji, która nie zapewnia gwarancji mocnej, ponieważ wymuszałoby to tworzenie kompletnej kopii zawartości wektora przy wstawianiu każdego elementu, a nie wszyscy programiści dbają o gwarancję mocną do tego stopnia, że byliby gotowi pogodzić się z tak dużym narzutem. Ci, którym na tym zależy, mogą samodzielnie opakować funkcję `vector::insert` w gwarancję mocną. Jest to bardzo proste — wystarczy skopiować wektor, wstawić nowy element do kopii, a kiedy operacja ta się powiedzie, zamienić oryginalny wektor na kopię i gotowe.

W przeciwnym razie funkcja powinna zapewniać gwarancję słabą.

Więcej informacji o tych zagadnieniach, między innymi o niezgłaszającej wyjątków wersji funkcji `swap` lub o tym, dlaczego destruktory nie zgłaszają wyjątków, znajdziesz w książkach *Exceptional C++*[4] [Sutter00] oraz *More Exceptional C++*[5] [Sutter02].

Zagadnienie 13. Specyfikacja wyjątków z praktycznego punktu widzenia **Stopień trudności: 6**

Teraz, kiedy społeczność programistów posiada już pewne doświadczenia ze specyfikacją wyjątków, możemy podsumować, gdzie i jak należy ją dodawać, aby osiągnąć najlepsze efekty. To zagadnienie dotyczy użyteczności specyfikacji wyjątków (lub jej braku), a także stopnia jej zależności od kompilatora.

Pytania do magistra

1. Co się dzieje, kiedy zgłaszany jest wyjątek niezgodny ze specyfikacją? Dlaczego? Opisz podstawowe przyczyny istnienia tej cechy w języku C++.

2. Jakie wyjątki może zgłaszać każda z poniższych funkcji?

[4] Wydanie polskie: *Wyjątkowy język C++. 47 łamigłówek, zadań programistycznych i rozwiązań*, WNT, 2002 — *przyp. tłum.*

[5] Wydanie polskie: *Wyjątkowy język C++. 40 nowych łamigłówek, zadań programistycznych i rozwiązań*, Helion, 2005 — *przyp. tłum.*

```
int Func();
int Gunc() throw();
int Hunc() throw(A, B);
```

Pytania do profesora

3. Czy specyfikacja wyjątków stanowi o typie funkcji? Wyjaśnij.

4. Czym są specyfikacje wyjątków i co robią? Wyjaśnij szczegółowo.

5. Kiedy warto dodać do funkcji specyfikację wyjątków? Co powoduje, że decydujesz się dodać tę właściwość, a co nie?

Rozwiązanie

Prace nad nowym standardem języka C++, C++0x są dobrą okazją do analizy tego, czego nauczyliśmy się na podstawie doświadczeń z obecnym standardem [C++03]. Znacząca większość standardowych właściwości języka C++ jest przydatna i to właśnie o nich mówi się najwięcej, ponieważ nie ma sensu rozwodzić się nad mniej istotnymi cechami. Te słabsze i mniej użyteczne właściwości są przeważnie ignorowane i zanikają z braku użytkowników, aż wiele osób zapomni nawet o ich istnieniu (nie zawsze jest to złe). Dlatego możesz znaleźć stosunkowo mało artykułów na temat mniej przydatnych właściwości, takich jak `valarray`, `bitset`, ustawienia lokalne czy dozwolone wyrażenie `5[a]` (chociaż to ostatnie pojawia się w pewnej odmianie w jednym z kolejnych zagadnień). To samo dotyczy, o czym się przekonasz, specyfikacji wyjątków.

Przyjrzyjmy się teraz bliżej dotychczasowym doświadczeniom ze specyfikacją wyjątków w języku C++.

Naruszanie specyfikacji

1. ***Co się dzieje, kiedy zgłaszany jest wyjątek niezgodny ze specyfikacją? Dlaczego? Opisz podstawowe przyczyny istnienia tej cechy w języku C++.***

Celem specyfikacji wyjątków jest umożliwienie w czasie wykonywania programu przeprowadzenia sprawdzianu, który pozwala zagwarantować, że funkcja zgłasza jedynie wyjątki określonego typu lub nie zgłasza żadnych wyjątków. Na przykład specyfikacja wyjątków funkcji przedstawionej poniżej gwarantuje, że funkcja `f` zgłasza jedynie wyjątki typu `A` i `B`:

```
int f() throw( A, B );
```

Jeśli zostanie zgłoszony wyjątek spoza tej listy, zostanie wywołana funkcja `unexpected`. Poniżej przedstawiony jest prosty przykład:

```
// Przykład 13.1
//
int f() throw( A, B ) {      // A i B nie są związane z C
   throw C();                // Powoduje wywołanie funkcji unexpected
}
```

Za pomocą standardowej funkcji `set_unexpected` możesz zarejestrować własną funkcję do obsługi wyjątków nieoczekiwanych. Taka funkcja nie może przyjmować żadnych argumentów i musi zwracać typ `void`:

```
void MyUnexpectedHandler() { /*...*/ }
std::set_unexpected( &MyUnexpectedHandler; );
```

Pozostaje jednak pytanie, co powinna robić funkcja do obsługi nieoczekiwanego wyjątku? Na pewno nie może zwracać sterowania za pomocą zwykłej instrukcji `return`. Możliwe są za to dwie opcje:

- Funkcja ta może przekształcić wyjątek na inny, dopuszczony przez specyfikację wyjątków, zgłaszając własny wyjątek, zgodny ze specyfikacją. Rozwijanie stosu jest następnie kontynuowane od miejsca, w którym zostało zatrzymane.
- Funkcja może też wywołać funkcję `terminate`, która kończy działanie programu. Można także zastąpić samą funkcję `terminate`, jednak tylko inną funkcją, która także kończy działanie programu.

Dotychczasowe doświadczenia

Łatwo zrozumieć przyczyny istnienia specyfikacji wyjątków. W programie w języku C++, jeśli nie jest powiedziane inaczej, dowolna funkcja może zgłosić wyjątek dowolnego typu. Przyjrzyj się funkcji, którą nazwałem `Func` (ponieważ nazwa `f` jest straszliwie nadużywana).

2. ***Jakie wyjątki może zgłaszać każda z poniższych funkcji?***

```
// Przykład 13.2(a)
//
int Func();          // Może zgłosić dowolny wyjątek
```

Domyślnie funkcja `Func` może zgłosić dowolny wyjątek, jak jest to napisane w komentarzu. Często wiadomo, jakie wyjątki może zgłaszać dana funkcja. Wtedy oczywiście chcemy przekazać kompilatorowi i innym programistom informacje, które pozwolą ograniczyć typy wyjątków wydostające się z funkcji. Na przykład:

```
// Przykład 13.2(b)
//
int Gunc() throw();          // Nie zgłasza żadnych wyjątków
int Hunc() throw( A, B ); // Może zgłaszać jedynie wyjątki typu A i B
```

W tych przypadkach specyfikacja wyjątków funkcji pozwala określić, jakie typy wyjątków mogą zgłaszać funkcje `Gunc` i `Hunc`. Komentarze w prosty sposób opisują, co wynika z powyższych specyfikacji. Niedługo wrócimy do tego „prosto”, ponieważ okazuje się, że bliskość rzeczywistości jest zwodnicza.

Można się intuicyjnie spodziewać, że opisanie wyjątków, które może zgłaszać dana funkcja, jest korzystne, że im więcej informacji, tym lepiej. Jednak nie zawsze jest to prawdą, a diabeł tkwi w szczegółach. Chociaż same założenia są poprawne, sposób specyfikacji tej właściwości w języku C++ powoduje, że specyfikacja wyjątków nie zawsze jest użyteczna i często okazuje się szkodliwa.

Problem pierwszy — „system rozmywania typów”

3. ***Czy specyfikacja wyjątków jest częścią typu funkcji? Wyjaśnij.***

John Spicer, chluba Edison Design Group oraz autor dużych fragmentów rozdziału dotyczącego szablonów w standardzie języka C++, nazwał podobno specyfikację wyjątków języka C++ „systemem rozmywania typów”. Jedną z najistotniejszych cech języka C++ jest ścisła kontrola typów i jest to bardzo korzystne. Dlaczego mielibyśmy nazywać specyfikację wyjątków „systemem rozmywania typów”, zamiast częścią systemu kontroli typów?

Istnieją dwie proste przyczyny:

- specyfikacja wyjątków nie określa typu funkcji,
- oprócz sytuacji, w których określa.

Zastanów się najpierw nad przykładem, w którym specyfikacja wyjątków nie określa typu funkcji. Przyjrzyj się krytycznie poniższemu fragmentowi kodu:

```
// Przykład 13.3(a). Nie możesz użyć specyfikacji wyjątków w definicji typu
//
void f() throw(A, B);
typedef void (*PF)() throw(A, B); // Błąd składni
PF pf = f;                        // Z powodu błędu program nie dojdzie do tego miejsca
```

Specyfikacja wyjątków w definicji typu jest niedozwolona. Język C++ nie pozwala na kompilację powyższego kodu, dlatego specyfikacja wyjątków nie może określać typu funkcji… przynajmniej nie w kontekście definicji typu. Jednak w innych przypadkach specyfikacja wyjątków określa typ funkcji, na przykład wtedy, kiedy napiszesz tę samą deklarację funkcji bez słowa kluczowego `typedef`:

```
// Przykład 13.3(b). Możesz jednak, jeśli pominiesz słowo kluczowe typedef
//
void f() throw(A, B);
void (*pf)() throw(A, B);   // Poprawne
pf = f;                     // Poprawne
```

Możesz tak przypisać wskaźnik do funkcji, o ile specyfikacja wyjątków obiektu docelowego nie jest bardziej restrykcyjna niż specyfikacja obiektu źródłowego:

```
// Przykład 13.3(c). Także koszerne, z niską zawartością cukru i bez tłuszczu.
//
void f() throw(A,B);
void (*pf)() throw(A,B,C);   // Poprawne
pf = f;                      // Poprawne, typ pf jest mniej restrykcyjny
```

Specyfikacja wyjątków określa także typy funkcji wirtualnych, kiedy próbujesz je przesłonić:

```
// Przykład 13.3(d). Specyfikacja wyjątków wpływa na funkcje wirtualne
//
class C {
   virtual void f() throw(A,B);   // Taka sama specyfikacja wyjątków
};
```

```
class D : C {
   void f();                          // Błąd, w tym miejscu specyfikacja ma znaczenie
};
```

Tak więc pierwszy problem związany ze specyfikacją wyjątków we współczesnym języku C++ polega na tym, że stanowią one system rozmywania typów, który działa według innych reguł niż pozostała część systemu kontroli typów.

Problem drugi — (nie)zrozumienie

Drugi problem wiąże się z wiedzą o działaniu specyfikacji wyjątków. Jak zauważyło to wiele poważanych osób, włączając w to autorów specyfikacji wyjątków biblioteki Boost [BoostES], programiści zwykle używają specyfikacji wyjątków w taki sposób, jakby działała ona według reguł, które pasują programistom, a nie tak, jak rzeczywiście działa.

Wynika z tego poniższe pytanie:

4. ***Czym są specyfikacje wyjątków i co robią? Wyjaśnij szczegółowo.***

Wiele osób uważa, że specyfikacja wyjątków ma następujące właściwości:

- gwarantuje, że funkcja będzie zgłaszać jedynie wyjątki wyszczególnione w specyfikacji (często żadne);
- umożliwia kompilatorowi optymalizację opartą na informacji, że zgłaszane będą tylko wyjątki wyszczególnione w specyfikacji (często żadne).

Oczekiwania te ponownie są tylko pozornie zbliżone do rzeczywistości. Przyjrzyj się ponownie fragmentowi kodu z przykładu 13.2(b):

```
// Przykład 13.2(b). Powtórka i dwa potencjalne kłamstewka
//
int Gunc() throw();          // Nie zgłasza żadnych wyjątków              ← ?
int Hunc() throw(A,B);       // Może zgłaszać jedynie wyjątki typu A i typu B ← ?
```

Czy komentarze te są poprawne? Nie do końca. Funkcja `Gunc` może zgłosić wyjątek, a funkcja `Hunc` może zgłosić wyjątek innego typu niż `A` lub `B`! Kompilator może jedynie zagwarantować, że bezlitośnie potraktuje takie funkcje, jeśli zrobią coś takiego… przy czym najczęściej równie bezlitośnie potraktuje także cały program.

Ponieważ funkcje `Gunc` i `Hunc` mogą w rzeczywistości zgłaszać wyjątki, których zgłaszać nie powinny, kompilator nie tylko nie może założyć, że taka sytuacja się nie zdarzy, ale musi także pełnić rolę ochroniarza, który dba o to, aby przechwycić wszystkie niepoprawnie zgłoszone wyjątki. Jeśli tak się stanie, musi zostać wywołana funkcja `unexpected`. Najczęściej przerywa ona działanie programu. Dlaczego? Ponieważ istnieją tylko dwa sposoby zakończenia funkcji `unexpected`, z których żaden nie wiąże się z wywołaniem instrukcji `return`:

- Możliwe jest zgłoszenie innego wyjątku znajdującego się w specyfikacji wyjątków. Jeśli tak się stanie, wyjątek przekazywany jest tak jak w normalnej sytuacji. Pamiętaj jednak, że funkcja `unexpected` jest globalna — w całym

programie istnieje tylko jedna jej wersja. Jest bardzo mało prawdopodobne, aby taka funkcja globalna wykonywała odpowiednie działania we wszystkich przypadkach, w wyniku czego program przechodzi do funkcji `terminate`, nie przechodzi przez blok `catch` i kończy działanie.

- Zgłosić (ten sam lub inny) wyjątek, którego także nie ma w specyfikacji wyjątków. Jeśli oryginalna funkcja umożliwia zgłaszanie wyjątków typu `bad_exception`, wtedy przekazany zostanie wyjątek właśnie tego typu. Jeśli nie, program przechodzi do funkcji `terminate`, nie przechodzi przez blok `catch`…

Ponieważ naruszenie specyfikacji wyjątków przeważnie wiąże się z zakończeniem działania programu, usprawiedliwione jest nazwanie tej sytuacji „bezlitosnym potraktowaniem programu".

Wcześniej miałeś okazję zobaczyć, jakie możliwości wiele osób przypisuje specyfikacji wyjątków. Poniżej znajduje się poprawiona wersja tych oczekiwań, która dokładniej przedstawia rzeczywiste możliwości [sic][6]:

- ~~Gwarantuje~~ ***Wymusza w czasie wykonania programu***, że funkcja będzie zgłaszać jedynie wyjątki wyszczególnione w specyfikacji (często żadne).
- Umożliwia ***lub uniemożliwia*** kompilatorowi optymalizację opartą na ~~informacji, że zgłaszane będą tylko wyjątki wyszczególnione w specyfikacji (często żadne)~~ ***sprawdzeniu, czy wyszczególnione w specyfikacji wyjątki rzeczywiście zostały zgłoszone***.

Aby zrozumieć, co robi kompilator, przyjrzyj się poniższemu fragmentowi kodu, który przedstawia ciało jednej z przykładowych funkcji:

```
// Przykład 13.4(a)
//
int Hunc() throw(A,B) {
   return Junc();
}
```

Kompilator musi wygenerować kod taki jak poniżej. Szybkość działania takiego kodu jest zwykle porównywalna z kodem, który napisałbyś samodzielnie (oprócz czasu spędzonego na pisaniu):

```
// Przykład 13.4(b). Wersja przykładu 13.4(a) rozwinięta przez kompilator
//
int Hunc()
try {
   return Junc();
}
catch( A ) {
   throw;
}
catch( B ) {
   throw;
```

[6] Tak, to taki żart.

```
}
catch( ... ) {
   std::unexpected(); // Nie wywołuje return! Jeśli masz szczęście, zgłasza wyjątek typu A
                         lub typu B
}
```

Teraz możesz łatwiej zrozumieć, dlaczego zamiast optymalizacji kodu opartej na założeniu, że zgłaszane są jedynie określone typy wyjątków, dzieje się coś wręcz przeciwnego. Kompilator musi wykonać *więcej zadań, aby wymusić* zgłaszanie jedynie określonych wyjątków w czasie wykonywania programu.

Zakres specyfikacji wyjątków

Większość osób jest zaskoczona, kiedy dowiaduje się, że specyfikacja wyjątków może wiązać się z większymi kosztami wykonywania programu. Jedna z przyczyn takiego stanu rzeczy została właśnie wystarczająco obszernie wyjaśniona. Specyfikacja wyjątków powoduje narzut związany z niejawnym generowaniem bloków `try` i `catch`, chociaż w przypadku wydajnych kompilatorów może być on niezauważalny.

Istnieją przynajmniej dwie inne przyczyny, dla których specyfikacja wyjątków może negatywnie wpływać na wydajność programu:

- Niektóre kompilatory automatycznie przekształcają funkcje inline ze specyfikacjami wyjątków na zwykłe funkcje, stosują także inne heurystyki, na przykład przekształcanie funkcji inline, które zawierają więcej niż określoną liczbę zagnieżdżonych wyrażeń lub pętle w dowolnej postaci.
- Niektóre kompilatory w ogóle nie optymalizują kodu związanego z wyjątkami i dodają generowane bloki `try` i `catch` nawet wtedy, kiedy funkcja na pewno nie będzie zgłaszać wyjątków.

Pomijając wydajność czasu wykonania, specyfikacja wyjątków może wydłużyć czas tworzenia programu, ponieważ zwiększa zależności. Na przykład usunięcie typu ze specyfikacji wyjątków funkcji wirtualnej w klasie bazowej to szybki i prosty sposób na problemy w wielu klasach pochodnych (jeśli tylko szukasz takiego sposobu). Możesz go wypróbować w pracy w piątkowe popołudnie i przeprowadzić ankietę na temat liczby listów elektronicznych z pogróżkami, które będą czekać na Ciebie w poniedziałek.

Dlatego kolejne pytanie brzmi:

5. ***Kiedy warto dodać do funkcji specyfikację wyjątków? Co powoduje, że decydujesz się dodać tę właściwość, a co nie?***

Poniżej znajdują się prawdopodobnie najlepsze wskazówki sformułowane na podstawie dotychczasowych doświadczeń ze specyfikacją wyjątków:

Wniosek 1: Nigdy nie pisz specyfikacji wyjątków.

Wniosek 2: Możesz ewentualnie pisać puste specyfikacje wyjątków, ale odradzałbym nawet to.

Doświadczenia twórców biblioteki Boost ze specyfikacjami niezgłaszającymi wyjątków dla funkcji, które nie są inline, to jedyny przypadek, w którym specyfikacje wyjątków „mogą na niektórych kompilatorach przynosić pewne korzyści". Stwierdzenie to nie jest może olśniewające, ale stanowi użyteczną wskazówkę, jeśli chcesz tworzyć przenośny kod, który można skompilować na więcej niż jednym kompilatorze.

W praktyce jest jeszcze gorzej, ponieważ okazuje się, że popularne implementacje różnią się w sposobie obsługi specyfikacji wyjątków. Przynajmniej jeden z kompilatorów języka C++ (wszystkie wersje kompilatora Microsoftu do czasu wydania książki, czyli do wersji 7.1 z roku 2003) przetwarzają specyfikacje wyjątków, ale ich nie wymuszają, przez co redukują je do specjalnych komentarzy. Czekaj, to jeszcze nie koniec. Jednocześnie istnieją poprawne optymalizacje, które kompilator wykonuje poza funkcją, *oparte na* wymuszaniu specyfikacji wyjątków wewnątrz każdej funkcji i kompilator Microsoftu w wersjach 7.x wykonuje je. Pomysł polega na tym, że jeśli funkcja próbuje zgłosić wyjątek, którego nie powinna, wtedy wewnętrzna funkcja zatrzymuje program i sterowanie nigdy nie wraca do funkcji wywołującej. Ponieważ jednak sterowanie wraca do funkcji wywołującej, w kodzie wygenerowanym w miejscu wywołania można założyć, że żaden wyjątek nie został zgłoszony, co pozwala wyeliminować zewnętrzne bloki `try` i `catch`.

W przypadku kompilatorów, które nie wymuszają przestrzegania specyfikacji wyjątków, ale *polegają* na niej, znaczenie pustej specyfikacji `throw()` zmienia się. Zamiast „sprawdź listę, zatrzymaj mnie, jeśli zgłoszę niedozwolony wyjątek" oznacza to teraz: „zaufaj mi, załóż, że nigdy nie zgłoszę wyjątku i wykonaj optymalizację". Bądź więc ostrożny. Jeśli zdecydujesz się użyć pustej specyfikacji wyjątków, przeczytaj dokumentację kompilatora, którego używasz, aby sprawdzić, w jaki sposób kompilator ją obsługuje. Możesz być poważnie zaskoczony. Bądź ostrożny, kieruj uważnie.

Podsumowanie

W skrócie — nie musisz się martwić o specyfikacje wyjątków. Nawet specjaliści tego nie robią.

Poniżej, w nieco mniejszym skrócie, przedstawione są najważniejsze zagadnienia:

- Specyfikacja wyjątków może powodować nieoczekiwane spadki wydajności, wynikające na przykład z przekształcania funkcji inline ze specyfikacjami wyjątków na zwykłe funkcje.
- Błąd czasu wykonania `unexpected` nie zawsze jest tym, czego chciałbyś w przypadku typów błędów przechwytywanych za pomocą specyfikacji wyjątków.
- Nie możesz utworzyć użytecznej specyfikacji wyjątków dla szablonów funkcji, ponieważ zwykle nie wiesz, jakie wyjątki mogą zgłaszać typy, na których funkcja ta operuje.

Kiedy przedstawiałem te zagadnienia jako część większego wykładu na niedawnej konferencji, zapytałem, kto z około 100 osób każdorazowo używa specyfikacji wyjątków. Około połowa podniosła ręce. Wtedy jakiś żartowniś z tyłu sali powiedział (całkiem

słusznie), że powinienem także zapytać, ile z tych osób w pewnym momencie rezygnuje ze specyfikacji wyjątków. Zapytałem. Zgłosiło się mniej więcej tyle samo osób, co poprzednio. Ta sytuacja mówi sama za siebie. Czołowi projektanci bibliotek w korporacji Boost mają podobne doświadczenia i dlatego ich strategia dotycząca pisania specyfikacji wyjątków sprowadza się do prostego „nie rób tego" [BoostES].

To prawda, że wiele osób o dobrych intencjach domagało się włączenia do języka specyfikacji wyjątków i dlatego znalazły się one w standardzie. Przypomina mi to sympatyczny wierszyk, który pierwszy raz przeczytałem około 15 lat temu, kiedy pojawił się w listach elektronicznych w czasie ferii zimowych. Słowa śpiewane są na melodię „Wśród nocnej ciszy", a wierszyk zatytułowany był „Wśród nocnej implementacji" lub „Wśród nocnego kryzysu". Opowiada on o doświadczonym programiście, który ślęczy po nocach w czasie ferii, aby zdążyć przed terminem. W tym celu dokonuje wielu cudów, które pozwalają utworzyć działający system doskonale spełniający wymagania. Niestety, na koniec zostaje na lodzie, o czym mówią cztery ostatnie linijki wierszyka:

Program gotowy, skończone testy,

nawet poprawki klienta są w nim.

Użytkownik jednak prycha i uśmiecha się krzywo z cicha:

„O to prosiłem, lecz nie tego chcę, o to prosiłem, lecz nie tego chcę".

To samo można powiedzieć, przyglądając się dotychczasowym doświadczeniom ze specyfikacją wyjątków. Swego czasu właściwość ta przedstawiała się obiecująco… i jest dokładnie tym, czego niektóre osoby oczekiwały.

Uważaj, czego sobie życzysz, bo może się spełnić.