

## IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

## KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

## TWÓJ KOSZYK

DODAJ DO KOSZYKA

## CENNIK I INFORMACJE

ZAMÓW INFORMACJE  
O NOWOŚCIACH

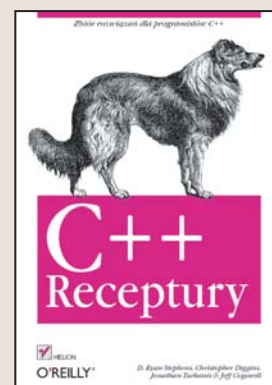
ZAMÓW CENNIK

## CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

# C++. Receptury

Autorzy: D. Ryan Stephens, Christopher Diggins, Jonathan Turkanis, Jeff Cogswell  
Tłumaczenie: Przemysław Szeremiota  
ISBN: 83-246-0374-3  
Tytuł oryginału: [C++ Cookbook](#)  
Format: B5, stron: 560



### Zbiór rozwiązań dla programistów C++

- Operacje na klasach i obiektach
- Obsługa błędów i wyjątków
- Przetwarzanie dokumentów XML

C++ jest jednym z najpopularniejszych języków programowania. Jego implementacje dostępne są praktycznie dla wszystkich platform systemowych. Programiści posługujący się językiem C++ napisali setki tysięcy aplikacji. Codziennie jednak stają przed koniecznością rozwiązywania podobnych problemów, związanych na przykład z przetwarzaniem dat, manipulowaniem ciągami tekstowymi czy stosowaniem standardowych kontenerów. W takich sytuacjach na pewno zadają sobie pytanie – czy warto ponownie wymyślać koło? Przecież gotowe rozwiązania znacznie przyspieszyłyby pracę.

Książka „C++. Receptury” może pełnić funkcję skarbnicy porad dla programistów. Znajdziesz w niej rozwiązania problemów, z jakimi spotykasz się w codziennej pracy. Każda analiza uzupełniona jest przykładowym kodem źródłowym, który można wykorzystać we własnych projektach. Autorzy położyli szczególny nacisk na prostotę i przenośność kodu oraz wykorzystanie, tam gdzie to możliwe, biblioteki standardowej.

- Kompilowanie aplikacji
- Właściwa organizacja kodu źródłowego
- Operacje na liczbach, tekstach i danych
- Stosowanie kontenerów
- Programowanie obiektowe
- Przetwarzanie plików
- Strumienie
- Operacje matematyczne i statystyczne
- Wielowątkowość i biblioteka Boost
- Praca z dokumentami XML

**Przyspiesz pracę nad aplikacją, stosując gotowe i sprawdzone rozwiązania**



---

# Spis treści

<b>Wstęp .....</b>	<b>9</b>
<b>1. Tworzenie aplikacji w języku C++ .....</b>	<b>15</b>
1.0. Wprowadzenie do systemów kompilacji	15
1.1. Pobieranie i instalowanie GCC	28
1.2. Kompilowanie programu „Hello, World!” w wierszu poleceń	31
1.3. Kompilowanie biblioteki statycznej w wierszu poleceń	36
1.4. Kompilowanie biblioteki dynamicznej w wierszu poleceń	38
1.5. Kompilowanie aplikacji wieloskładnikowej w wierszu poleceń	45
1.6. Instalowanie pakietu Boost.Build	50
1.7. Kompilowanie programu „Hello, World!” za pomocą Boost.Build	52
1.8. Kompilowanie biblioteki statycznej za pomocą Boost.Build	56
1.9. Kompilowanie biblioteki dynamicznej za pomocą Boost.Build	57
1.10. Kompilowanie aplikacji wieloskładnikowej za pomocą Boost.Build	58
1.11. Kompilowanie biblioteki statycznej w IDE	61
1.12. Kompilowanie biblioteki dynamicznej w IDE	64
1.13. Kompilowanie aplikacji wieloskładnikowej w IDE	68
1.14. Pobieranie i instalowanie GNU make	73
1.15. Kompilowanie programu „Hello, World!” za pomocą GNU make	75
1.16. Kompilowanie biblioteki statycznej za pomocą GNU make	82
1.17. Kompilowanie biblioteki dynamicznej za pomocą GNU make	87
1.18. Kompilowanie aplikacji wieloskładnikowej za pomocą GNU make	88
1.19. Definiowanie symboli (makrodefinicji)	92
1.20. Ustalanie opcji wiersza polecenia w IDE	94
1.21. Kompilacja próbna	95
1.22. Kompilacja ostateczna	98
1.23. Wybieranie wersji biblioteki wykonawczej	101
1.24. Wymuszanie zgodności ze standardem języka C++	104
1.25. Automatyzacja konsolidacji pliku źródłowego z wybraną biblioteką	107
1.26. Korzystanie z szablonów eksportowanych	109

<b>2. Organizacja kodu .....</b>	<b>113</b>
2.0. Wprowadzenie	113
2.1. Gwarantowanie jednokrotnego włączenia pliku nagłówkowego	114
2.2. Gwarantowanie obecności jednego egzemplarza zmiennej dla wielu plików źródłowych	116
2.3. Ograniczanie włączania nagłówków za pomocą deklaracji zapowiadających	117
2.4. Unikanie kolizji nazw za pomocą przestrzeni nazw	119
2.5. Włączanie pliku funkcji inline	125
<b>3. Liczby .....</b>	<b>127</b>
3.0. Wprowadzenie	127
3.1. Konwersja ciągu na typ liczbowy	127
3.2. Konwersja liczb na ciągi	130
3.3. Sprawdzanie, czy ciąg zawiera poprawną liczbę	133
3.4. Porównywanie wartości zmiennoprzecinkowych w zadanym zakresie dokładności	135
3.5. Przetwarzanie ciągu zawierającego liczbę w zapisie naukowym	137
3.6. Konwersja pomiędzy typami liczbowymi	139
3.7. Określanie granicznych wartości typów liczbowych	141
<b>4. Ciągi i teksty .....</b>	<b>145</b>
4.0. Wprowadzenie	145
4.1. Dopełnianie ciągu	146
4.2. Przycinanie ciągu	147
4.3. Zapisywanie ciągów w sekwencji	152
4.4. Określanie długości ciągu	155
4.5. Odwracanie ciągu	157
4.6. Podział ciągu	158
4.7. Wyodrębnianie elementów leksykalnych	160
4.8. Scalanie sekwencji ciągów	163
4.9. Wyszukiwanie w ciągach	165
4.10. Szukanie n-tego wystąpienia podciągu	168
4.11. Usuwanie podciągu z ciągu	169
4.12. Zmiana wielkości liter w ciągu	171
4.13. Porównywanie ciągów bez uwzględniania wielkości liter	173
4.14. Wyszukiwanie w ciągu bez uwzględniania wielkości liter	175
4.15. Zamiana tabulacji na spacje w pliku tekstowym	177
4.16. Zawijanie wierszy w pliku tekstowym	179
4.17. Zliczanie znaków, słów i wierszy w pliku tekstowym	181
4.18. Zliczanie wystąpień poszczególnych słów w pliku tekstowym	184
4.19. Ustawianie marginesów w pliku tekstowym	186
4.20. Justowanie tekstu w pliku	189

4.21. Eliminowanie nadmiarowych znaków odstępu w pliku tekstowym	191
4.22. Autokorekta tekstu przy zmianach bufora	192
4.23. Wczytywanie danych z pliku wartości rozdzielanych przecinkami	195
4.24. Podział ciągu na podstawie wyrażeń regularnych	197
<b>5. Daty i godziny .....</b>	<b>199</b>
5.0. Wprowadzenie	199
5.1. Odczytywanie bieżącej daty i godziny	199
5.2. Formatowanie ciągów reprezentujących daty i godziny	202
5.3. Arytmetyka dat i godzin	204
5.4. Konwersja pomiędzy strefami czasowymi	206
5.5. Określanie numeru dnia w roku	207
5.6. Definiowanie typów wartości ograniczonych do zakresu	209
<b>6. Gospodarowanie danymi — kontenery .....</b>	<b>213</b>
6.0. Wprowadzenie	213
6.1. Kontenery zamiast tablic	214
6.2. Efektywne stosowanie wektorów	218
6.3. Kopiowanie wektora	222
6.4. Przechowywanie wskaźników w wektorze	224
6.5. Przechowywanie obiektów na liście	225
6.6. Kojarzenie danych z ciągami znaków	230
6.7. Kontenery haszowane	235
6.8. Sekwencje uporządkowane	240
6.9. Kontenery w kontenerach	243
<b>7. Algorytmy .....</b>	<b>247</b>
7.0. Wprowadzenie	247
7.1. Przeglądanie zawartości kontenera	248
7.2. Usuwanie obiektów z kontenera	254
7.3. Tworzenie sekwencji pseudolosowych	257
7.4. Porównywanie zakresów	259
7.5. Scalanie danych	262
7.6. Sortowanie zakresu elementów	266
7.7. Partycjonowanie zakresu	268
7.8. Przetwarzanie sekwencji za pomocą operacji dla zbiorów	270
7.9. Przekształcanie elementów sekwencji	273
7.10. Własne algorytmy uogólnione	275
7.11. Wypisywanie elementów zakresu do strumienia	278

<b>8. Klasy .....</b>	<b>283</b>
8.0. Wprowadzenie	283
8.1. Inicjalizowanie składowych klas	284
8.2. Tworzenie obiektów w funkcjach (wzorzec Factory)	287
8.3. Konstruktory i destruktory w służbie zarządzania zasobami (RAII)	289
8.4. Automatyczne dodawanie nowych egzemplarzy klasy do kontenera	291
8.5. Jedna kopia składowej klasy	293
8.6. Określanie dynamicznego typu obiektu	295
8.7. Wykrywanie zależności pomiędzy klasami różnych obiektów	297
8.8. Nadawanie identyfikatorów egzemplarzom klas	298
8.9. Tworzenie klasy-jedynaka	301
8.10. Tworzenie interfejsu z abstrakcyjną klasą bazową	303
8.11. Pisanie szablonu klasy	307
8.12. Pisanie szablonu funkcji	312
8.13. Przeciążanie operatorów inkrementacji i dekrementacji	314
8.14. Przeciążanie operatorów arytmetycznych i operatorów przypisania pod kątem intuicyjności zachowania obiektów klasy	317
8.15. Wywoływanie funkcji wirtualnej klasy bazowej	323
<b>9. Wyjątki i bezpieczeństwo .....</b>	<b>325</b>
9.0. Wprowadzenie	325
9.1. Tworzenie klasy wyjątku	325
9.2. Uodpornianie konstruktora klasy	329
9.3. Uodpornianie listy inicjalizacyjnej konstruktora	332
9.4. Uodpornianie metod klasy	335
9.5. Bezpieczne kopiowanie obiektu	339
<b>10. Strumienie i pliki .....</b>	<b>345</b>
10.0. Wprowadzenie	345
10.1. Wyrównywanie tekstu w kolumnach	346
10.2. Formatowanie wartości zmiennoprzecinkowych	350
10.3. Pisanie własnego manipulatora strumienia	353
10.4. Adaptacja klasy do zapisu obiektów do strumienia	356
10.5. Adaptacja klasy do odczytu obiektów ze strumienia	359
10.6. Pozyskiwanie informacji o pliku	361
10.7. Kopiowanie pliku	363
10.8. Usuwanie i zmiana nazwy pliku	366
10.9. Tymczasowe nazwy plików i pliki tymczasowe	368
10.10. Tworzenie katalogu	370
10.11. Usuwanie katalogu	372
10.12. Przeglądanie zawartości katalogu	374

10.13. Wyłuskiwanie ciągu rozszerzenia pliku	376
10.14. Wyłuskiwanie nazwy pliku z ciągu ścieżki dostępu	377
10.15. Wyłuskiwanie ścieżki dostępu	379
10.16. Zmiana rozszerzenia nazwy pliku	380
10.17. Montowanie ścieżek dostępu	381
<b>11. Matematyka, statystyka .....</b>	<b>385</b>
11.0. Wprowadzenie	385
11.1. Określanie liczby elementów w kontenerze	386
11.2. Wyszukiwanie największej bądź najmniejszej wartości w kontenerze	387
11.3. Obliczanie sumy i średniej wartości elementów kontenera	390
11.4. Filtrowanie wartości spoza zadanego zakresu	392
11.5. Obliczanie wariancji, odchylenia standardowego i innych wskaźników statystycznych	394
11.6. Generowanie liczb losowych	397
11.7. Inicjalizacja kontenera liczbami losowymi	399
11.8. Wektory liczb o dynamicznych rozmiarach	400
11.9. Wektory liczb o stałych rozmiarach	401
11.10. Obliczanie iloczynu skalarnego	404
11.11. Obliczanie długości wektora	405
11.12. Obliczanie odległości pomiędzy wektorami	406
11.13. Implementowanie iteratora kroczącego	407
11.14. Macierze o dynamicznych rozmiarach	411
11.15. Macierze o stałych rozmiarach	414
11.16. Mnożenie macierzy	416
11.17. Szybka transformata Fouriera	418
11.18. Współrzędne biegunowe	420
11.19. Arytmetyka zbiorów bitowych	421
11.20. Reprezentowanie wielkich liczb całkowitych	425
11.21. Liczby stałoprzecinkowe	429
<b>12. Wielowątkowość .....</b>	<b>431</b>
12.0. Wprowadzenie	431
12.1. Tworzenie wątku	432
12.2. Synchronizacja dostępu do zasobu	435
12.3. Sygnalizacja pomiędzy wątkami	443
12.4. Jednokrotna inicjalizacja wspólnych zasobów	446
12.5. Argumenty funkcji wątku	447
<b>13. Internacjonalizacja .....</b>	<b>451</b>
13.0. Wprowadzenie	451
13.1. Literały Unicode	452

13.2. Wczytywanie i wypisywanie liczb	453
13.3. Wczytywanie i wypisywanie dat i godzin	457
13.4. Wczytywanie i wypisywanie wartości pieniężnych	461
13.5. Sortowanie ciągów zlokalizowanych	466
<b>14. XML .....</b>	<b>469</b>
14.0. Wprowadzenie	469
14.1. Przetwarzanie prostych dokumentów XML	470
14.2. Praca z ciągami Xerces	477
14.3. Przetwarzanie złożonych dokumentów XML	480
14.4. Manipulowanie dokumentami XML	489
14.5. Walidacja dokumentu XML względem definicji DTD	493
14.6. Walidacja dokumentu XML względem schematu	497
14.7. Przekształcanie dokumentów XML przy użyciu XSLT	501
14.8. Obliczanie wartości wyrażenia XPath	506
14.9. XML w utrwalaniu i odtwarzaniu kolekcji obiektów	512
<b>15. Różne .....</b>	<b>517</b>
15.0. Wprowadzenie	517
15.1. Wskaźniki funkcji w wywołaniach zwrotnych	517
15.2. Wskaźniki składowych klas	519
15.3. Blokowanie modyfikacji argumentu wywołania funkcji	521
15.4. Blokowanie modyfikacji obiektu w metodzie wywołanej na rzecz tego obiektu	524
15.5. Operatory niebędące metodami klasy	526
15.6. Inicjalizacja sekwencji wartościami wymienianymi po przecinkach	528
<b>Skorowidz .....</b>	<b>531</b>

---

# Wyjątki i bezpieczeństwo

## 9.0. Wprowadzenie

Dotarliśmy do receptur dotyczących obsługi wyjątków w języku C++. C++ posiada solidną obsługę wyjątków, która w połączeniu z kilkoma technikami pozwala na pisanie kodu prostego do diagnozowania i efektywnie obsługującego sytuacje wyjątkowe.

Pierwsza receptura dotyczy typowej dla C++ semantyki rzucania i przechwytywania wyjątków, a następnie sposobu pisania klasy reprezentującej wyjątek. To na początek dla tych wszystkich, którzy na obsłudze wyjątków C++ znają się mało albo wcale. Ta sama receptura opisuje też standardowe klasy wyjątków, definiowane w nagłówkach `<stdexcept>` i `<exception>`.

Pozostałe receptury ilustrują techniki optymalnego stosowania wyjątków i definiują przy okazji kilka kluczowych terminów i pojęć. Lektura wyjaśni, że rzucanie wyjątków w obliczu sytuacji nieoczekiwanych i przechwytywanie ich jedynie w celu wypisania komunikatu wcale nie stanowi o solidności oprogramowania. Efektywne stosowanie mechanizmów obsługi wyjątków dostępnych w języku C++ polega na zabezpieczeniu programu przed wyciekami zasobów przy rzucaniu wyjątków i ogólnym zadbaniu o poprawne zachowanie kodu w obliczu wyjątków. Tu można odnieść się do dwóch ukutych na tę okoliczność pojęć, a mianowicie pojęcia *zwykłej* i *silnej* gwarancji odporności na wyjątki. Receptury będą więc opisywały techniki wdrażania tych gwarancji w konstruktorach i rozmaitych metodach klas.

## 9.1. Tworzenie klasy wyjątku

### Problem

Chcemy utworzyć własną klasę wyjątku nadającego się do rzucania i przechwytywania.

### Rozwiązanie

W charakterystycznym dla wyjątków rzucaniu i przechwytywaniu może uczestniczyć dowolny typ C++, który spełnia kilka nieskomplikowanych wymagań. Wymagania te obejmują dostępność odpowiedniego konstruktora kopiującego i destruktora. Chociaż podstawowe wymagania są



proste, same wyjątki to zagadnienie dość skomplikowane, więc przy projektowaniu klasy reprezentującej sytuację wyjątkową trzeba wziąć pod uwagę szereg aspektów. Przykład prostej klasy wyjątku prezentuje listing 9.1.

Listing 9.1. Prosta klasa wyjątku

```
#include <iostream>
#include <string>

using namespace std;

class Exception {
public:
    Exception(const string& msg) : msg_(msg) {}
    ~Exception() {}

    string getMessage() const {return(msg_);}
private:
    string msg_;
};

void f() {
    throw(Exception("Oho!"));
}

int main() {
    try {
        f();
    }
    catch(Exception& e) {
        cout << "zrzuciono wyjątek: " << e.getMessage() << endl;
    }
}
```

## Analiza

Obsługa wyjątków w języku C++ angażuje trzy słowa kluczowe: `try`, `catch` i `throw`. Składnia ich stosowania prezentuje się następująco:

```
try {
    // coś, co prowokuje zrzucenie wyjątku, np:
    throw(Exception("Oho!"));
} catch(Exception& e) {
    // obsługa wyjątku e
}
```

W języku C++ (podobnie jest w językach Java i C#) wyjątek to odpowiednik listu w butelce, rzuconego w morze tuż przed przymusowym opuszczeniem szalup — rozbitkowie podejmują ten desperacki krok w nadziei, że ktoś wyłowi komunikat (w programie będzie to któryś z kolejnych wywołujących). Wyjątki stanowią alternatywę dla prostszych, klasycznych sposobów sygnalizacji błędów, np. kodów bądź komunikatów o błędzie. Semantyka stosowania wyjątków, obejmująca „próbowanie” operacji (ang. *try*), „zrzucanie” (ang. *throw*) wyjątku i jego „przechwytywanie” (ang. *catch*) różni się znacznie od semantyki innych operacji w programach C++, przez co przed przystąpieniem do opisu klasy wyjątku wypadałoby powiedzieć sobie, co oznacza ich zrzucanie i przechwytywanie.

W obliczu sytuacji wyjątkowej, kiedy trzeba powiadomić o niej wywołującego, wpychamy list do butelki i rzucamy ją w morze:

```
throw(Exception("Coś nie tak"));
```

Środowisko wykonawcze konstruuje wtedy obiekt `Exception`, a potem zaczyna zwijanie stosu aż do miejsca, w którym uda się znaleźć blok kodu chronionego, który został już rozpoczęty, ale jeszcze się nie zakończył. Jeśli nie uda się go znaleźć, co oznacza wycofanie sterowania do funkcji `main` (głównego zasięgu bieżącego wątku) i niemożność dalszego zwijania stosu, wywoływana jest specjalna funkcja `terminate`. Ale jeśli uda się znaleźć po drodze rozpoczęty blok `try`, następuje przeglądanie odpowiadających mu kolejnych klauzul `catch` w poszukiwaniu klauzuli dopasowanej do typu zrzuconego wyjątku. Może to być coś takiego:

```
catch(Exception& e) { //...
```

W tym momencie przy użyciu konstruktora kopiującego klasy `Exception` tworzony jest nowy obiekt typu `Exception` na podstawie obiektu zrzuconego — bo ten zrzucony w zasięgu `throw` jest obiektem tymczasowym. Pierwotny wyjątek jest usuwany tuż poza zasięgiem, a program podejmuje wykonanie kodu klauzuli `catch` z nowym obiektem wyjątku.

Jeśli w bloku `catch` zechcemy przekazać wyjątek dalej w górę stosu wywołań, powinniśmy zastosować instrukcję `throw` bez argumentów:

```
throw;
```

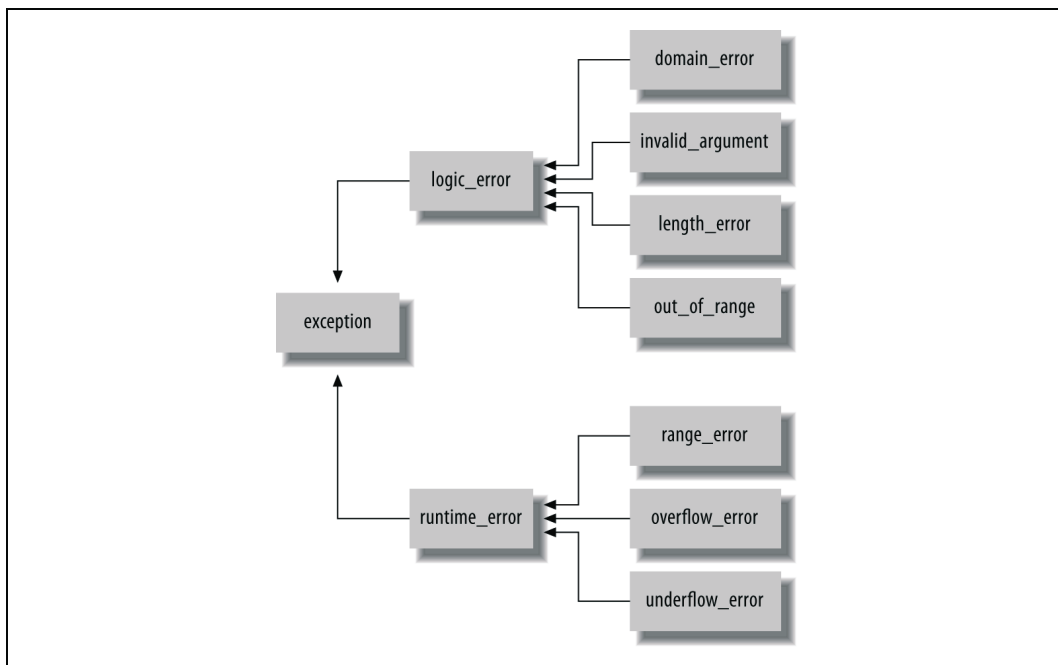
W ten sposób wymusimy dalsze zwijanie stosu aż do znalezienia następnego napoczętego bloku `try` z klauzulą `catch` dającą się dopasować do typu wyjątku. W ten sposób można obsługiwać wyjątek kaskadowo — w każdym zasięgu realizować tyle, ile można, a potem zrzucić wyjątek dalej, do następnych obsługujących.

Tak w wielkim skrócie przedstawia się droga wyjątku od zrzucenia do przechwycenia. Wyposażeni w podstawową wiedzę możemy wrócić do analizy listingu 9.1. Mamy tu tworzenie obiektu wyjątku `Exception` na podstawie ciągu znaków, przekazywanych wskaźnikiem albo obiektem `string`, a następnie zrzucenie wyjątku. Tak zdefiniowana klasa wyjątku jest wyjątkowo mało przydatna, bo stanowi w zasadzie jedynie kopertę dla najwykleszego komunikatu o błędzie. Niemal identyczny efekt osiągnęlibyśmy, zrzucając w roli wyjątku najwykleszy obiekt typu `string`:

```
try {
    throw(string("Coś nie tak!"));
} catch(string& s) {
    cout << "zrzucono wyjątek: " << s << endl;
}
```

Nie jest to najlepsza praktyka, ale ilustruje naturę wyjątków: są one obiektami niemal zupełnie dowolnych typów języka C++. Zrzucić można wartości typu `int`, `char`, `class`, `struct` itp. Lepiej jednak skorzystać z hierarchii klas wyjątków: własnej albo zapożyczonej z biblioteki standardowej.

Największą chyba zaletą stosowania specjalnej hierarchii klas wyjątków jest możliwość wyrażania charakteru sytuacji wyjątkowej za pośrednictwem typu wyjątku, a nie tylko za pośrednictwem wartości kodu błędu, wartości poziomu zagrożenia czy treści komunikatu. Takie też podejście zastosowano w implementacji standardowych klas wyjątków, definiowanych w nagłówku `<stdexcept>`. Klasą bazową całej hierarchii klas z nagłówka `<stdexcept>` jest klasa `exception`, sama definiowana w nagłówku `<exception>`. Układ hierarchii standardowych klas wyjątków prezentuje rysunek 9.1.



Rysunek 9.1. Standardowe klasy wyjątków

Każda standardowa klasa wyjątku sygnalizuje swoją nazwą kategorię sytuacji wyjątkowych, które jej obiekty mają identyfikować. Na przykład klasa `logic_error` reprezentuje okoliczności, które powinny zostać wychwycone podczas pisania, a później — przeglądu kodu, a jej podklasy doprecyzowują te okoliczności: naruszenie warunku wstępnego, przekazanie indeksu spoza zakresu, przekazanie nieodpowiedniego argumentu i tak dalej. Uzupełnieniem klasy „błędów logicznych” jest klasa błędów wykonawczych, reprezentowana przez `runtime_error` i jej pochodne. Tu zaliczymy sytuacje, których nie dało się wykluczyć w kodzie programu, jak przekroczenie zakresu czy przepełnienie bądź niedopełnienie wartości.

Klasy standardowe reprezentują dość ograniczony zbiór sytuacji wyjątkowych, więc nie zawsze można dobrać taką, która w pełni odpowiada konkretnym okolicznościom. Zwykle hierarchię tę trzeba uzupełnić o sytuacje charakterystyczne dla dziedziny zastosowań, np. `database_error`, `network_error` czy `painting_error` (odpowiednio: wyjątek komunikacji z bazą danych, wyjątek transmisji sieciowej czy wyjątek operacji odrysowywania). Zanim zagłębimy się w szczegóły technik uzupełniania hierarchii klas wyjątków, przyjrzyjmy się sposobowi działania wyjątków.

Skoro biblioteka standardowa korzysta wyłącznie ze standardowych (no proszę!) klas wyjątków, można się spodziewać, że klasy biblioteki standardowej będą w obliczu sytuacji wyjątkowych rzucać klasy z przedstawionej wcześniej hierarchii; i tak kontener `vector` (a konkretnie, jedna z jego metod) może rzucić wyjątek `out_of_range` w reakcji na indeks elementu spoza zakresu wektora:

```

std::vector<int> v;
int i = -1;

// wypełnianie v...

```

```

try {
    i = v.at(v.size());    // o jeden element za daleko
}
catch (std::out_of_range& e) {
    std::cerr << "Oho, zrzuciono wyjątek: " << e.what() << '\n';
}

```

Metoda `vector<>::at` zrzuci wyjątek `out_of_range`, kiedy otrzyma za pośrednictwem argumentu wywołania indeks wykraczający poza bieżący rozmiar kontenera, co dotyczy każdego indeksu większego od `v.size() - 1`. Wiedza ta pozwala nam na oprogramowanie najbardziej odpowiedniej obsługi tego akurat wyjątku. Jeśli jednak nie spodziewamy się konkretnego wyjątku, a przynajmniej nie mamy możliwości różnicowania ich obsługi, możemy pokusić się o przechwycenie wyjątków z całej hierarchii, posługując się klasą bazową wyjątków standardowych:

```

catch (std::exception& e) {
    std::cerr << "Nieznany bliżej wyjątek: " << e.what() << '\n';
}

```

Taka klauzula `catch` przechwyci każdy obiekt wyjątku klasy pochodnej względem klasy `exception`. Metoda `what` to wirtualna metoda klasy wyjątku, zwracająca zależny od implementacji komunikat.

Zmierzamy do zatoczenia pełnego okręgu. Cytat z listingu 9.1 uzupełniony obszernym komentarzem ma stanowić ilustrację zalet stosowania klas wyjątków. Klasy wyjątków są przydatne z dwóch względów: ze względu na możliwość różnicowania obsługi sytuacji wyjątkowych na podstawie typu obiektu wyjątku oraz ze względu na dostępność komunikatu nadającego się do zaprezentowania użytkownikowi programu. Hierarchia klas wyjątków pozwala programistom korzystającym z naszej biblioteki na pisanie kodu bezpiecznego i łatwego w diagnostyce, a dostępność komunikatu to wyraźna korzyść dla końcowych użytkowników programu, którzy zyskują więcej informacji przy zgłaszaniu problemów dostawcom oprogramowania.

Tematyka wyjątków to tematyka obszerna i złożona, a ich bezpieczne i efektywne stosowanie i obsługiwanie to chyba najtrudniejszy aspekt praktycznej inżynierii oprogramowania w ogóle, a inżynierii języka C++ w szczególności. Jak napisać konstruktor, tak aby nie dopuścić do wycieku pamięci w obliczu zrzucenia wyjątku w ciele konstruktora albo na liście inicjalizacyjnej? Co oznacza odporność na wyjątki? Odpowiedzi poznamy w kolejnych recepturach.

## 9.2. Uodpornianie konstruktora klasy

### Problem

Konstruktor ma zabezpieczać podstawowe i silne gwarancje odporności na wyjątki. Znaczenie obu tych gwarancji zostanie wyjaśnione w analizie rozwiązania.

### Rozwiązanie

Do porządkowania stanu tworzonego obiektu w obliczu wyjątku należy zastosować w konstruktorze blok kodu chronionego `try` i klauzulę `catch`. Postępowanie to, na przykładzie dwóch prostych klas (`Device` i `Broker`), ilustruje listing 9.2. Klasa `Broker` tworzy na sterce dwa obiekty

klasy Device, dbając o uporządkowanie stertry w przypadku zrzucenia wyjątku przy konstrukcji któregoś z obiektów.

Listing 9.2. Konstruktor odporny na wyjątki

```
#include <iostream>
#include <stdexcept>

using namespace std;

class Device {
public:
    Device(int devno) {
        if (devno == 2)
            throw runtime_error("Poważny problem");
    }
    ~Device() {}
};

class Broker {
public:
    Broker (int devno1, int devno2) :
        dev1_(NULL), dev2_(NULL) {
        try {
            dev1_ = new Device(devno1); // ujęcie tworzenia obiektów
            dev2_ = new Device(devno2); // stertry w bloku try...
        }
        catch (...) {
            delete dev1_; // ...porządkowanie stertry i przerzucenie
            throw; // wyjątku do wywołującego.
        }
    }
    ~Broker() {
        delete dev1_;
        delete dev2_;
    }

private:
    Broker();
    Device* dev1_;
    Device* dev2_;
};

int main() {
    try {
        Broker b(1, 2);
    }
    catch(exception& e) {
        cerr << "Wyjątek: " << e.what() << endl;
    }
}
```

## Analiza

Kiedy mówimy, że konstruktor, metoda, destruktory albo cokolwiek innego jest „odporny na wyjątki”, mamy na myśli to, że oferuje gwarancje blokowania wycieku zasobów i nie pozostawia obiektu w stanie niespójnym. W języku C++ gwarancje te określane są mianem gwarancji *zwykłej* (brak wycieku zasobów) i gwarancji *silnej* (spójność stanu obiektu).

Zwykła gwarancja odporności na wyjątki oznacza, że rzucenie wyjątku nie doprowadzi do wycieku zasobów przy okazji realizowanej operacji i że obiekty uczestniczące w operacji pozostaną w stanie zdadności do użycia (co oznacza, że będzie można wywoływać na ich rzecz metody, a przynajmniej destruktor — czyli że obiekty nie zostaną uszkodzone). Gwarancja zwykła oznacza też, że program po rzuceniu wyjątku pozostanie w stanie spójnym, choć niekoniecznie z góry znanym. Zasada jest więc prosta: jeśli gdziekolwiek w ciele metody dochodzi do rzucenia wyjątku, metoda nie może osierocić obiektów sterty, a obiekty uczestniczące w operacji będą mogły być przez wywołującego albo przywrócone do pierwotnego stanu, albo przynajmniej będzie je można usunąć. Druga z gwarancji, czyli gwarancja *silna*, to pewność, że niepowodzenie operacji nie zmieni stanu uczestniczącego w niej obiektu (tak jakby się w ogóle nie odbyła). Dotyczy to pokonstrukcyjnych operacji na obiektach, ponieważ z definicji obiekt, który rzuci wyjątek z konstruktora, nie może zostać uznany za w pełni skonstruowany, a zatem nie istnieje wcale i nigdy nie był w stanie spójnym. Do odporności metod klas wrócimy w recepturze 9.4. Tymczasem skupmy się na konstruktorach.

Kod z listingu 9.2 definiuje dwie klasy: `Device` (urządzenie) i `Broker` (pośrednik), które służą jedynie jako atrapy ilustrujące model, w którym występuje klasa zarządzająca urządzeniami i same urządzenia (np. klasa inicjująca połączenie pomiędzy dwoma urządzeniami i pośredniczącą w komunikacji pomiędzy nimi). Jeśli dostępne jest tylko jedno z dwóch urządzeń, pośrednik jest bezużyteczny, więc utworzenie pośrednika wraz parą obiektów urządzeń należy traktować łącznie, jako transakcję — albo wykonaną w całości, albo niebyłą. Jeśli nie uda się pozyskać któregośkolwiek z pary obiektów, drugi musi zostać zwolniony. W ten sposób zapewniamy brak wycieku pamięci i innych zasobów.

Zadanie realizujemy przy użyciu słów kluczowych `try` i `catch`. W konstruktorze obiektu klasy `Broker` operacje przydziału pary obiektów `Device` są umieszczane w bloku kodu chronionego (bloku `try`), z przechwytywaniem wszelkich wyjątków operacji w stosownej klauzuli `catch`:

```
try {
    dev1_ = new Device(devno1);
    dev2_ = new Device(devno2);
}
catch (...) {
    delete dev1_;
    throw;
}
```

Wielokropek w klauzuli `catch` oznacza dopasowanie obiektu wyjątku dowolnego typu. Dokładnie tego nam trzeba, bo nie wiemy, jaki wyjątek może zostać rzucony z konstruktorów obiektów, ale niezależnie od tego konsekwentnie zamierzamy uporządkować stan programu. Przechwycony wyjątek powinniśmy przy tym przerzucić w górę stosu wywołań tak, aby ten, kto konkretyzował klasę `Broker`, również mógł obsłużyć wyjątek, choćby przez wypisanie komunikatu o błędzie.

W ramach obsługi wyjątku usuwamy jedynie obiekt `dev1_`, ponieważ ostatnią szansę na rzucenie wyjątku jest wywołanie operatora `new` dla składowej `dev2_`. Jeśli operator rzuci wyjątek, nie dojdzie do przypisania wartości do składowej `dev2_`, więc nie będzie też czego zwalniać wywołaniem `delete`. Gdyby jednak konstruktor robił coś jeszcze po przydzieleniu obiektu `dev2_`, należałoby zwolnić oba obiekty, jak tu:

```
try {
    dev1_ = new Device(devno1);
    dev2_ = new Device(devno2);
    foo_ = new MyClass(); // ostatnia szansa na wyjątek
}
```

```

}
catch (...) {
    delete dev1_;
    delete dev2_;
    throw;
}

```

W tym przypadku nie musimy zajmować się problemem zwalniania wskaźników, którym nigdy nie przypisano poprawnej wartości (o ile tylko będą właściwie inicjalizowane wskazaniami NULL), bo wywołanie `delete` ze wskaźnikiem o wartości NULL nie ma żadnego efektu. Innymi słowy, jeśli wyjątek zostanie zgłoszony już przez konstruktor `dev1_`, to wykonanie instrukcji `delete dev2_` nie stanowi żadnego zagrożenia dla stabilności programu, bo składowa `dev2_` ma wciąż wartość NULL, przypisaną z poziomu listy inicjalizacyjnej konstruktora.

Zgodnie z komentarzem z receptury 9.1 projektowanie solidnej i elastycznej strategii stosowania i obsługi wyjątków nie jest sprawą prostą; to samo dotyczy zapewniania odporności na wyjątki. Czytelników zainteresowanych szerszym omówieniem tego tematu zachęcam do lektury książki *Exceptional C++* Herba Suttera (Addison\_Wesley).

## Zobacz również

Recepturę 9.3.

## 9.3. Uodpornianie listy inicjalizacyjnej konstruktora

### Problem

Mamy zamiar zainicjalizować składowe obiektu klasy z poziomu listy inicjalizacyjnej konstruktora, a więc nie możemy skorzystać z rozwiązania prezentowanego w recepturze 9.2.

### Rozwiązanie

Rozwiązanie polega na zastosowaniu specjalnej składni `try-catch`, przechwytyjącej wyjątki rzucane z listy inicjalizacyjnej konstruktora. Przykład na listingu 9.3.

*Listing 9.3. Obsługa wyjątków listy inicjalizacyjnej*

```

#include <iostream>
#include <stdexcept>

using namespace std;

// klasa jakiegoś urzędnika
class Device {
public:
    Device(int devno) {
        if (devno == 2)
            throw runtime_error("Poważny problem");
    }
    ~Device() {}
private:
    Device();

```

```

};

class Broker {

public:
    Broker (int devno1, int devno2)
        try : dev1_(Device(devno1)), // przydział w ramach listy
            dev2_(Device(devno2)) {} // inicjalizacyjnej
        catch (...) {
            throw; // tu zarejestrowanie komunikatu albo przekształcenie
                // błędu (zobacz komentarz w punkcie "Analiza")
        }
    ~Broker() {}

private:
    Broker();
    Device dev1_;
    Device dev2_;
};

int main() {

    try {
        Broker b(1, 2);
    }
    catch(exception& e) {
        cerr << "Wyjątek: " << e.what() << endl;
    }
}

```

## Analiza

Składnia obsługi wyjątków listy inicjalizacyjnej różni się nieco od klasycznej składni języka C++, bo w roli ciała konstruktora występuje blok try osadzony na liście inicjalizacyjnej. Najważniejszym z naszego punktu widzenia elementem listingu 9.3 jest konstruktor klasy Broker:

```

Broker (int devno1, int devno2) // nagłówek konstruktora, jak zwykle
    try : dev1_(Device(devno1)), // idea podobna, jak w try {...}
        dev2_(Device(devno2)) {
// ciało konstruktora
    }
    catch (...) { // klauzula catch "poza"
        throw; // ciałem konstruktora
    }
}

```

Zachowanie try i catch jest zgodne z oczekiwaniem; jedyną różnicą względem klasycznej składni bloku kodu chronionego try jest to, że w tym przypadku kod ten stanowi również listę inicjalizacyjną konstruktora: słowo kluczowe try poprzedza dwukropek rozpoczynający tę listę, a za nią zaczyna się właściwy blok try, rozciągający się na całe ciało konstruktora. Cokolwiek zostanie rzucone w tym bloku — czy to z listy inicjalizacyjnej, czy z ciała konstruktora — zostanie przechwycone w klauzuli catch znajdującej się jakby już poza ciałem konstruktora. Rzecz jasna w ciele konstruktora można do woli osadzać klasyczne bloki try-catch, ale zagnieżdżanie bloków kodu chronionego wygląda odpychająco.

Przykład z listingu 9.3 różni się od kodu z listingu 9.2 nie tylko przesunięciem operacji inicjalizacji składowych na listę inicjalizacyjną konstruktora — także tym, że nie dochodzi tu do tworzenia obiektów na stercie przy użyciu operatora new. Ma to ilustrować szereg aspektów związanych z bezpieczeństwem inicjalizacji składowych klas.



Przed wszystkim przydział obiektów na stosie zamiast na stercie pozwala kompilatorowi na stosowanie wbudowanych mechanizmów zabezpieczających. Otóż, jeśli którykolwiek z obiektów przydzielanych na liście inicjalizacyjnej spowoduje wyjątek, jego pamięć zostanie automatycznie zwolniona w ramach zwijania stosu przez mechanizm propagacji wyjątku. Po drugie zaś, w wyniku działania tego samego mechanizmu usunięte zostaną wszystkie obiekty już skonstruowane, a wszystko odbędzie się automatycznie, bez potrzeby stosowania jawnego wywołania `delete`.

Niekiedy trzeba jednak przydzielać obiekty właśnie na stercie. Rozważmy więc podejście zastosowane w pierwotnej wersji klasy `Broker`, na listingu 9.2. Przecież wskaźniki też można inicjalizować z poziomu listy inicjalizacyjnej, prawda?

```
class BrokerBad {  
  
public:  
    BrokerBad (int devno1, int devno2)  
        try : dev1_(new Device(devno1)), // przydział obiektów na stercie  
            dev2_(new Device(devno2)) {} // w liście inicjalizacyjnej  
        catch (...) {  
            if (dev1_) {  
                delete dev1_; // nie powinno się skompilować,  
                delete dev2_; // a jeśli się da, należy tego unikać  
            }  
            throw; // przerzucenie wyjątku do wywołującego  
        }  
    ~BrokerBad() {  
        delete dev1_;  
        delete dev2_;  
    }  
  
private:  
    BrokerBad();  
    Device* dev1_;  
    Device* dev2_;  
};
```

Otóż nie. Pojawiają się bowiem dwa problemy. Przed wszystkim zaś kompilator nie powinien zezwolić na stosowanie takich konstrukcji, bo kod bloku `catch` konstruktora nie powinien odwoływać się do składowych obiektu — przecież w tym momencie one jeszcze w ogóle nie istnieją. Po drugie, jeśli nawet kompilator „przepuści” taki kod, należałoby go unikać. Załóżmy, że dojdzie do rzucenia wyjątku przy konstrukcji obiektu dla składowej `dev1_`. W ramach obsługi wyjątku dojdzie do próby wykonania poniższego kodu:

```
        catch (...) {  
            if (dev1_) {  
                delete dev1_; // jaką wartość ma dev1_?  
                delete dev2_; // usuwanie wskaźnika o nieokreślonej wartości  
            }  
            throw; // przerzucenie wyjątku do wywołującego  
        }  
    }
```

Jeśli przy konstrukcji obiektu dla składowej `dev1_` zostanie rzucony wyjątek, operator `new` będzie miał okazję zwrócić adresu nowo przydzielonego obszaru pamięci i tym samym nie zmieni wartości `dev1_`. Cóż to więc będzie za wartość? Otóż, nie można tego określić, bo składowa nie została wcześniej zainicjalizowana. W efekcie dojdzie do wywołania `delete dev1_`, przy nieokreślonej wartości `dev1_`, co skończy się najpewniej usunięciem zupełnie przypadkowego wskaźnika, czyli w najlepszym razie — natychmiastowym załamaniem programu (zwolnieniem z pracy, koniecznością życia w hańbie itd.).

Aby uniknąć potencjalnego załamania programu (albo i kariery), należy na liście inicjalizacyjnej ograniczać się do inicjalizowania składowych wskaźnikowych wartościami NULL, a przydział pamięci sterty odłożyć do ciała konstruktora. Tam można będzie spokojnie przechwycić i obsłużyć ewentualne niepowodzenia przydziału i uporządkować stan obiektu, bo wywołanie delete na rzecz wartości NULL nie daje żadnych negatywnych skutków (nie daje żadnych skutków w ogóle):

```
BrokerBetter (int devno1, int devno2) :
dev1_(NULL), dev2_(NULL) {
    try {
        dev1_ = new Device(devno1);
        dev2_ = new Device(devno2);
    }
    catch(...) {
        delete dev1_;           //zawsze w porządku
        throw;
    }
}
```

Podsumowując: jeśli trzeba korzystać w klasie ze składowych wskaźnikowych, należy je inicjalizować wartościami NULL z poziomu listy inicjalizacyjnej, a faktyczny przydział pamięci odłożyć do ciała konstruktora, gdzie można go ująć w bloku kodu chronionego. Tam też można bezpiecznie zwolnić wskazania w ramach obsługi ewentualnego wyjątku przydziału. Jednak wszędzie tam, gdzie wskaźniki da się zastąpić zmiennymi automatycznymi, lepiej zastosować właśnie takie zmienne i inicjalizować je na liście inicjalizacyjnej konstruktora z użyciem specjalnej składni try-catch — znakomicie ułatwi to obsługę ewentualnych wyjątków inicjalizacji.

## Zobacz również

Recepturę 9.2.

## 9.4. Uodpornianie metod klasy

### Problem

Piszemy metodę, która ma dawać zwykłą i silną gwarancję odporności na wyjątki, a więc nie dopuszczać do wycieków zasobów i nie pozostawiać obiektu w niepoprawnym stanie.

### Rozwiązanie

Trzeba wytypować operacje potencjalnie rzucające wyjątki i wykonać je w pierwszej kolejności, oczywiście w bloku kodu chronionego (try-catch). Aktualizację stanu obiektu wolno podjąć dopiero po zakończeniu wykonania kodu potencjalnie ryzykownego. Przykład uodpornienia metody na wyjątki prezentuje listing 9.4.

*Listing 9.4. Metoda odporna na wyjątki*

```
class Message {
public:
    Message(int bufSize = DEFAULT_BUF_SIZE) :
```

```

        bufSize_(bufSize),
        initBufSize_(bufSize),
        msgSize_(0),
        buf_(NULL) {
            buf_ = new char[bufSize];
        }

~Message() {
    delete[] buf_;
}

// dołączanie znaków
void appendData(int len, const char* data) {
    if (msgSize_+len > MAX_SIZE) {
        throw out_of_range("Rozmiar danych przekracza limit.");
    }
    if (msgSize_+len > bufSize_) {

        int newBufSize = bufSize_;
        while ((newBufSize *= 2) < msgSize_+len);

        char* p = new char[newBufSize]; // przydział pamięci
                                        // dla nowego bufora

        copy(buf_, buf_+msgSize_, p); // kopiowanie poprzedniej zawartości
        copy(data, data+len, p+msgSize_); // kopiowanie nowych danych

        msgSize_ += len;
        bufSize_ = newBufSize;

        delete[] buf_; // pozbycie się poprzedniego bufora
        buf_ = p; // i przestawienie wskaźnika bufora na nowy
    }
    else {
        copy(data, data+len, buf_+msgSize_);
        msgSize_ += len;
    }
}

// kopiowanie danych do bufora wskazanego przez wywołującego
int getData(int maxLen, char* data) {
    if (maxLen < msgSize_) {
        throw out_of_range("Bufor docelowy nie pomieści danych.");
    }
    copy(buf_, buf_+msgSize_, data);
    return(msgSize_);
}

private:
    Message(const Message& orig) {} // por. recepturę 9.5.
    Message& operator=(const Message& rhs) {} //
    int bufSize_;
    int initBufSize_;
    int msgSize_;
    char* buf_;
};

```

## Analiza

Klasa `Message` z listingu 9.4 to klasa reprezentująca komunikat tekstowy składający się ze znaków; obiekty takiej klasy mogłyby służyć np. do kopertowania komunikatów tekstowych bądź danych binarnych przekazywanych pomiędzy systemami. Najbardziej interesuje nas metoda

appendData klasy, która dołącza dane przekazane za pośrednictwem argumentu wywołania do bieżącej zawartości bufora. Jeśli to konieczne, bufor jest powiększany (tzn. odbywa się przydział nowego bufora). Metoda daje silną gwarancję odporności na wyjątki, choć ta jej zaleta nie rzuca się w oczy.

Spójrzmy choćby na tę część metody appendDate:

```
if (msgSize_+len > bufSize_) {  
    int newBufSize = bufSize_;  
    while ((newBufSize *= 2) < msgSize_+len);  
  
    char* p = new char[newBufSize];
```

Zadaniem tego fragmentu metody jest powiększenie bufora. Rozmiar nowego bufora jest obliczany przez podwajanie bieżącego rozmiaru bufora, do skutku. Ten fragment jest zupełnie bezpieczny, bo wyjątek mógłby zostać rzucony jedynie przez operator new, a i to nie spowodowałoby żadnego ryzyka: obiekt nie został jeszcze poddany żadnym modyfikacjom, nie doszło też do przydzielenia żadnych nowych zasobów. Jeśli system operacyjny nie będzie w stanie zrealizować przydziału pamięci dla nowego bufora, operator new rzuci wyjątek bad\_alloc.

Jeśli uda się skutecznie przydzielić pamięć, można przystąpić do aktualizacji stanu obiektu, co polega na skopiowaniu danych do nowego bufora i zaktualizowaniu składowej wskazującej bufor:

```
copy(buf_, buf_+msgSize_, p);  
copy(data, data+len, p+msgSize_);  
  
msgSize_ += len;  
bufSize_ = newBufSize;  
  
delete[] buf_;  
buf_ = p;
```

Żadna z powyższych operacji nie może rzucać wyjątków, więc całość jest zupełnie bezpieczna (a to dlatego, że bufor zawiera sekwencję znaków typu char; wyjaśnienia skutków takiej decyzji projektowej należy szukać w recepturze 9.5).

Jak widać, rozwiązanie jest proste; do tego odzwierciedla dobrze ogólną strategię implementowania metod z silną gwarancją odporności na wyjątki, wedle której wszystko, co może spowodować wyjątek należy wykonać najpierw, a kiedy już wszystkie ryzykowne operacje zostaną pomyślnie wykonane, można przystąpić do modyfikowania stanu obiektu, na rzecz którego nastąpiło wywołanie. W metodzie appendData do obliczania rozmiaru nowego bufora służy zmienna tymczasowa; w ten sposób eliminujemy przedwczesną modyfikację stanu obiektu, ale czy naprawdę wystarczy to do zapewnienia zwykłej gwarancji, czyli braku wycieku zasobów? Owszem, choć ledwo.

Otóż, algorytm copy dla każdego elementu kopiowanej sekwencji wywołuje operator przypisania operator=. W klasie z listingu 9.4 elementy te są typu char, a wiadomo, że przypisanie znaku do innego znaku nie może spowodować wyjątku. Więc „owszem, ledwo”, bo odporność na wyjątki nie wynika z charakteru algorytmu copy, a jedynie ze specjalnych cech typu, na którym ów algorytm operuje. Nie wolno więc zakonotować sobie, że copy nie rzuca wyjątków.

Żałujemy na chwilę, że zamiast zwykłego bufora znaków chcemy w klasie Message przechowywać tablicę elementów dowolnego typu. Klasę taką moglibyśmy zdefiniować przy użyciu szablonu klasy, jak na listingu 9.5.

Listing 9.5. Uogólniona klasa komunikatu

```
template<typename T>
class MessageGeneric {

public:
    MessageGeneric(int bufSize = DEFAULT_BUF_SIZE) :
        bufSize_(bufSize),
        initBufSize_(bufSize),
        msgSize_(0),
        buf_(new T[bufSize]) {}

    ~MessageGeneric() {
        delete[] buf_;
    }

    void appendData(int len, const T* data) {
        if (msgSize_+len > MAX_SIZE) {
            throw out_of_range("Rozmiar danych przekracza limit.");
        }
        if (msgSize_+len > bufSize_) {

            int newBufSize = bufSize_;
            while ((newBufSize *= 2) < msgSize_+len);

            T* p = new T[newBufSize];

            copy(buf_, buf_+msgSize_, p); // czy copy może zrzucić wyjątek?
            copy(data, data+len, p+msgSize_);

            msgSize_ += len;
            bufSize_ = newBufSize;

            delete[] buf_; // pozbycie się poprzedniego bufora
            buf_ = p; // i przestawienie wskaźnika bufora na nowy
        }
        else {
            copy(data, data+len, buf_+msgSize_);
            msgSize_ += len;
        }
    }

    // kopiowanie danych do bufora wskazanego przez wywołującego
    int getData(int maxLen, T* data) {
        if (maxLen < msgSize_) {
            throw out_of_range("Bufor docelowy nie pomieści danych.");
        }
        copy(buf_, buf_+msgSize_, data);
        return(msgSize_);
    }

private:
    MessageGeneric(const MessageGeneric& orig) {}
    MessageGeneric& operator=(const MessageGeneric& rhs) {}
    int bufSize_;
    int initBufSize_;
    int msgSize_;
    T* buf_;
};
```

Tu musimy być ostrożniejsi, bo nie możemy już zakładać niczego co do typu elementów bufora. Nie możemy więc mieć pewności, czy `T::operator=` nie zrzuci wyjątków. Trzeba się przygotować na najgorsze.

Mianowicie, wywołanie algorytmu `copy` należy ująć w bloku kodu chronionego:

```
try {
    copy(buf_, buf_+msgSize_, p);
    copy(data, data+len, p+msgSize_);
}
catch(...) { // nie obchodzi nas, co zostało zrzucone;
    delete[] p; // tak czy inaczej, trzeba posprzątać
    throw; // i przerzucić wyjątek do wywołującego
}
```

Operator wielokropka wymusza przechwycenie każdego zrzuconego wyjątku, można więc mieć pewność, że jeśli `T::operator=` rzuca wyjątki, to nie przegapimy żadnego, niezależnie od ich typu. W każdym przypadku zdołamy posprzątać po sobie, zwalniając bufor przydzielony w pamięci sterty.

Jeśli chodzi o ścisłość, to algorytm `copy` faktycznie sam z siebie nie rzuca nigdy żadnych wyjątków. Czyni to najwyżej wykorzystywany w `copy` operator przypisania `operator=`. Algorytm `copy`, tak jak i pozostałe algorytmy biblioteki standardowej, jest neutralny względem wyjątków: wszystko, co zostanie zrzucone w czasie jego wykonania, zostanie przerzucone do wywołującego; żaden z algorytmów nie „zjada” wyjątków; wszystkie natomiast rezerwują sobie prawo do przechwycenia wyjątku, realizacji pewnych operacji porządkujących i przerzucenia niezmiennego wyjątku do wywołującego.

Uodpornianie metod własnych klas na wyjątki to dość żmudna praca. Wymaga uwzględnienia wielu subtelnosci i zidentyfikowania wszystkich miejsc, w których może dojść do zrzucenia wyjątków. Gdzie może to nastąpić? W każdym wywołaniu funkcji. Operatory typów macierzystych nie rzucają wyjątków, a destruktory z zasady nie powinny ich rzucać; poza tym wszystkie wywołania funkcji, czy to samodzielnych, czy to metod klas, operatorów, konstruktorów itd., to potencjalne źródła wyjątków. Przykłady z listingów 9.4 i 9.5 ilustrują stosunkowo wąski zakres wyjątków; klasy zawierają niewiele składowych, a zachowanie metod jest dobrze rozpoznane. Jednak w miarę mnożenia się składowych i metod, a także przy dziedziczeniu i funkcjach wirtualnych zachowanie silnej gwarancji odporności na wyjątki staje się wyzwaniem.

Trzeba więc zachować zdrowy rozsądek i uodpornić aplikację na wyjątki tylko w takim zakresie, w jakim jest to niezbędne. Jeśli na przykład piszemy program opartego na oknach dialogowych kreatora stron WWW, powinniśmy w terminarzu projektu zarezerwować czas na analizę i testowanie niezbędne do zachowania silnej odporności na wyjątki. Być może okaże się, że użytkownicy akceptują w niektórych sytuacjach (byle nie za często!) lakoniczne komunikaty o błędach w rodzaju „Nieznany błąd, wyłączam”. Z drugiej strony, w oprogramowaniu sterującym kątem nachylenia rotora śmigłowca klient będzie zapewne nalegał na wdrożenie wszelkich możliwych zabezpieczeń i zadbanie o to, aby pilot nie ujrzał nigdy komunikatu „Nieznany błąd, wyłączam”.

## 9.5. Bezpieczne kopiowanie obiektu

### Problem

Zaimplementować podstawowe operacje kopiowania obiektów klasy (operator przypisania i konstruktor kopiujący), tak aby kopiowanie było odporne na wyjątki.

## Rozwiązanie

W ramach rozwiązania należałoby wdrożyć taktykę z receptury 9.4, polegającą na przesuwaniu ryzykownych operacji na początek ścieżki wykonania, przed operacje aktualizujące stan obiektów. W przykładzie powrócimy do klasy `Message`, uzupełniając ją tym razem o operator przypisania i konstruktor kopiujący — zobacz listing 9.6.

Listing 9.6. Odporne na wyjątki implementacje operacji kopiujących

```
#include <iostream>
#include <string>

const static int DEFAULT_BUF_SIZE = 3;
const static int MAX_SIZE         = 4096;

class Message {
public:
    Message(int bufSize = DEFAULT_BUF_SIZE) :
        bufSize_(bufSize),
        initBufSize_(bufSize),
        msgSize_(0),
        key_("") {
        buf_ = new char[bufSize]; //uwaga: w ciele konstruktora
    }

    ~Message( ) {
        delete[] buf_;
    }

    // odporny na wyjątki konstruktor kopiujący
    Message(const Message& orig) :
        bufSize_(orig.bufSize_),
        initBufSize_(orig.initBufSize_),
        msgSize_(orig.msgSize_),
        key_(orig.key_) { // tu może dojść do zrzucenia wyjątku...

        buf_ = new char[orig.bufSize_]; //...tu też
        copy(orig.buf_, orig.buf_+msgSize_, buf_); // tu już nie
    }

    // odporne na wyjątki przypisanie, z odwołaniem do konstruktora kopiującego
    Message& operator=(const Message& rhs) {

        Message tmp(rhs); // konstrukcja (kopiująca) obiektu tymczasowego
        swapInternals(tmp); // podmiana składowych
        return(*this); // po opuszczeniu zasięgu dojdzie do usunięcia tmp
        // wraz z oryginalnymi danymi
    }

    const char* data( ) {
        return(buf_);
    }

private:
    void swapInternals(Message& msg) {
        // key_ nie jest typem wbudowanym i może zrzucić wyjątki,
        // więc zostanie podmieniony jako pierwszy
        swap(key_, msg.key_);

        // jeśli nie pojawił się wyjątek, można podmienić wartości proste
        swap(bufSize_, msg.bufSize_);
    }
};
```

```

        swap(initBufSize_, msg.initBufSize_);
        swap(msgSize_,      msg.msgSize_);
        swap(buf_,          msg.buf_);
    }
    int bufSize_;
    int initBufSize_;
    int msgSize_;
    char* buf_;
    string key_;
};

```

## Analiza

Cała czarna robota jest tu składana na barki konstruktora kopiującego i metody `swapInternals`. Konstruktor kopiujący inicjalizuje składowe typów prostych i jedną składową typu klasy z poziomu listy inicjalizacyjnej. W ciele konstruktora następuje przydział pamięci dla bufora i skopiowanie danych do tegoż bufora. Proste, prawda? Ale dlaczego w tej kolejności? Można by przecież posłużyć się argumentem, że całość inicjalizacji powinna się odbyć w ramach listy inicjalizacyjnej konstruktora, ale to otworzyłoby furtkę całej grupie subtelnych błędów.

Założmy przykładowo, że przydział bufora zrealizujemy w ramach listy inicjalizacyjnej, jak tu:

```

Message(const Message& orig) :
    bufSize_(orig.bufSize_),
    initBufSize_(orig.initBufSize_),
    msgSize_(orig.msgSize_),
    key_(orig.key_),
    buf_(new char[orig.bufSize_]) {
    copy(orig.buf_, orig.buf_+msgSize_, buf_);
}

```

Można by pomyśleć, że to całkiem dobre rozwiązanie, bo jeśli wywołanie `new` przydzielające bufor zawiedzie, dojdzie do automatycznego usunięcia wszystkich pozostałych, w pełni już skonstruowanych obiektów. Ale taki wygodny automatyzm wcale nie jest pewny, bo składowe *nie są* inicjalizowane w kolejności, którą określa lista inicjalizacyjna, ale zgodnie z kolejnością występowania w deklaracji klasy. Oto kolejność deklaracji składowych:

```

int bufSize_;
int initBufSize_;
int msgSize_;
char* buf_;
string key_;

```

Jak widać, składowa `buf_` będzie inicjalizowana przed `key_`. Jeśli inicjalizacja `key_` spowoduje wyjątek, nie dojdzie do zwolnienia `buf_`, a w pamięci zalegać będzie niewskazywany niczym obszar pamięci. Można się przed tym zabezpieczyć blokiem `try-catch` w ciele konstruktora (zobacz recepturę 9.2), ale prościej będzie przenieść inicjalizację `buf_` do ciała konstruktora i w ten sposób na pewno wymusić realizację po zakończeniu przetwarzania listy inicjalizacyjnej.

Wywołanie algorytmu `copy` na pewno nie zrzuci wyjątków, bo algorytm operuje na typie prostym. Znowu mamy do czynienia z subtelnością odporności na wyjątki: wywołanie algorytmu `copy` mogłoby spowodować wyjątek, gdyby kopiowanie dotyczyło obiektów, a nie zmiennych wbudowanych (np. gdyby rzecz dotyczyła kontenera elementów typu `T`); w takim przypadku należałoby ująć wywołanie w bloku `try` i zwalniać pamięć w ramach obsługi wyjątku.

Alternatywną metodą kopiowania obiektu jest wywoływanie operatora przypisania (`operator=`). Ponieważ operator przypisania ma potrzeby zbliżone do konstruktora kopiującego (bo



oba mają przepisać wartości składowych obiektu źródłowego do składowych obiektu docelowego), często w implementacji operatora przypisania odwołujemy się do gotowego już konstruktora kopiującego. Równie częstym ulepszeniem jest zastosowanie prywatnej metody do podmiany składowych obiektów. Chciałbym być autorem tej techniki, ale muszę oddać zaślugi Herbowi Sutterowi i Stephenowi Dewhurstowi, bo to od nich ją ściągnąłem.

Technika ta, zastosowana na listingu 9.6, nie powinna wymagać szerszego komentarza, jednak na wszelki wypadek pozwolę ją sobie opisać. Weźmy na warsztat pierwszy wiersz operatora przypisania, konstruującego obiekt tymczasowy przy użyciu konstruktora kopiującego klasy:

```
Message tmp(rhs);
```

Otrzymaliśmy w ten sposób kopię obiektu źródłowego przypisania. Zasadniczo `tmp` jest teraz równoważne `rhs`. Wystarczy już tylko podmienić składowe `*this` ze składowymi `tmp`:

```
swapInternals(tmp);
```

Do metody `swapInternals` wrócimy za moment, tymczasem zaś składowe `*this` odzwierciedlają składowe `tmp`. A skoro `tmp` było wierną kopią `rhs`, to `*this` jest teraz równoważne `rhs`. Został jeszcze ten obiekt tymczasowy; na szczęście nie stanowi żadnego problemu, bo w momencie zwracania `*this` jego zasięg się skończy i zostanie usunięty, znikając wraz z poprzednimi wartościami składowych:

```
return(*this);
```

I już. Czy to odporne na wyjątki? Na pewno odporna jest operacja konstrukcji `tmp`, bo zadbaliliśmy o odporność naszego konstruktora kopiującego. Realizacja przypisania opiera się w znacznej mierze na wywołaniu metody `swapInternals`, więc to jej odporność na wyjątki będzie tu decydująca.

Metoda `swapInternals` wymienia wszystkie składowe bieżącego obiektu ze składowymi obiektu przekazanego w wywołaniu. Wymiana odbywa się za pośrednictwem standardowego algorytmu `swap`, przyjmującego parę argumentów `a` i `b`, tworzącego kopię `a`, przypisującego `b` do `a` i wreszcie przypisującego do `b` kopię `a`. Algorytm `swap` jest neutralny wobec wyjątków i na nie odporny, bo jedynymi wyjątkami zrzucanymi z algorytmu mogą być te przetrzucane z obiektów, na których operuje. Algorytm nie korzysta z pamięci sterty, gwarantuje więc brak wycieku zasobów.

Ponieważ składowa `key_` nie jest składową typu prostego, co oznacza, że operacje na niej mogą prowokować wyjątki, jest wymieniana w pierwszej kolejności. Jeśli w trakcie wymiany dojdzie do zrzucenia wyjątku, unikniemy naruszenia pierwotnego stanu pozostałych składowych. Nie zyskamy jednak pewności nienaruszenia składowej `key_`. W tym aspekcie jesteśmy całkowicie zdani na łaskę i niełaskę twórcy implementacji klasy obiektu i jej gwarancji odporności na wyjątki. Jeśli ta podmiana nie sprowokuje wyjątku, jesteśmy w domu, bo wiadomo, że operacje na typach wbudowanych już ich nie sprowokują. Widać więc, że `swapInternals` jest metodą dającą i zwykłą, i silną gwarancję odporności na wyjątki.

A co, gdyby wymiana miała objąć większą liczbę wartości typów złożonych? Gdybyśmy mieli w klasie dwie składowe typu `string`, początek metody `swapInternals` mógłby wyglądać tak:

```
void swapInternals(Message& msg) {
    swap(key_, msg.key_);
    swap(myObj_, msg.myObj_);
    //...
```

Mamy problem: jeśli druga podmiana zrzuci wyjątek, w jaki sposób bezpiecznie wycofamy pierwszą podmianę? Składowa `key_` będzie miała nową wartość, ale w obliczu niepowodzenia podmiany `myObj` trzeba tę wartość uznać za niepoprawną. Jeśli wywołujący przechwyci wyjątek i zechce kontynuować wykonanie jakby nigdy nic, będzie miał do czynienia z obiektem w niespójnym stanie, a na pewno z innym niż ten, z którym zaczynał operację. Można skopiować `key_` do ciągu tymczasowego, ale ktoś zagwarantuje, że kopiowanie nie zrzuci wyjątku?

Problem można wyeliminować z użyciem obiektów serty:

```
void swapInternals(Message& msg) {  
    // key_ jest typu string* a myObj jest typu MyClass*  
    swap(key_, msg.key_);  
    swap(myObj_, msg.myObj_);  
}
```

Oznacza to, co prawda, konieczność zarządzania przydziałami pamięci, ale proces uodparniania na wyjątki często wymaga reorganizacji projektów, warto więc zaplanować go już w pierwszych fazach projektowania.

Wytyczne wynikające z niniejszej receptury stanowią powtórzenie strategii postulowanej w poprzednich: prace ryzykowne trzeba wykonywać najpierw (w blokach `try-catch`), uzależniając od ich powodzenia wykonanie operacji modyfikujących stan obiektu. Jeśli operacje ryzykowne spowodują wyjątek, można będzie podjąć operacje porządkowe, przywracając pierwotny stan obiektu.

## Zobacz również

Receptury 9.2 i 9.3.