Programowanie funkcyjne w języku C++

Tworzenie lepszych aplikacji

Ivan Čukić

Tytuł oryginału: Functional Programming in C++: How to improve your C++
programs using functional techniques

Tłumaczenie: Jacek Janusz

ISBN: 978-83-283-4703-8

Original edition copyright © 2019 by Manning Publications Co.
All rights reserved.

Polish edition copyright © 2019 by HELION SA.
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Helion SA dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/profun.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/profun
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Przedmowa 9

O książce 11

O autorze 15

Rozdział 1. Wprowadzenie do programowania funkcyjnego *17*

1.1. Co to jest programowanie funkcyjne? 18

1.1.1. Związek z programowaniem obiektowym *19*

1.1.2. Praktyczny przykład porównania programowania imperatywnego i deklaratywnego *20*

1.2. Funkcje czyste 24

1.2.1. Unikanie stanu mutowalnego *27*

1.3. Myślenie w sposób funkcjonalny 29

1.4. Korzyści wynikające z programowania funkcyjnego 31

1.4.1. Zwięzłość i czytelność kodu *32*

1.4.2. Współbieżność i synchronizacja *33*

1.4.3. Ciągła optymalizacja *33*

1.5. Przekształcanie C++ w funkcyjny język programowania 34

1.6. Czego nauczysz się w trakcie czytania tej książki? 36

Podsumowanie 36

Rozdział 2. Pierwsze kroki z programowaniem funkcyjnym *39*

2.1. Funkcje używające innych funkcji? 40

2.2. Przykłady z biblioteki STL 42

2.2.1. Obliczanie średnich *42*

2.2.2. Zwijanie *45*

2.2.3. Przycinanie łańcucha *48*

2.2.4. Partycjonowanie kolekcji na podstawie predykatu *50*

2.2.5. Filtrowanie i transformacja *52*

2.3. Problemy ze składaniem algorytmów STL 53

2.4. Tworzenie własnych funkcji wyższego rzędu 55

2.4.1. Otrzymywanie funkcji w postaci argumentów *55*

2.4.2. Implementacja z pętlami *56*

2.4.3. Rekurencja oraz optymalizacja przez rekurencję ogonową *57*

2.4.4. Implementacja przy użyciu zwijania *61*

Podsumowanie 62

Kup ksi k Pole ksi k

Rozdział 3. Obiekty funkcyjne *63*

3.1. Funkcje i obiekty funkcyjne 64

3.1.1. Automatyczna dedukcja typu zwracanego 64

3.1.2. Wskaźniki do funkcji 67

3.1.3. Przeciążanie operatora wywołania 68

3.1.4. Tworzenie generycznych obiektów funkcyjnych 70

3.2. Wyrażenia lambda i domknięcia 73

3.2.1. Składnia wyrażenia lambda 74

3.2.2. Co się kryje wewnątrz wyrażenia lambda? 75

3.2.3. Tworzenie dowolnych zmiennych składowych w wyrażeniach lambda 77

3.2.4. Uogólnione wyrażenia lambda 79

3.3. Tworzenie obiektów funkcyjnych, które są jeszcze bardziej zwięzłe niż wyrażenia lambda 80

3.3.1. Obiekty funkcyjne operatorów w bibliotece STL 83

3.3.2. Obiekty funkcyjne operatorów w innych bibliotekach 84

3.4. Opakowywanie obiektów funkcyjnych przy użyciu std::function 86

Podsumowanie 88

Rozdział 4. Tworzenie nowych funkcji na podstawie istniejących *89*

4.1. Częściowe stosowanie funkcji 90

4.1.1. Ogólny sposób zamiany funkcji dwuargumentowych na jednoargumentowe 92

4.1.2. Użycie std::bind do wiązania wartości z określonymi argumentami funkcji 95

4.1.3. Zamienianie ze sobą argumentów funkcji dwuargumentowej 97

4.1.4. Użycie std::bind z funkcjami mającymi więcej argumentów 98

4.1.5. Użycie wyrażeń lambda jako alternatywy dla std::bind 101

4.2. Rozwijanie: inny sposób podejścia do funkcji 103

4.2.1. Rozwijanie funkcji w prostszy sposób 104

4.2.2. Użycie rozwijania podczas dostępu do bazy danych 106

4.2.3. Rozwijanie a częściowe stosowanie funkcji 109

4.3. Złożenie funkcji 110

4.4. Podnoszenie funkcji — kolejne podejście 113

4.4.1. Odwracanie elementów par w kolekcji 116

Podsumowanie 117

Rozdział 5. Czystość: unikanie stanu mutowalnego *119*

5.1. Problemy ze stanem mutowalnym 120

5.2. Funkcje czyste i przejrzystość referencyjna 122

5.3. Programowanie bez efektów ubocznych 125

5.4. Stan mutowalny i niemutowalny w środowisku współbieżnym 129

5.5. Duże znaczenie kwalifikatora const 132

5.5.1. Logiczna i wewnętrzna zgodność z deklaracją const 134

5.5.2. Optymalizacja funkcji składowych dla zmiennych tymczasowych 136

5.5.3. Pułapki związane z użyciem słowa kluczowego const 138

Podsumowanie 140

Kup ksi k Pole ksi k

Rozdział 6. Wartościowanie leniwe *141*

6.1. Wartościowanie leniwe w C++ 142
6.2. Wartościowanie leniwe jako technika optymalizacyjna 145
6.2.1. Sortowanie kolekcji przy wykorzystaniu wartościowania leniwego 145
6.2.2. Wyświetlanie elementów w interfejsach użytkownika 147
6.2.3. Przycinanie drzew rekurencyjnych przez buforowanie wyników funkcji 148
6.2.4. Programowanie dynamiczne jako forma wartościowania leniwego 150
6.3. Uogólnione zapamiętywanie 152
6.4. Szablony wyrażeń i leniwe łączenie łańcuchów 155
6.4.1. Czystość i szablony wyrażeń 159
Podsumowanie 160

Rozdział 7. Zakresy *163*

7.1. Wprowadzenie do zakresów 165
7.2. Tworzenie widoków danych tylko do odczytu 166
7.2.1. Użycie funkcji filter z zakresami 166
7.2.2. Użycie funkcji transform z zakresami 167
7.2.3. Wartościowanie leniwe wartości zakresów 168
7.3. Mutowalność zmiennych w zakresach 170
7.4. Używanie zakresów ograniczonych i nieskończonych 172
7.4.1. Użycie zakresów ograniczonych do optymalizacji obsługi zakresów wejściowych 172
7.4.2. Tworzenie zakresów nieskończonych przy użyciu wartowników 173
7.5. Używanie zakresów do obliczania częstości pojawiania się słów 175
Podsumowanie 178

Rozdział 8. Funkcyjne struktury danych *179*

8.1. Niemutowalne listy łączone 180
8.1.1. Dodawanie i usuwanie elementów z początku listy 180
8.1.2. Dodawanie i usuwanie elementów z końca listy 181
8.1.3. Dodawanie i usuwanie elementów z wnętrza listy 182
8.1.4. Zarządzanie pamięcią 183
8.2. Niemutowalne struktury danych podobne do wektorów 185
8.2.1. Wyszukiwanie elementu w drzewie trie 187
8.2.2. Dołączanie elementów w drzewie trie 189
8.2.3. Aktualizacja elementów w drzewie trie 191
8.2.4. Usuwanie elementów z końca drzewa trie 192
8.2.5. Inne operacje i całkowita wydajność drzewa trie 192
Podsumowanie 193

Rozdział 9. Algebraiczne typy danych i dopasowywanie do wzorców *195*

9.1. Algebraiczne typy danych 196
9.1.1. Typy sumy uzyskiwane poprzez dziedziczenie 197
9.1.2. Typy sumy uzyskiwane dzięki uniom i typowi std::variant 200
9.1.3. Implementacja określonych stanów 204
9.1.4. Szczególny typ sumy: wartości opcjonalne 205
9.1.5. Użycie typów sumy w celu obsługi błędów 207

Kup ksi k Pole ksi k

9.2. Modelowanie dziedziny za pomocą algebraicznych typów danych 212
9.2.1. Rozwiązanie najprostsze, które czasami jest niewystarczające 213
9.2.2. Rozwiązanie bardziej zaawansowane: podejście zstępujące 214
9.3. Lepsza obsługa algebraicznych typów danych za pomocą dopasowywania do wzorców 215
9.4. Zaawansowane dopasowywanie do wzorców za pomocą biblioteki Mach7 218
Podsumowanie 219

Rozdział 10. Monady 221

10.1. Funktory pochodzące od nie Twojego rodzica 222
10.1.1. Obsługa wartości opcjonalnych 223
10.2. Monady: więcej możliwości dla funktorów 226
10.3. Proste przykłady 229
10.4. Składane zakresy i monady 231
10.5. Obsługa błędów 234
10.5.1. Typ std::optional<T> jako monada 234
10.5.2. Typ expected<T, E> jako monada 236
10.5.3. Monada Try 237
10.6. Obsługa stanu przy użyciu monad 238
10.7. Monada współbieżnościowa i kontynuacyjna 240
10.7.1. Typ future jako monada 242
10.7.2. Implementacja wartości przyszłych 244
10.8. Składanie monad 245
Podsumowanie 247

Rozdział 11. Metaprogramowanie szablonów 249

11.1. Zarządzanie typami w czasie kompilacji 250
11.1.1. Debugowanie typów dedukowanych 252
11.1.2. Dopasowywanie do wzorców podczas kompilacji 254
11.1.3. Udostępnianie metainformacji o typach 257
11.2. Sprawdzanie właściwości typu w czasie kompilacji 258
11.3. Tworzenie funkcji rozwiniętych 261
11.3.1. Wywoływanie wszystkich obiektów wywoływalnych 263
11.4. Tworzenie języka dziedzinowego 265
Podsumowanie 271

Rozdział 12. Projektowanie funkcyjne systemów współbieżnych 273

12.1. Model aktora: myślenie komponentowe 274
12.2. Tworzenie prostego źródła wiadomości 278
12.3. Modelowanie strumieni reaktywnych w postaci monad 281
12.3.1. Tworzenie ujścia w celu odbierania wiadomości 283
12.3.2. Transformowanie strumieni reaktywnych 286
12.3.3. Tworzenie strumienia z określonymi wartościami 288
12.3.4. Łączenie strumienia strumieni 289
12.4. Filtrowanie strumieni reaktywnych 290
12.5. Obsługa błędów w strumieniach reaktywnych 291

Kup ksi k Pole ksi k

12.6. **Odpowiadanie klientowi** 293
12.7. **Tworzenie aktorów ze stanem mutowalnym** 297
12.8. **Tworzenie systemów rozproszonych z użyciem aktorów** 298
Podsumowanie 299

Rozdział 13. Testowanie i debugowanie 301

13.1. **Czy program, który się kompiluje, jest poprawny?** 302
13.2. **Testy jednostkowe i funkcje czyste** 304
13.3. **Testy generowane automatycznie** 305
13.3.1. Generowanie przypadków testowych 306
13.3.2. Testowanie oparte na właściwościach 307
13.3.3. Testy porównawcze 309
13.4. **Testowanie systemów współbieżnych opartych na monadach** 311
Podsumowanie 314

Skorowidz 315

Kup ksi k Pole ksi k

1

Wprowadzenie do programowania funkcyjnego

W niniejszym rozdziale omówiono następujące zagadnienia:

- Zrozumienie programowania funkcyjnego.
- Myślenie o celu zamiast o krokach algorytmu.
- Zrozumienie funkcji czystych.
- Korzyści z programowania funkcyjnego.
- Przekształcanie C++ w funkcyjny język programowania.

Będąc programistami, musimy w trakcie naszej kariery uczyć się wielu języków programowania. Jednakże zwykle koncentrujemy się tylko na dwóch lub trzech spośród nich, które nam najbardziej odpowiadają. Często słyszy się stwierdzenie, że nauka nowego języka programowania jest łatwa — różnice między językami występują głównie w składni, a większość z nich udostępnia mniej więcej te same funkcje. Jeśli znamy język C++, nauczenie się języka Java lub C# powinno być łatwe — i na odwrót.

To stwierdzenie ma pewne zalety. Jeśli jednak uczymy się nowego języka, zwykle próbujemy symulować styl programowania, którego używaliśmy w poprzednim. Gdy po raz pierwszy spotkałem się na mojej uczelni z funkcyjnym językiem programowania, zacząłem od nauczenia się, w jaki sposób można skorzystać z jego funkcji w celu

zasymulowania pętli `for` i `while` oraz rozgałęzień typu `if-then-else`. Takie podejście przyjęła większość z nas, by zdać egzamin i nigdy już nie wracać do poznanej wcześniej wiedzy.

Istnieje takie powiedzenie, że jeśli jedynym narzędziem, jakie masz, jest młotek, będziesz zachęcany, aby traktować każdy problem jak gwóźdź. Stosuje się je również w sytuacji odwrotnej: jeśli masz gwóźdź, będziesz chciał użyć dowolnego narzędzia w taki sposób, jakby było młotkiem. Wielu programistów, którzy chcą przetestować działanie funkcyjnego języka programowania, podejmuje decyzję, że nie warto się go uczyć, ponieważ nie widzą korzyści w jego zastosowaniu. Starają się oni używać nowego narzędzia w taki sam sposób, w jaki używali starego.

Dzięki tej książce nie nauczysz się nowego języka programowania, lecz poznasz alternatywny sposób używania znanego Ci języka (C++). Jest to jednak sposób tak różniący się od tego, co do tej pory poznałeś, że często będziesz **czuł się** tak, jakbyś poznawał i wykorzystywał nowy język. Dzięki temu nowemu stylowi programowania można tworzyć bardziej zwięzłe programy oraz kod, który jest bezpieczniejszy, łatwiejszy do zrozumienia i analizowania, a także — ośmielę się napisać — piękniejszy niż kod napisany w zwykły sposób w języku C++.

1.1. Co to jest programowanie funkcyjne?

Programowanie funkcyjne to dawny paradygmat programowania, który narodził się w środowisku akademickim w latach pięćdziesiątych XX wieku i przez długi czas pozostawał związany z tym środowiskiem. Chociaż zawsze był gorącym tematem dla badaczy naukowych, nigdy nie stał się popularny w „świecie realnym". Zamiast niego wszędzie zaczęły panować języki imperatywne (najpierw proceduralne, później zorientowane obiektowo).

Prognozuje się często, że pewnego dnia funkcyjne języki programowania będą rządzić światem, jednakże ten moment jeszcze nie nastąpił. Znane języki funkcyjne, takie jak Haskell i Lisp, nadal nie znajdują się na listach 10 najpopularniejszych języków programowania. Listy te są zarezerwowane dla tradycyjnie imperatywnych języków, takich jak C, Java i C++. Podobnie jak większość prognoz, również i ta musi zostać odpowiednio zinterpretowana, by mogła zostać uznana za spełnioną. Zamiast języków funkcyjnych, które stają się najbardziej popularne, dzieje się coś innego: najbardziej popularne języki programowania zaczynają wprowadzać funkcje inspirowane funkcjonalnymi językami programowania.

Co to **jest** programowanie funkcyjne? Na to pytanie trudno odpowiedzieć, ponieważ nie istnieje powszechnie przyjęta definicja. Jest takie powiedzenie, że jeśli zadasz powyższe pytanie dwóm programistom specjalizującym się w programowaniu funkcyjnym, otrzymasz (co najmniej) trzy różne odpowiedzi. Istnieje tendencja do definiowania programowania funkcyjnego poprzez związane z nim koncepcje, jak na przykład funkcje czyste, wartościowanie leniwe, dopasowywanie wzorców itp. Dana osoba zwykle wymienia cechy swojego ulubionego języka.

Kup ksi k Pole ksi k

Aby nikogo nie zrazić, zaczniemy od przesadnie matematycznej definicji, pochodzącej z grupy dyskusyjnej Usenet, dotyczącej programowania funkcjonalnego:

> *Programowanie funkcyjne to styl programowania, który kładzie nacisk na wyznaczanie wyrażeń, a nie na wykonywanie poleceń. Wyrażenia w tych językach są tworzone za pomocą funkcji i służą do łączenia wartości podstawowych. Język funkcyjny to język, który wspiera programowanie w stylu funkcyjnym i zachęca do niego.*
>
> — najczęściej zadawane pytania (FAQ), grupa dyskusyjna comp.lang.functional

W tej książce omówimy różne koncepcje związane z programowaniem funkcyjnym. Od Ciebie będzie zależało, jakie elementy uznasz za decydujące o tym, by język mógł być nazwany **funkcyjnym**.

Ogólnie mówiąc, programowanie funkcyjne to styl programowania, w którym główne elementy składowe programu są funkcjami, w przeciwieństwie do obiektów i procedur. Program napisany w stylu funkcyjnym nie zawiera poleceń, które należy wykonać, aby osiągnąć określony wynik, ale raczej definiuje, jaki powinien być ten wynik.

Rozważmy prosty przykład: obliczanie sumy listy liczb. W świecie imperatywnym implementujesz algorytm poprzez przetwarzanie listy i dodawanie liczb do zmiennej akumulacyjnej. Wyjaśniasz krok po kroku proces, w jaki sposób należy sumować listę liczb. Z drugiej strony, w stylu funkcyjnym musisz zdefiniować tylko to, co jest sumą listy liczb. Komputer wie, co należy zrobić, by wyznaczyć sumę. Jednym ze sposobów, w jaki możesz zaprezentować tę definicję, jest stwierdzenie, że suma listy liczb jest równa pierwszemu elementowi listy dodanemu do sumy reszty listy i że suma wynosi zero, jeśli lista jest pusta. Określasz sumę bez wyjaśniania, jak należy ją obliczyć.

Powyższa różnica w działaniu algorytmów jest źródłem powstania terminów *programowanie imperatywne* i *deklaratywne*. **Programowanie imperatywne** oznacza, że nakazujesz komputerowi zrobienie czegoś, wyraźnie określając każdy krok, który musi zostać zrealizowany w celu wyznaczenia wyniku. **Programowanie deklaratywne** oznacza, że podajesz, co należy zrobić, a zadaniem języka programowania jest dowiedzieć się, w jaki sposób można to zrealizować. Określasz, czym jest suma listy liczb, a język używa tej definicji do obliczenia sumy danej listy liczb.

1.1.1. Związek z programowaniem obiektowym

Nie można powiedzieć, co jest lepsze: najpopularniejszy paradygmat imperatywny, czyli programowanie obiektowe (OOP), czy najczęściej używany paradygmat deklaratywny — programowanie funkcyjne. Oba podejścia mają swoje zalety i wady.

Paradygmat obiektowy opiera się na tworzeniu abstrakcji dla danych. Pozwala programiście ukryć wewnętrzną reprezentację danych wewnątrz obiektu i umożliwić reszcie świata dostęp do nich jedynie za pośrednictwem interfejsu API.

Styl programowania funkcyjnego definiuje abstrakcje dotyczące funkcji. Pozwala to tworzyć struktury sterujące, które są bardziej złożone w porównaniu z językiem

Kup ksi k Pole ksi k

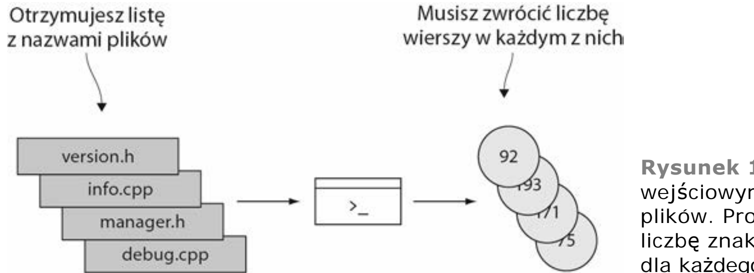
bazowym. Kiedy w języku C++ 11 wprowadzono pętlę `for` opartą na zakresach (nazwaną `foreach`), musiała ona zostać zaimplementowana w każdym kompilatorze C++ (a jest ich wiele). Korzystając z technik programowania funkcyjnego, można było to zrobić bez zmiany kompilatora. Wiele bibliotek zewnętrznych od dawna implementowało własne wersje pętli opartych na zakresach. Gdy używamy idiomów programowania funkcyjnego, możemy tworzyć nowe konstrukcje językowe, takie jak pętle `for` oparte na zakresach i inne, bardziej zaawansowane. Będą one użyteczne nawet podczas pisania programów w stylu imperatywnym.

W pewnych sytuacjach pierwszy paradygmat jest bardziej odpowiedni od drugiego — i na odwrót. Często najlepszym rozwiązaniem jest połączenie obu stylów. Wynika to z faktu, że wiele starych i nowych języków programowania stało się wielowymiarowymi zamiast bycia wiernymi swojemu podstawowemu paradygmatowi.

1.1.2. Praktyczny przykład porównania programowania imperatywnego i deklaratywnego

Aby zademonstrować różnicę między tymi dwoma stylami programowania, zacznijmy od prostego programu zaimplementowanego w stylu imperatywnym, a następnie przekonwertujmy go na funkcjonalny odpowiednik. Jednym ze sposobów pomiaru stopnia złożoności oprogramowania jest zliczanie wierszy kodu. Chociaż można debatować, czy jest to dobry wskaźnik, jest on doskonałym sposobem na wykazanie różnic między stylami imperatywnym i funkcyjnym.

Wyobraź sobie, że chcesz napisać funkcję, która pobiera listę plików i zlicza liczbę wierszy w każdym z nich (patrz rysunek 1.1). Aby przykład był jak najprostszy, policzysz w pliku tylko liczbę znaków nowego wiersza. Załóżmy również, że ostatni wiersz w pliku również kończy się takim znakiem.



Rysunek 1.1. Danymi wejściowymi programu jest lista plików. Program musi zwrócić liczbę znaków nowego wiersza dla każdego z plików

Myśląc w sposób imperatywny, mógłbyś wziąć pod uwagę następujące działania w celu rozwiązania problemu:

1. Otwieraj po kolei każdy z plików.
2. Zdefiniuj licznik do przechowywania liczby wierszy.
3. Odczytuj plik po jednym znaku naraz i zwiększaj licznik za każdym razem, gdy pojawi się znak nowego wiersza (`\n`).
4. Po dotarciu do końca pliku zapisz liczbę obliczonych wierszy.

Kup ksi k Pole ksi k

Poniższy listing 1.1 pozwala na czytanie z plików znak po znaku i zlicza liczbę znaków nowego wiersza.

Listing 1.1. Zliczanie liczby wierszy w sposób imperatywny

```
std::vector<int>
count_lines_in_files(const std::vector<std::string>& files)
{
    std::vector<int> results;
    char c = 0;
    for (const auto& file : files) {
        int line_count = 0;
        std::ifstream in(file);
        while (in.get(c)) {
            if (c == '\n') {
                line_count++;
            }
        }
        results.push_back(line_count);
    }
    return results;
}
```

Program wynikowy zawiera dwie zagnieżdżone pętle i kilka zmiennych służących do zachowania bieżącego stanu procesu. Chociaż przykład jest prosty, zawiera kilka miejsc, w których można popełnić błąd — niezainicjalizowaną (lub źle zainicjalizowaną) zmienną, nieprawidłowo zaktualizowany stan lub niewłaściwy warunek pętli. Kompilator zgłasza niektóre z tych błędów jako ostrzeżenia, lecz te, które zostaną przez niego pominięte, są zwykle trudne do wykrycia, ponieważ nasze mózgi działają w taki sposób, by je ignorować, podobnie jak w przypadku błędów w pisowni. Powinieneś spróbować napisać swój kod w sposób minimalizujący możliwość popełniania takich błędów.

Czytelnicy, którzy są bardziej zaawansowani w programowaniu przy użyciu języka C++, być może zauważyli, że zamiast „ręcznie" obliczać liczbę nowych wierszy, można było użyć standardowego algorytmu `std::count`. Język C++ zapewnia wygodny dostęp do abstrakcji takich jak iteratory strumieniowe, które umożliwiają traktowanie strumieni wejścia i wyjścia w sposób podobny do zwykłych kolekcji, na przykład list i wektorów. Możemy więc skorzystać z tych iteratorów w naszym algorytmie (listing 1.2).

Listing 1.2. Użycie algorytmu std::count w celu zliczania znaków nowego wiersza

```
int count_lines(const std::string& filename)
{
    std::ifstream in(filename);
    return std::count(
        std::istreambuf_iterator<char>(in),
        std::istreambuf_iterator<char>(),
        '\n');
}
std::vector<int>
count_lines_in_files(const std::vector<std::string>& files)
```

Zlicza znaki nowego wiersza od bieżącej pozycji w strumieniu aż do końca pliku

```
{
    std::vector<int> results;
    for (const auto& file : files) {
        results.push_back(count_lines(file));   <—— Zapisz wynik
    }
    return results;
}
```

Dzięki powyższemu rozwiązaniu nie interesujesz się tym, w jaki sposób powinien zostać zaimplementowany sposób liczenia. Po prostu stwierdzasz, że chcesz policzyć liczbę wierszy, które pojawiają się w danym strumieniu wejściowym. Taki sposób myślenia jest kluczowy podczas tworzenia programów w stylu funkcyjnym. Użyj abstrakcji, które pozwolą Ci zdefiniować **cel**, zamiast zastanawiać się, **jak** coś należy zrobić. Tak właśnie postępujemy w większości rozwiązań omawianych w tej książce. Dlatego też programowanie funkcyjne jest spokrewnione z programowaniem generycznym (szczególnie w języku C++) — oba pozwalają myśleć na wyższym poziomie abstrakcji w porównaniu z przyziemnym podejściem imperatywnego stylu programowania.

Język zorientowany obiektowo?

Uważam za zabawne stwierdzenie, które popiera większość programistów, że C++ jest językiem obiektowym. Jest ono nieprawdziwe, ponieważ prawie żaden element standardowej biblioteki języka programowania C++ (zwykle określanej jako **standardowa biblioteka szablonów** lub **STL**) nie wykorzystuje polimorfizmu opartego na dziedziczeniu, który stanowi sedno paradygmatu programowania obiektowego.

Biblioteka STL została stworzona przez Aleksandra Stiepanowa, zagorzałego krytyka programowania obiektowego. Chciał on stworzyć ogólną bibliotekę programistyczną i zrealizował swój zamiar przy użyciu systemu szablonów języka C++ połączonego z kilkoma technikami programowania funkcyjnego.

Jest to jeden z powodów, dla których w tej książce bardzo polegam na bibliotece STL. Nawet jeśli nie jest ona *właściwą* biblioteką do programowania funkcyjnego, modeluje wiele jego koncepcji, co czyni ją doskonałym punktem wyjścia umożliwiającym wkroczenie w świat programowania funkcjonalnego.

Zaletą tego rozwiązania jest to, że używasz mniej zmiennych zarządzających stanem programu, o które trzeba się martwić. Możesz więc definiować wysokopoziomowe cele, zamiast określać dokładne kroki, które należy podjąć, aby uzyskać wynik. Już nie obchodzi Cię, w jaki sposób zostało zrealizowane liczenie. Jedynym zadaniem funkcji `count_lines` jest konwertowanie jej danych wejściowych (nazwy pliku) na typ, który może zostać zrozumiany przez algorytm `std::count` (parę iteratorów strumieniowych).

Pójdźmy jeszcze dalej i zdefiniujmy cały algorytm w stylu funkcjonalnym (listing 1.3): określmy, **co** należy zrobić, zamiast tego, **jak** należy to zrobić. Pozostała jeszcze pętla `for` oparta na zakresie, która stosuje funkcję do wszystkich elementów w kolekcji i gromadzi wyniki. Jest to powszechnie stosowany wzorzec i należy się spodziewać, że język programowania obsługuje go w swojej standardowej bibliotece. W języku C++ jest nim algorytm `std::transform` (w innych językach jest on zwykle dostępny pod nazwą `map` lub `fmap`). Implementacja takiej samej logiki, jednak przy użyciu algorytmu `std::transform`, została przedstawiona w następnym listingu. Algorytm `std::transform`

przetwarza elementy kolekcji `files` jeden po drugim, przekształcając je za pomocą funkcji `count_lines` i przechowując wynikowe wartości w wektorze `results`.

Listing 1.3. Odwzorowywanie plików na liczbę wierszy przy użyciu algorytmu std::transform

```
std::vector<int>
count_lines_in_files(const std::vector<std::string>& files)
{
    std::vector<int> results(files.size());
    std::transform(files.cbegin(), files.cend(),     ← Określa, jakie elementy należy przekształcić
                   results.begin(),                  ← Gdzie przechowywać wyniki
                   count_lines);                     ← Funkcja transformacji

    return results;
}
```

Ten kod nie określa już kroków algorytmu, które należy wykonać, ale raczej sposób transformacji danych wejściowych w celu uzyskania pożądanego wyniku. Można wnioskować, że usunięcie zmiennych stanu i wykorzystanie implementacji algorytmu liczenia zdefiniowanego w bibliotece standardowej zamiast tworzenia własnego sprawia, iż kod jest mniej podatny na błędy.

Problem polega na tym, że najnowszy listing zawiera zbyt dużo standardowego kodu i dlatego nie może być uważany za bardziej czytelny od przykładu pierwotnego. Główna funkcja wykorzystuje tylko trzy ważne słowa:

- `transform` — to, co jest realizowane w kodzie,
- `files` — dane wejściowe,
- `count_lines` — funkcja transformacji.

Reszta jest dodatkiem.

Nasza funkcja byłaby znacznie czytelniejsza, gdybyś mógł zapisywać tylko ważne fragmenty kodu i pomijać pozostałe. W rozdziale 7. zobaczysz, że jest to możliwe dzięki bibliotece zakresów. W tym miejscu zaprezentuję tylko, w jaki sposób zmienia się funkcja po zaimplementowaniu zakresów i ich transformacji. Zakresy używają operatora | (potok), oznaczającego przetwarzanie kolekcji przez transformację (listing 1.4).

Listing 1.4. Transformacja przy użyciu zakresów

```
std::vector<int>
count_lines_in_files(const std::vector<std::string>& files)
{
    return files | transform(count_lines);
}
```

Powyższy kod wykonuje to samo co listing 1.3, lecz jest łatwiejszy do zrozumienia. Pobierasz listę wejściową, przekazujesz ją do transformacji i zwracasz wynik.

Ta forma zapisu ulepsza także zarządzanie kodem. Być może zauważyłeś, że funkcja `count_lines` ma wadę konstrukcyjną. Jeśli spojrzysz tylko na jej nazwę i typ (`count_lines: std::string -> int`), zobaczysz, że funkcja przyjmuje łańcuch, ale nie jest

Notacja w celu określenia typu funkcji

Język C++ nie używa jednego typu w celu reprezentowania funkcji (w rozdziale 3. zapoznasz się ze wszystkimi elementami, które C++ uważa za podobne do funkcji). Aby określić typy argumentów i typ zwracany przez funkcję bez dokładnego wskazania, jaki typ będzie ona miała w języku C++, musimy wprowadzić nową notację, niezależną od języka.

Zapis `f: (arg1_t, arg2_t, ..., argn_t) -> wynik_t` oznacza, że funkcja `f` przyjmuje *n* argumentów, gdzie `arg1_t` jest typem pierwszego argumentu, `arg2_t` jest typem drugiego itd. Funkcja `f` zwraca wartość typu `wynik_t`. Jeśli funkcja przyjmuje tylko jeden argument, pomijamy nawiasy wokół jego typu. W celu uproszczenia w tym zapisie unikamy również używania słów `const`.

Na przykład jeśli stwierdzimy, że funkcja `repeat` ma typ `(char, int) -> std::string`, oznacza to, że funkcja przyjmuje dwa argumenty — jeden znak i jedną liczbę całkowitą — a zwraca łańcuch. W języku C++ zostanie to zapisane w następujący sposób (druga wersja jest dostępna od wersji C++ 11):

```
std::string repeat(char c, int count);
auto repeat(char c, int count) -> std::string;
```

jasne, czy ten ciąg znaków reprezentuje nazwę pliku. Byłoby rzeczą normalną oczekiwać, że funkcja zlicza liczbę wierszy w podanym łańcuchu. Aby rozwiązać ten problem, możesz podzielić funkcję na dwie: pierwszą `open_file: std::string -> std::if-stream`, która pobiera nazwę pliku i zwraca strumień pliku; oraz drugą `count_lines: std::ifstream -> int`, która zlicza liczbę wierszy w danym strumieniu. Dzięki tej zmianie jest oczywiste, czego dotyczą nazwy argumentów i użyte typy. Zmiana funkcji `count_lines_in_files` opartej na zakresach wymaga dodania tylko jednej dodatkowej transformacji (listing 1.5).

Listing 1.5. Zmodyfikowana transformacja wykorzystująca zakresy

```
std::vector<int>
count_lines_in_files(const std::vector<std::string>& files)
{
    return files | transform(open_file)
                 | transform(count_lines);
}
```

To rozwiązanie jest dużo bardziej zwięzłe niż algorytm imperatywny z listingu 1.1 i znacznie prostsze do zrozumienia. Zaczynasz od zbioru nazw plików (nie ma znaczenia, jakiego typu kolekcji używasz), a następnie wykonujesz dwie transformacje dla każdego elementu z tej kolekcji. Najpierw pobierasz nazwę pliku i tworzysz strumień na jej podstawie, a w dalszej kolejności przetwarzasz go, aby zliczyć znaki nowego wiersza. Dokładnie to reprezentuje powyższy kod — bez zbędnej składni i nadmiarowej treści.

1.2. Funkcje czyste

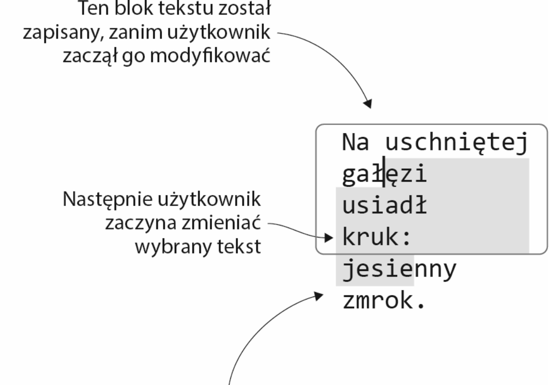
Jednym z najbardziej znaczących źródeł błędów w oprogramowaniu jest stan programu. Trudno jest śledzić wszystkie możliwe stany, w których program może się znajdować. Paradygmat programowania obiektowego daje możliwość umieszczania części stanów

Kup ksi k Pole ksi k

w obiektach, co ułatwia zarządzanie. Nie zmniejsza to jednak znacząco liczby możliwych stanów.

Załóżmy, że tworzysz edytor tekstu, a w zmiennej przechowujesz tekst napisany przez użytkownika. W pewnym momencie użytkownik klika przycisk *Zapisz*, a następnie kontynuuje swoją pracę. Program zapisuje tekst na nośniku, wysyłając na niego po jednym znaku naraz (jest to nieco uproszczone podejście, ale proszę o chwilę cierpliwości). Co się stanie, gdy użytkownik zmieni część tekstu w czasie trwania zapisu na dysk? Czy program zapisze tekst taki, jaki był w momencie, gdy użytkownik kliknął przycisk *Zapisz*, czy też zapisze bieżącą wersję, a może zrobi coś innego?

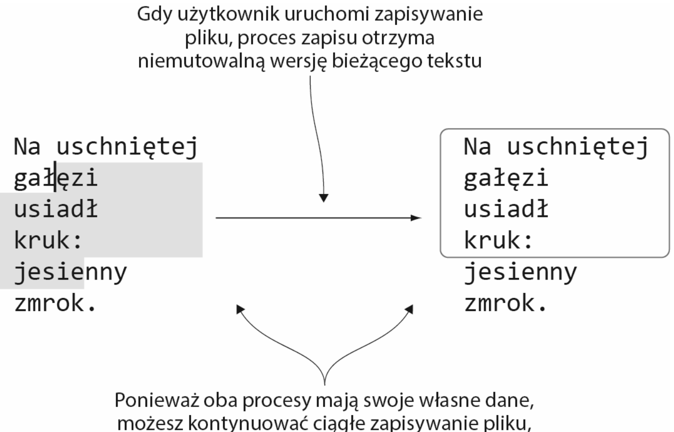
Problem polega na tym, że wszystkie trzy przypadki są możliwe. Odpowiedź na zadane pytanie będzie zależeć od postępu operacji zapisu i od tego, która część tekstu uległa zmianie. W przypadku przedstawionym na rysunku 1.2 program zapisze tekst, który nigdy nie był w edytorze.



Rysunek 1.2. Jeśli zezwolisz użytkownikowi na modyfikowanie tekstu podczas jego zapisywania, mogą zostać zapamiętane niepełne lub nieprawidłowe dane, tworząc uszkodzony plik

Niektóre fragmenty zapisanego pliku będą pochodzić z tekstu przed zmianą, a inne będą zawierać tekst po modyfikacji. Dane z dwóch różnych stanów zostaną zapisane w tym samym czasie.

Problem ten nie istniałby, gdyby funkcja zapisująca posiadała własną, niemutowalną kopię danych, które powinna zapisać (patrz rysunek 1.3). Największym problemem stanu mutowalnego jest to, że tworzy zależności między częściami programu, które nie muszą mieć ze sobą nic wspólnego. Powyższy przykład obejmuje dwie wyraźnie oddzielne czynności użytkownika: zapamiętywanie wprowadzanego tekstu i jego wpisywanie. Powinny być one możliwe do wykonania niezależnie od siebie. Istnienie wielu działań, które mogą być wykonywane w tym samym czasie i które współdzielą stan mutowalny, tworzy zależność między nimi i powoduje powstanie problemów podobnych do tych właśnie opisanych.



Rysunek 1.3. Jeśli utworzysz pełną kopię lub użyjesz struktury, która może zapamiętać w tym samym czasie wiele wersji danych, będziesz mógł rozdzielić procesy zapisywania pliku i zmiany tekstu w edytorze tekstowym

Michael Feathers, autor książki *Praca z zastanym kodem. Najlepsze techniki* (Helion, 2017), powiedział: „Programowanie obiektowe sprawia, że kod jest zrozumiały dzięki hermetyzacji zmieniających się elementów. Programowanie funkcyjne sprawia, że kod jest zrozumiały poprzez minimalizację zmieniających się elementów". Wynika z tego, że nawet lokalne zmienne mutowalne mogą być uznane za nieodpowiednie. Tworzą one zależności między różnymi częściami funkcji, utrudniając jej podział na kilka mniejszych.

Jedną z najpotężniejszych idei zawartych w programowaniu funkcyjnym są **funkcje czyste** (ang. *pure functions*): to funkcje, które wykorzystują (ale nie modyfikują) argumenty przekazane im w celu obliczenia wyniku. Jeśli funkcja czysta jest wywoływana wiele razy z tymi samymi argumentami, musi zawsze zwracać ten sam wynik i nie pozostawiać śladu, że została kiedykolwiek wywołana (brak **efektów ubocznych**). To wszystko oznacza, że funkcje czyste nie są w stanie zmienić stanu programu.

Jest to świetny pomysł, ponieważ nie musisz myśleć o stanie programu. Niestety, oznacza to również, że funkcje czyste nie mogą czytać ze standardowego wejścia, zapisywać do standardowego wyjścia, tworzyć ani usuwać plików, wstawiać wierszy do bazy danych itd. Gdybyśmy chcieli przesadnie egzekwować niemutowalność, musielibyśmy nawet zabraniać funkcjom czystym zmieniania rejestrów procesora, komórek pamięci lub czegokolwiek innego znajdującego się na poziomie sprzętu.

Sprawia to, że podana przed chwilą definicja funkcji czystych staje się bezużyteczna. Procesor wykonuje instrukcje jedną po drugiej i musi śledzić, która z ich powinna zostać wykonana w następnej kolejności. Nie można wykonać niczego w komputerze bez zmieniania przynajmniej wewnętrznego stanu procesora. Ponadto nie można pisać przydatnych programów, jeśli nie można się komunikować z użytkownikiem lub innym oprogramowaniem.

Z tego powodu zmniejszymy wymagania i udoskonalimy naszą definicję: **funkcją czystą** jest każda funkcja, która nie wykazuje widocznych (na poziomie wyższym) efektów ubocznych. Kod wywołujący funkcję nie powinien zauważyć żadnego śladu, że funkcja została wykonana, poza uzyskaniem wyniku wywołania. Nie będziemy ograniczać

Kup ksi k Pole ksi k

się do używania i tworzenia wyłącznie funkcji czystych, ale spróbujemy ograniczyć liczbę nieczystych, z których korzystamy.

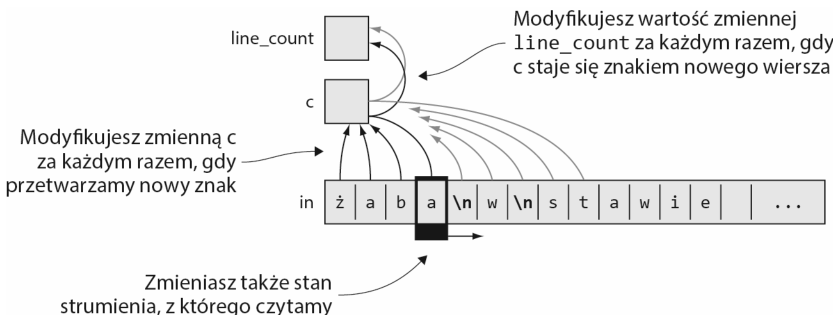
1.2.1. Unikanie stanu mutowalnego

Zaczęliśmy mówić o stylu programowania funkcyjnego, rozważając imperatywną implementację algorytmu, który zlicza znaki nowego wiersza w zbiorze plików. Funkcja, która zlicza znaki nowego wiersza, powinna zawsze zwracać tę samą tablicę liczb całkowitych, gdy zostanie wywołana z tą samą listą plików (pod warunkiem, że pliki nie zostały zmienione przez obiekt zewnętrzny). Oznacza to, że funkcja może być zaimplementowana jako funkcja czysta.

Przyglądając się początkowej implementacji tej funkcji z listingu 1.1, możemy zauważyć kilka instrukcji, które są nieczyste:

```
for (const auto& file: files) {
    int line_count = 0;
    std::ifstream in(file);
    while (in.get(c)) {
        if (c == '\n') {
            line_count++;
        }
    }
    results.push_back(line_count);
}
```

Wywołanie metody `.get` dla strumienia wejściowego powoduje jego zmianę, a także modyfikuje wartość zapisaną w zmiennej `c`. Kod zmienia tablicę `results`, dodając do niej nowe wartości, i modyfikuje wartość zmiennej `line_count` poprzez jej inkrementację (na rysunku 1.4 pokazano zmiany stanu podczas przetwarzania pojedynczego pliku). Ta funkcja zdecydowanie nie jest zaimplementowana w czysty sposób.



Rysunek 1.4. Podczas wyznaczania liczby znaków nowego wiersza występujących w jednym pliku modyfikowanych jest kilka niezależnych zmiennych. Pewne zmiany są zależne od innych, a niektóre nie są zależne

Nie jest to jednak jedyne pytanie, które musisz sobie zadać. Inną ważną kwestią jest to, czy zanieczyszczenia danej funkcji są obserwowalne z zewnątrz. Wszystkie zmienne mutowalne w tej funkcji są lokalne — nie są nawet współdzielone między możliwymi współbieżnymi wywołaniami funkcji — i nie są widoczne dla kodu wywołującego ani

Kup ksi k Pole ksi k

żadnego zewnętrznego obiektu. Użytkownicy tej funkcji mogą uważać ją za czystą, nawet jeśli jej implementacja taka nie jest. Przynosi to korzyści dla kodu wywołującego, ponieważ może on polegać na tym, że nie zmieniasz stanu funkcji, choć wciąż musisz zarządzać swoim własnym stanem. Czyniąc to, powinieneś upewnić się, że nie zmieniasz niczego, co nie należy do Ciebie. Naturalnie byłoby lepiej, gdybyś ograniczył zmienne stanu i starał się, by implementacja funkcji była jak najbardziej czysta. Jeśli upewnisz się, że w swojej implementacji używasz tylko funkcji czystych, nie będziesz musiał się zastanawiać, czy nie masz do czynienia z żadnymi zmianami stanu, ponieważ nic nie modyfikujesz.

Drugie rozwiązanie (listing 1.2) umieszcza proces zliczania w funkcji o nazwie `count_lines`. Ta funkcja również wygląda z zewnątrz jak czysta, nawet jeśli wewnętrznie deklaruje strumień wejściowy i modyfikuje go. Niestety, ze względu na interfejs programowy klasy `std::ifstream`, jest to najlepsze rozwiązanie, jakie możesz uzyskać:

```
int count_lines(const std::string& filename)
{
    std::ifstream in(filename);
    return std::count(
        std::istreambuf_iterator<char>(in),
        std::istreambuf_iterator<char>(),
        '\n');
}
```

Na tym etapie nie poprawiamy funkcji `count_lines_in_files` w jakikolwiek znaczący sposób. Przenosimy jedynie część zanieczyszczeń do innego miejsca i zachowujemy dwie zmienne mutowalne. W przeciwieństwie do `count_lines`, funkcja `count_lines_in_files` nie obsługuje operacji wejścia i wyjścia. Została ona również zaimplementowana tylko w powiązaniu z funkcją `count_lines`, którą Ty (tworząc kod wywołujący) możesz uznać za czystą. Nie ma powodu, by zawierała jakieś nieczyste fragmenty. Kolejna wersja kodu, która korzysta z notacji zakresów, implementuje funkcję `count_lines_in_files` bez wykorzystania żadnego lokalnego stanu (mutowalnego lub niemutowalnego). Cała funkcja została zdefiniowana z punktu widzenia wywołań innej funkcji dla określonych danych wejściowych:

```
std::vector<int>
count_lines_in_files(const std::vector<std::string>& files)
{
    return files | transform(count_lines);
}
```

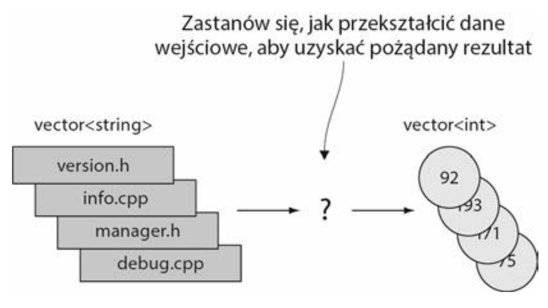
Przedstawione rozwiązanie jest doskonałym przykładem stylu programowania funkcyjnego. Jest ono krótkie i zwięzłe, a jego działanie jest oczywiste. Co więcej, sprawą oczywistą jest, że kod nie wykonuje żadnych innych czynności — nie ma widocznych efektów ubocznych. Po prostu zwraca pożądane dane wyjściowe dla określonych danych wejściowych.

Kup ksi k Pole ksi k

1.3. Myślenie w sposób funkcjonalny

Tworzenie kodu najpierw w stylu imperatywnym, a następnie stopniowe zmienianie go, aż stanie się funkcyjny, byłoby działaniem nieefektywnym i nieproduktywnym, dlatego też powinieneś w inny sposób zastanawiać się nad rozwiązaniem problemów. Zamiast myśleć o krokach algorytmu, powinieneś rozważyć, czym są dane wejściowe, jakie powinny być wyniki i jakie transformacje należy wykonać, aby odwzorować wejście na wyjście.

Na rysunku 1.5 zaprezentowano listę nazw plików, a Twoim zadaniem jest obliczenie liczby wierszy znajdujących się w każdym z nich. Pierwszą rzeczą, którą powinno się wziąć pod uwagę, jest to, że można uprościć ten problem, przetwarzając w danym momencie tylko pojedynczy plik. Masz listę nazw plików, ale możesz przetwarzać każdy z nich niezależnie od pozostałych. Jeśli możesz znaleźć sposób rozwiązania problemu dla pojedynczego pliku, możesz także łatwo rozwiązać zadanie pierwotne (rysunek 1.6).



Rysunek 1.5. Myśląc funkcjonalnie, rozważasz transformacje, które musisz zastosować dla danych wejściowych, aby uzyskać pożądany wynik w danych wyjściowych

Upraszczanie problemu poprzez określanie wymaganej transformacji tylko dla pojedynczego elementu z danego zbioru

string

manager.h → ? → 171

int

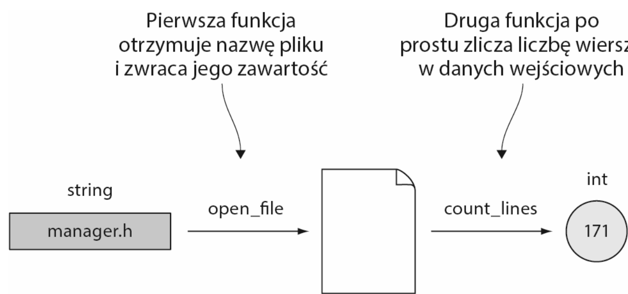
Rysunek 1.6. Tę samą transformację możesz wykonać dla każdego elementu z kolekcji. Pozwala to zająć się prostszym problemem transformacji pojedynczego elementu zamiast całej ich kolekcji

Obecnie głównym problemem jest zdefiniowanie funkcji, która pobiera nazwę pliku i wyznacza liczbę wierszy w pliku reprezentowanym przez tę nazwę. Wynika z tego, że otrzymujesz pewną daną (nazwę pliku), lecz potrzebujesz czegoś innego (zawartości pliku, by można było policzyć liczbę znaków nowego wiersza). Dlatego potrzebna jest funkcja, która może zwrócić zawartość pliku, jeśli zostanie podana jego nazwa. Od Ciebie zależy, czy zawartość ma zostać zwrócona jako ciąg znaków, strumień plikowy, czy jeszcze coś innego. Kod musi po prostu być w stanie udostępniać tylko jeden znak naraz, aby można było go przekazać do funkcji, która zlicza liczbę wierszy.

Gdy masz już funkcję, która zwraca zawartość pliku (`std::string` → `std::ifstream`), możesz wywołać inną funkcję, która zlicza wiersze na podstawie otrzymanego wcześniej wyniku (`std::if-stream` → `int`). Złożenie tych dwóch funkcji poprzez przekazanie

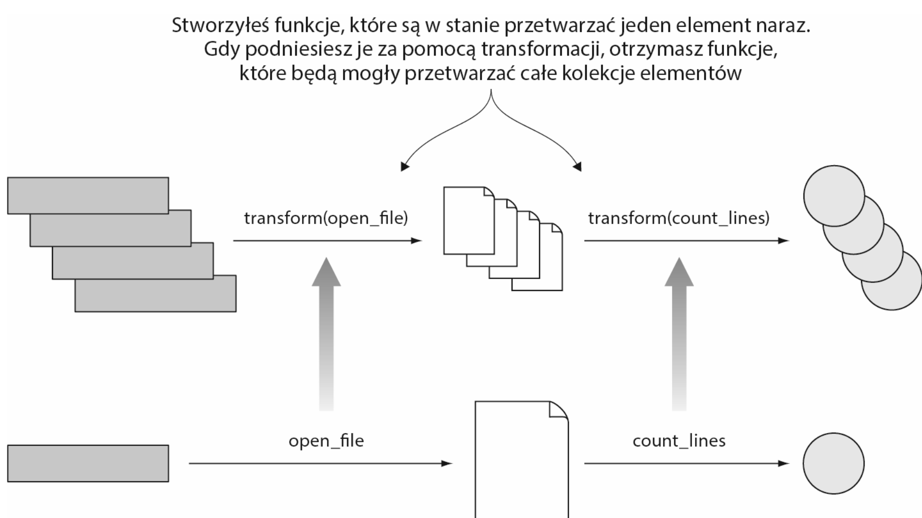
Kup ksi k Pole ksi k

strumienia `ifstream` utworzonego przez pierwszą z nich jako danej wejściowej dla drugiej daje pożądaną funkcjonalność (patrz rysunek 1.7).



Rysunek 1.7. Możesz podzielić większy problem, polegający na zliczaniu liczby wierszy w pliku o znanej nazwie, na dwa mniejsze problemy: otwieranie pliku, mając podaną jego nazwę, a następnie zliczanie w nim liczby wierszy

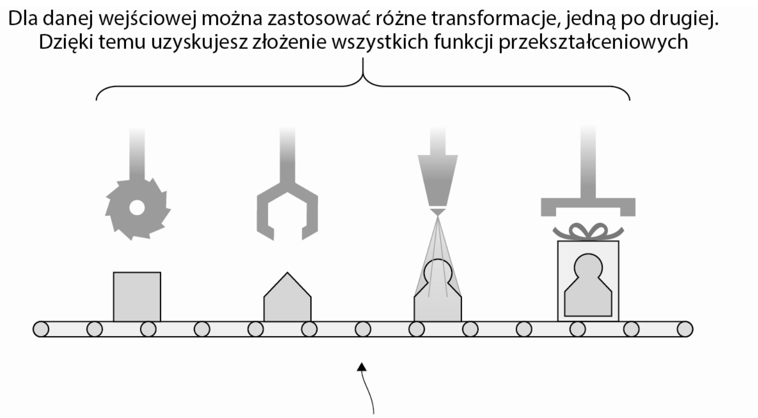
Dzięki temu pomysłowi rozwiązałeś problem. Musisz *podnieść* dwie funkcje, aby móc obsługiwać nie tylko pojedynczą wartość, lecz także zbiór tych wartości. Jest to koncepcyjnie równoznaczne z tym, co realizuje algorytm `std::transform` (z bardziej skomplikowanym interfejsem API): przyjmuje funkcję, która może zostać użyta z pojedynczą wartością, a następnie tworzy transformację, która może działać na całym zbiorze wartości (patrz rysunek 1.8). Na razie traktuj **podnoszenie** (ang. *lifting*) jako ogólny sposób przekształcania funkcji, które wykorzystują prosty typ danych, do funkcji obsługujących bardziej złożone struktury danych zawierające wartości tego typu. Podnoszenie zostanie dokładniej omówione w rozdziale 4.



Rysunek 1.8. Za pomocą algorytmu transform można tworzyć funkcje, które potrafią przetwarzać kolekcje elementów pochodzących z funkcji mogących przetwarzać tylko jeden element naraz

Na tym prostym przykładzie zaprezentowano funkcyjne podejście pozwalające na dzielenie większych problemów programistycznych na mniejsze, niezależne zadania, które łatwo można złożyć. Przydatną analogią związaną ze złożeniem i podniesieniem funkcji

jest ruchoma linia montażowa (rysunek 1.9). Na początku mamy surowiec, z którego zostanie wytworzony produkt końcowy. Ten materiał przechodzi przez maszyny, które go przekształcają, by wreszcie otrzymać produkt finalny. Dzięki linii montażowej myślisz o transformacjach, przez które przechodzi produkt, zamiast o działaniach, które musi wykonać maszyna.



Rysunek 1.9. Składanie i podnoszenie funkcji można porównać do ruchomej linii montażowej. Różne transformacje działają z pojedynczymi elementami. Podnosząc te transformacje, by mogły przetwarzać kolekcje elementów, i składając je w taki sposób, aby wynik jednej transformacji został przekazany do następnej, otrzymasz linię montażową, która stosuje serię transformacji dla dowolnej liczby elementów

W tym przypadku surowiec jest daną wejściową, którą otrzymujesz, a maszyny są funkcjami zastosowanymi dla tej danej. Każda funkcja jest wysoce wyspecjalizowana w wykonywaniu jednego prostego zadania i nie „interesuje" jej reszta linii montażowej. Wymaga ona tylko prawidłowych danych wejściowych; nie obchodzi jej jednak, skąd one pochodzą. Elementy wejściowe są umieszczane jeden po drugim na linii montażowej (możesz także mieć wiele linii montażowych, które umożliwiają równoległe przetwarzanie większej liczby elementów). Każda dana jest transformowana i w rezultacie otrzymujesz kolekcję przekształconych elementów.

1.4. Korzyści wynikające z programowania funkcyjnego

Różne aspekty programowania funkcyjnego zapewniają uzyskiwanie różnych korzyści. Omówimy je we właściwym czasie; zacznijmy jednak od przedstawienia kilku podstawowych korzyści, które chcielibyśmy uzyskać w większości przypadków.

Najbardziej oczywistą sprawą, którą większość osób zauważa po rozpoczęciu wdrażania programów w stylu funkcjonalnym, jest to, że kod staje się znacznie krótszy. Niektóre projekty nawet zawierają w kodzie oficjalne adnotacje, takie jak: „Mógłby to być jeden wiersz w języku Haskell". Dzieje się tak, ponieważ narzędzia oferowane przez programowanie funkcyjne są proste, lecz jednocześnie bardzo ekspresyjne, a większość

Kup ksi k Pole ksi k

funkcjonalności można zaimplementować na wyższym poziomie bez przejmowania się kłopotliwymi szczegółami.

Ta cecha w połączeniu z czystością sprawiła, że styl programowania funkcyjnego stał się w ostatnich latach przedmiotem zainteresowania. Czystość podnosi poziom poprawności kodu, a ekspresyjność pozwala tworzyć zwięzłe programy (w których możesz popełniać mniej błędów).

1.4.1. Zwięzłość i czytelność kodu

Osoby, które programują w stylu funkcyjnym, twierdzą, że łatwiej jest zrozumieć tworzone przez nie programy. Jest to ocena subiektywna, a programiści, którzy są przyzwyczajeni do pisania i czytania kodu imperatywnego, mogą się z nią nie zgodzić. Obiektywnie można powiedzieć, że programy napisane w stylu funkcyjnym są krótsze i bardziej treściwe. Było to widoczne we wcześniejszym przykładzie: rozpoczynaliśmy od 20 wierszy kodu, a skończyliśmy na pojedynczym wierszu dla funkcji `count_lines_in_files` i około 5 wierszach dla funkcji `count_lines`, która zawierała przede wszystkim treści narzucone przez C++ i STL. Osiągnięcie tego celu było możliwe dzięki wykorzystaniu abstrakcji wyższego poziomu dostarczonych przez funkcyjne składniki biblioteki STL.

Jedną z przykrych prawd jest to, że wielu programistów języka C++ nie korzysta z abstrakcji wyższego poziomu, takich jak algorytmy biblioteki STL. Mają oni ku temu różne powody, poczynając od samodzielnego pisania bardziej wydajnego kodu, a kończąc na unikaniu tworzenia programu, którego nie będą mogli zrozumieć ich współpracownicy. Przyczyny te są czasem rzeczywiście ważne, ale nie jest tak w większości przypadków. Brak korzystania z bardziej zaawansowanych funkcji języka programowania, którego używasz, zmniejsza jego możliwości i ekspresyjność oraz sprawia, że kod staje bardziej złożony i trudniejszy w utrzymaniu.

W 1987 roku Edsger Dijkstra opublikował artykuł zatytułowany „Instrukcja GOTO powinna być uważana za szkodliwą”. Opowiadał się za odrzuceniem instrukcji `GOTO`, która w tamtym okresie była nadużywana, na rzecz programowania strukturalnego i używania konstrukcji wyższego poziomu, w tym procedur, pętli i rozgałęzień `if-then-else`:

> *Nadmierne użycie instrukcji `GOTO` powoduje, że bardzo trudno znaleźć sensowny zestaw parametrów opisujących postęp procesu. Instrukcja `GOTO` w obecnym stanie jest po prostu zbyt prymitywna; staje się ona wielkim zaproszeniem do robienia bałaganu w programie.*[1]

W wielu przypadkach również pętle i rozgałęzienia są zbyt prymitywne. Podobnie jak instrukcja `GOTO`, także pętle i rozgałęzienia mogą sprawić, że programy będą trudniejsze do napisania i zrozumienia. Często można je zastąpić konstrukcjami wyższego poziomu należącymi do programowania funkcyjnego. Ten sam kod jest niejednokrotnie używany w wielu miejscach, a programiści nawet tego nie zauważają, ponieważ

[1] „Communications of the ACM”, 11, nr 3 (marzec 1968).

Kup ksi k Pole ksi k

działa on z różnymi typami lub ma odmienne zachowanie, którego można po prostu nie brać pod uwagę.

Korzystając z istniejących abstrakcji dostarczonych przez STL lub bibliotekę zewnętrzną, a także tworząc własne, możesz uczynić swój kod bezpieczniejszym i krótszym. Ułatwisz także wyszukiwanie błędów w tych abstrakcjach, ponieważ ten sam kod zostanie wykorzystany w wielu miejscach.

1.4.2. Współbieżność i synchronizacja

Głównym problemem podczas tworzenia systemów współbieżnych jest współdzielony stan zmienny. Szczególnej uwagi wymaga upewnienie się, że komponenty nie przeszkadzają sobie nawzajem.

Zrównoleglenie programów wykorzystujących funkcje czyste jest operacją banalną, ponieważ funkcje te niczego nie modyfikują. Nie ma potrzeby realizowania jawnej synchronizacji przy użyciu operacji niepodzielnych lub muteksów. Kod napisany dla systemu jednowątkowego można uruchamiać w wielu wątkach bez prawie żadnych zmian. Więcej na ten temat dowiesz się w rozdziale 12.

Rozważ następujący fragment kodu, który w wektorze `xs` sumuje pierwiastki kwadratowe wartości:

```
std::vector<double> xs = {1.0, 2.0, ...};
auto result = sum(xs | transform(sqrt));
```

Jeśli implementacja funkcji `sqrt` jest czysta (a nie ma żadnego powodu, by tak nie sądzić), algorytm sumowania może automatycznie podzielić dane wejściowe na porcje i obliczyć dla nich sumy częściowe w oddzielnych wątkach. Kiedy wszystkie wątki się zakończą, wystarczy zebrać wyniki i je podsumować.

Niestety, język C++ nie zna (jeszcze) pojęcia funkcji czystej, więc współbieżność nie może być realizowana w sposób automatyczny. Zamiast tego musisz jawnie wywołać równoległą wersję algorytmu `sum`. Funkcja `sum` może nawet w trakcie działania być w stanie wykryć liczbę rdzeni procesora i wykorzystać tę informację przy podejmowaniu decyzji, na ile fragmentów należy podzielić wektor `xs`. Jeśli napisałeś powyższy kod przy użyciu pętli `for`, nie można go uwspółbieżnić w prosty sposób. Zamiast pozostawić podejmowanie decyzji bibliotece udostępniającej algorytm sumujący, musisz się zastanowić, czy zmienne nie zostały zmodyfikowane w tym samym czasie przez różne wątki, a następnie wygenerować optymalną liczbę wątków dla systemu, w którym uruchomisz program.

UWAGA Po rozpoznaniu, że ciała pętli są czyste, kompilatory C++ mogą czasami wykonywać automatyczną wektoryzację lub inne optymalizacje. Te optymalizacje wpływają wówczas również na kod, który wykorzystuje standardowe algorytmy, ponieważ są one zazwyczaj wewnętrznie implementowane za pomocą pętli.

1.4.3. Ciągła optymalizacja

Używanie abstrakcji programowania wyższego poziomu pochodzących z STL lub innych zaufanych bibliotek ma inną wielką zaletę: Twój program będzie się stawał coraz lepszy, nawet jeśli nie zmienisz w nim żadnej instrukcji. Każde ulepszenie języka

Kup ksi k Pole ksi k

programowania, implementacja kompilatora lub używanej biblioteki poprawi również sam program. Chociaż to stwierdzenie dotyczy zarówno funkcyjnych, jak i niefunkcyjnych abstrakcji wyższego poziomu, użycie programowania funkcyjnego znacznie zwiększa ilość kodu, który może je wykorzystywać.

Wydaje się to oczywiste, ale wielu programistów woli samodzielnie tworzyć niskopoziomowy kod o *wysokim poziomie wydajności*, czasami nawet w asemblerze. Takie podejście może przynieść korzyści, ale w większości przypadków po prostu optymalizuje kod dla konkretnej platformy docelowej i uniemożliwia kompilatorowi zoptymalizowanie go dla innej.

Rozważmy funkcję `sum`. Możesz ją zoptymalizować pod kątem systemu, który wstępnie wczytuje instrukcje, i sprawić, że wewnętrzna pętla będzie używać w każdej iteracji dwóch elementów lub ich większej liczby zamiast sumować liczby po kolei. Zmniejszy to liczbę skoków w kodzie, więc procesor będzie częściej pobierał poprawne instrukcje. To oczywiście poprawiłoby wydajność platformy docelowej. Ale co się stanie, jeśli uruchomisz ten sam program na innej platformie? W przypadku niektórych platform optymalna będzie pętla oryginalna; dla innych lepszym rozwiązaniem będzie sumowanie większej liczby pozycji przy każdej iteracji pętli. Niektóre systemy mogą nawet udostępnić określoną instrukcję procesora, która wykona dokładnie to, czego potrzebuje funkcja.

Samodzielnie optymalizując kod w powyższy sposób, nie zrealizujesz zamierzonego zadania dla wszystkich platform oprócz jednej. Jeśli używasz abstrakcji wyższego poziomu, polegasz na innych programistach, którzy tworzą kod zoptymalizowany. Większość implementacji STL zapewnia istnienie określonych optymalizacji dla platform i używanych na nich kompilatorów.

1.5. Przekształcanie C++ w funkcyjny język programowania

C++ narodził się jako rozszerzenie języka programowania C w celu umożliwienia programistom tworzenia kodu zorientowanego obiektowo (początkowo nazywał się on „C z klasami”). Nawet w jego pierwszej standardowej wersji (C++ 98) trudno było go nazwać językiem **zorientowanym obiektowo**. Dzięki wprowadzeniu szablonów i utworzeniu biblioteki STL, która rzadko korzysta z dziedziczenia i metod wirtualnych, język C++ stał się językiem multiparadygmatycznym.

Biorąc pod uwagę projekt i implementację biblioteki STL, można nawet argumentować, że C++ nie jest przede wszystkim językiem obiektowym, lecz językiem programowania generycznego. **Programowanie generyczne** opiera się na założeniu, że można stworzyć kod, który wykorzystuje ogólne pojęcia, a następnie zastosować go do dowolnej struktury, która pasuje do tych pojęć. Biblioteka STL, przykładowo, udostępnia szablon `vector`, którego można używać z różnymi typami, takimi jak liczby całkowite, łańcuchy oraz typy zdefiniowane przez użytkowników spełniające określone warunki wstępne. Kompilator generuje następnie zoptymalizowany kod dla każdego z określonych typów. Taką funkcjonalność nazywa się zwykle **polimorfizmem sta-**

Kup ksi k Pole ksi k

tycznym lub **polimorfizmem czasu kompilacji**, w przeciwieństwie do **polimorfizmu dynamicznego** lub **polimorfizmu czasu wykonania** dostępnego przez dziedziczenie i metody wirtualne.

W przypadku programowania funkcyjnego w języku C++ znaczenie szablonów nie polega (głównie) na używaniu klas kontenerowych, takich jak wektory, lecz na tym, że możliwe stało się stworzenie algorytmów STL, czyli zestawu wspólnych wzorców algorytmicznych, takich jak sortowanie i zliczanie. Większość z tych algorytmów pozwala na przekazywanie im niestandardowych funkcji w celu dostosowywania działania bez wykorzystywania wskaźników do funkcji i konstrukcji `void*`. W ten sposób możesz na przykład zmienić kolejność sortowania czy też określić, które elementy powinny być uwzględniane przy liczeniu.

Możliwość przekazywania funkcji jako argumentów do innej funkcji oraz posiadania funkcji, które zwracają nowe funkcje (lub dokładniej mówiąc, konstrukcje, które *wyglądają* jak funkcje, co omówimy w rozdziale 3.), spowodowała powstanie znormalizowanej wersji C++ jako języka funkcyjnego. Wersje C++ 11, C++ 14 i C++ 17 wprowadziły sporo możliwości, które znacznie ułatwiają pisanie programów w stylu funkcyjnym. Dodatkowe opcje to głównie lukier składniowy — jest on jednak ważny i wyraża się w postaci słowa kluczowego `auto` oraz wyrażenia lambda (te konstrukcje omówimy w rozdziale 3.). Opcje te przyniosły również znaczne ulepszenia zestawu algorytmów standardowych. Następna wersja standardu jest planowana na 2020 rok i oczekuje się, że wprowadzi ona jeszcze więcej możliwości inspirowanych programowaniem funkcyjnym, takich jak zakresy, koncepcje i współprogramy, które są obecnie zawarte w specyfikacji technicznej.

Ewolucja standardu ISO C++

Język programowania C++ jest standardem ISO. Każda nowa wersja przed wydaniem jest poddawana rygorystycznemu procesowi. Język podstawowy i biblioteka standardowa są opracowywane przez komisję, więc każda nowa funkcja jest szczegółowo omawiana i zatwierdzana, zanim stanie się częścią ostatecznej propozycji dla nowej wersji standardu. Gdy wreszcie wszystkie zmiany zostaną dołączone do definicji standardu, musi on zostać poddany kolejnemu, ostatecznemu głosowaniu, które ma miejsce w przypadku każdego nowego standardu ISO.

Od 2012 roku komitet dzieli swoje prace na podgrupy. Każda grupa zajmuje się określoną funkcją językową, która po uznaniu jej za gotową jest dostarczana jako specyfikacja techniczna. Specyfikacje techniczne nie są związane z głównym standardem i mogą później zostać do niego dołączone.

Celem specyfikacji technicznej jest przetestowanie przez programistów nowych funkcji i wykrycie błędów, zanim te funkcje znajdą się w głównym standardzie. Dostawcy kompilatorów nie są zobowiązani do implementacji specyfikacji technicznej, ale zazwyczaj to robią. Więcej informacji na ten temat można znaleźć na stronie *https://isocpp.org/std/status*.

Chociaż większość zagadnień, które omówimy w tej książce, może być wykorzystywana w starszych wersjach języka C++, skoncentrujemy się głównie na C++ 14 i C++ 17.

Kup ksi k Pole ksi k

1.6. Czego nauczysz się w trakcie czytania tej książki?

Ta książka jest skierowana przede wszystkim do doświadczonych programistów, którzy codziennie używają języka C++ i chcą wykorzystywać w swojej pracy bardziej zaawansowane narzędzia. Aby uzyskać jak najwięcej korzyści z czytania tej książki, powinieneś się zapoznać z podstawowymi opcjami języka C++, takimi jak system typów C++, referencje, konstrukcje `const`, szablony, przeciążanie operatorów itd. Nie musisz być zaznajomiony z funkcjami wprowadzonymi w wersji C++ 14 oraz 17, które bardziej szczegółowo zostały omówione w tej książce. Te funkcje nie są jeszcze szeroko stosowane i prawdopodobnie wielu czytelników nie będzie z nimi zaznajomionych.

Rozpoczniemy od podstawowych pojęć, takich jak funkcje wyższego rzędu, które pozwolą Ci zwiększyć ekspresyjność języka i sprawić, że Twoje programy będą krótsze. Pokażemy także, jak można zaprojektować oprogramowanie bez stanów mutowalnych, aby uniknąć problemów z jawną synchronizacją we współbieżnych systemach informatycznych. Następnie zmienimy bieg na wyższy i zajmiemy się bardziej zaawansowanymi tematami, takimi jak zakresy (prawdziwie modularna alternatywa dla algorytmów biblioteki standardowej) i algebraiczne typy danych (które można wykorzystać do zmniejszenia liczby stanów, w których program może się znajdować). Na koniec omówimy jeden z najczęściej wymienianych idiomów programowania funkcyjnego — mające złą sławę **monady**. Dowiemy się, w jakiś sposób można wykorzystać różne monady do wdrażania złożonych systemów, które można składać ze sobą.

Po ukończeniu czytania tej książki będziesz w stanie zaprojektować i wdrożyć bezpieczniejsze, współbieżne systemy, które można bez większego wysiłku skalować poziomo. Będziesz mógł także między innymi implementować program w sposób minimalizujący lub nawet uniemożliwiający użycie go w niepoprawnym stanie z powodu pojawienia się błędu, a także traktować oprogramowanie jako przepływ danych i używać najnowszego wynalazku C++, czyli zakresów, aby zdefiniować ten przepływ. Dzięki tym umiejętnościom będziesz w stanie stworzyć terser czy też kod mniej podatny na błędy, nawet jeśli zajmujesz się systemami oprogramowania zorientowanymi obiektowo. A gdybyś chciał w pełni wykorzystać styl funkcyjny, pomoże Ci on w bardziej przejrzysty i strukturalny sposób projektować systemy oprogramowania, co zobaczysz w rozdziale 13. podczas implementacji prostej usługi internetowej.

Podsumowanie

- Główną zasadą filozoficzną programowania funkcyjnego jest to, że nie powinieneś zajmować się sposobem, w jaki coś powinno działać, ale raczej tym, co powinno *robić*.
- Oba style — programowanie funkcyjne i programowanie obiektowe — mają wiele do zaoferowania. Powinieneś wiedzieć, kiedy można używać tylko określonego z nich, a kiedy można je łączyć ze sobą.

Kup ksi k Pole ksi k

- C++ jest multiparadygmatycznym językiem programowania, którego można używać do tworzenia programów w różnych stylach — proceduralnym, obiektowym i funkcyjnym — a także do łączenia tych stylów z programowaniem generycznym.
- Programowanie funkcyjne współgra z programowaniem generycznym, szczególnie w przypadku języka C++. Oba style zachęcają programistów, by nie myśleli na poziomie sprzętu, lecz na wyższych poziomach abstrakcji.
- Podnoszenie funkcji pozwala przekształcać funkcje działające na pojedynczych wartościach na takie, które operują na kolekcjach wartości. Dzięki złożeniu funkcji daną wartość można przetworzyć w łańcuchu transformacji, w którym każde przekształcenie przekazuje wynik do następnego.
- Unikanie stanu mutowalnego ulepsza jakość kodu i eliminuje potrzebę stosowania muteksów w kodzie wielowątkowym.
- Podejście funkcyjne oznacza posługiwanie się danymi wejściowymi i transformacjami, które należy wykonać, aby uzyskać pożądany wynik.

Kup ksi k Pole ksi k

Skorowidz

A

akcje, 170, 171
aktor, 274, 276
  transformacji, 290
  ujścia, 284
  ze stanem mutowalnym, 297
algebraiczne typy danych, 196, 212
algorytm, 42
  filter, 167
  quicksort, 147
  std::accumulate, 43, 62, 250
  std::copy_if, 54
  std::count, 21
  std::count_if, 70, 90
  std::for_each, 77, 100
  std::transform, 23, 54
algorytmy STL, 53
anonimowy obiekt funkcyjny, 74
argumenty powiązane, 101
automatyczna dedukcja typu, 64
automatyczne generowanie testów, 305

B

biblioteka
  Boost, 84, 96
  JSON, 292
  Mach7, 218
  range-v3, 171
  SObjectizer, 276
  STL, 22, 42, 49
błędy, 207, 234
  parsowania, 291
buforowanie wyników funkcji, 148

C

ciąg Fibonacciego, 148
ciągła optymalizacja, 33
częściowe stosowanie funkcji, 90, 92, 109
czytelność kodu, 32

D

dane
  mutowalne, 131
  niemutowalne, 131
  tylko do odczytu, 166
debugowanie, 301
  typów dedukowanych, 252
dedukcja
  argumentów, 285
  typu zwracanego, 64
deklaracja const, 134
domknięcia, 73, 75
dopasowywanie
  do wzorców, 215, 218, 254
  typów zdekonstruowanych, 219
dostęp do bazy danych, 106
drzewo
  rekurencji, 149
  składni, 267, 271
  trie, 186, 310
    aktualizacja elementów, 191
    całkowita wydajność, 192
    dołączanie elementów, 189
    usuwanie elementów, 192
    wyszukiwanie elementu, 187
    złożoność obliczeniowa, 192
DSL, domain-specific language, 266
dziedziczenie, 197

F

filtrowanie, 52, 53
kolekcji, 40
strumieni reaktywnych, 290
wiadomości w strumieniu, 290
funkcja, 64
.then, 245
.with_name, 137
.with_surname, 137
append_name_if, 62
bind2nd, 94
construct, 229
count_lines, 22, 23
count_lines_in_files, 28
employees_names, 139
employment_history, 135
filter, 166, 169
greater_than, 94
join, 228, 231, 295
lev, 152
make_memoized, 153
max, 124
mbind, 230–233, 236, 239, 245
mcompose, 246
names_for, 56
next_position, 127
print_person, 99
query, 107
some_function, 138
std::bind, 92, 95, 98, 109
std::greater, 97
std::invoke, 263
std::less, 97
std::visit, 216, 220
string_to_lower, 176
swap, 210
take, 169
to_html, 239
transform, 167, 225, 226, 231, 288
user_full_name, 226, 239
funkcje
buforowanie wyników, 148
częściowe stosowanie, 90, 92, 109
czyste, 24, 122, 304
dwuargumentowe, 92
jednoargumentowe, 92
kontynuacyjne, 243
monadyczne, 246
opakowujące, 93
opakowujące operatory, 83
podniesione, 114
pomocnicze, 60
przeciążanie, 137
rekurencyjne, 154
rozwijanie, 103, 106, 108, 261
składowe, 133, 136, 137, 144
tworzenie, 89
w postaci argumentów, 55
wiązanie argumentów, 95, 96, 99, 100
wytwórcze, 143
wyższego rzędu, 40, 55
z zakresami, 167
zapamiętujące, 153, 154
złożenie, 110
funkcyjne struktury danych, 179
funktor, 70, 222, 226
fuzzing, 314

G

generowanie
przypadków testowych, 306–310
trójek pitagorejskich, 233
generyczne obiekty funkcyjne, 70, 72
getter, 213, 274
gniazdo, 294
grupowanie, 164

I

idiom
copy-and-swap, 211, 269
erase-remove, 52
implementacja
funkcji, 56
metafunkcji, 256
określonych stanów, 204
przy użyciu zwijania, 61
rekurencyjna, 58
wartości przyszłych, 244
inkrementacja, 167
interfejs
aktora, 277
użytkownika
wyświetlanie elementów, 147
iteratory, 49, 165

J

język
  C++, 34
  C++ 17, 44, 49
języki dziedzinowe, 265
JSON, 292

K

kacze typowanie, 63
klasa
  boost::future, 244
  curried, 264
  expected, 220, 293
  expected<T, E>, 291
  lazy_val, 142
  optional, 293
  std::function, 86
klienty, 294
kolekcja łańcuchów, 114
kolekcje
  odwracanie elementów, 116
kompilacja, 250, 254, 258
konstruktor, 143
  kopiujący, 210
  przenoszący, 210
kopiowanie przy zapisie, 186
krotki, 197
kwalifikator const, 132, 133, 139

L

labirynt, 126
leniwe
  łączenie łańcuchów, 155
  wartościowanie, 141, 145, 168
listy
  dodawanie elementów, 180–182
  niemutowalne, 180
  usuwanie elementów, 180–182

Ł

łańcuch
  przycinanie, 48
łączenie łańcuchów, 155, 159
  leniwe, 155
  strumieni, 289
  wydajne, 159
łączność, 49

M

metafunkcja, 250, 253
  is_same, 256
  remove_reference_t, 256
  std::remove_cv_t, 253
  void_t, 258
metainformacje o typach, 257
metaprogramowanie szablonów, 249, 271
model aktora, 274
modelowanie
  dziedziny, 212
  strumieni reaktywnych, 281
modyfikator transform, 286
monada, 221, 226, 234, 236, 281, 293
  expected, 236, 237, 293
  Try, 237
monady
  kontynuacyjne, 240, 243
  obsługa stanu, 238
  składanie, 236, 245
  typ future, 242
  typy opcjonalne, 234
  wiązanie, 243
  współbieżnościowe, 240
muteksy, 130, 144
mutowalność zmiennych, 170
myślenie komponentowe, 274

N

nagłówek type_traits, 254, 271
nawias trójkątny, 84
niemutowalne
  listy łączone, 180
  struktury danych, 185
niemutowalność, 26
notacja, 41
  infiksowa, 47

O

obiekt
  expected, 293, 295
  ujścia, 283
obiekty
  funkcyjne, 64
    anonimowe, 74
    generyczne, 70, 72

obiekty
funkcyjne
opakowywanie, 86
operatorów, 83, 84
zwięzłe, 80
JSON, 291
wywoływalne, 87
obliczanie
ciągu Fibonacciego, 150
częstości, 175
iloczynu, 44
sumy kolekcji, 45
średnich, 42, 120
obsługa
algebraicznych typów danych, 215
błędów, 207, 234, 235
parsowania, 291
w strumieniach, 291
stanu, 238
wartości opcjonalnych, 223
oczyszczanie pamięci, 184
odbieranie wiadomości, 283
odległość Levenshteina, 151, 160
odpowiadanie klientowi, 293
odwracanie elementów, 116
określanie typów funkcji, 24, 41
opakowanie propagate_const, 140
opakowywanie obiektów funkcyjnych, 86
operator, 83
|, 23
>, 102
inkrementacji, 167
new, 209
rzutowania, 144
wyłuskania, 168
wywołania, 68, 92
wywołania dla transakcji, 269
operatory
arytmetyczne, 83
bitowe, 83
logiczne, 83
porównania, 83
opóźniona deklaracja typu, 64
optymalizacja, 57
funkcji składowych, 136
małych obiektów, 88
obsługi zakresów, 172

P

pamięć, 183
oczyszczanie, 184
podręczna, 150
partycjonowanie kolekcji, 50
pary, 197
pętla, 56
for oparta na zakresach, 173, 178
rekurencyjna, 128
podnoszenie
funkcji, 30, 113, 115
wyrażenia lambda, 116
polimorfizm, 35, 67
prawo Amdahla, 131
predykat, 50, 71
filtrowania, 42
programowanie
deklaratywne, 19
dynamiczne, 150
funkcyjne, 18
generyczne, 34
imperatywne, 19
obiektowe, 19
przechowywanie łańcuchów, 157
przechwytywanie
typu, 78
wyrażenia lambda, 78
przeciążanie
funkcji, 137
operatora wywołania, 68, 70
przejrzystość referencyjna, 122
przekazywanie
perfekcyjne argumentów, 66
przez referencję, 66
przez wartość, 50
przenoszenie obiektu, 138
przycinanie łańcucha, 48

R

redukcja, 45
referencje
do funkcji, 67
do r-wartości, 67
podwójne, 67
przekazujące, 67
stałe, 133
uniwersalne, 67

rekurencja, 57, 148
ogonowa, 57, 59
usuwanie struktury, 185
wyznaczanie odległości Levenshteina, 152
rekurencyjna pętla, 128
rozwijanie funkcji, 103, 104, 109
rzutowanie, 144

S

serwer, 296
setter, 269, 274
składanie
algorytmów STL, 53
monad, 236, 245
składnia
placement new, 209
wyrażenia lambda, 74
słowo kluczowe
auto, 65
const, 132, 138
constexpr, 132
mutable, 136
this, 134
sortowanie, 98, 145, 146
lista par, 177
specyfikator
constexpr, 263, 265
mutable, 77
sprawdzanie właściwości
sortowania, 309
typu, 258
stan, 24
mutowalny, 27, 120, 129, 140, 297
niemutowalny, 129
STL, 22
struktura
klasy szablonowej, 92
przechowująca gniazdo, 294
struktury danych
funkcyjne, 179
niemutowalne, 185
strumienie
asynchroniczne, 282
reaktywne, 282, 286, 290
strumień std::cerr, 124
symbole
wieloznaczne, 75
zastępcze, 95, 96
synchronizacja, 33
systemy
rozproszone, 298
współbieżne, 273
szablon, 113, 249, 271
wyrażeń, 155, 159, 161
szkielety programowe, 276

Ś

środowisko współbieżne, 129

T

testowanie, 301
funkcji, 307
oparte na właściwościach, 307
systemów współbieżnych, 311
testy
generowane automatycznie, 305
jednostkowe, 304
porównawcze, 309
transakcje, 269
transformacja, 23, 24, 52
join, 289
strumieni reaktywnych, 286
tworzenie
aktorów, 297
funkcji rozwiniętych, 261
języka dziedzinowego, 265
obiektu funkcyjnego, 72, 80, 94
predykatu, 71
strumienia, 288
systemów rozproszonych, 298
ujścia, 283
widoków danych, 166
źródła wiadomości, 278
typ
expected, 237
expected<T, E>, 208, 210, 236, 252
future, 242
std::variant, 200–202
std::optional, 206, 223
std::vector, 229
std::optional<T>, 234
with_client, 295
typowanie
kacze, 70
silne, 70

typy
  danych algebraiczne, 196, 212
  dedukowane, 252
  funkcji, 41
  sumy, 197, 200
  wyliczeniowe, 197

U

unie, 200
unikanie stanu mutowalnego, 27
uogólnione zapamiętywanie, 152
uruchamianie usługi, 285
usługa odbierająca połączenia, 278
usuwanie
  elementów
    z końca drzewa trie, 192
    z końca listy, 181
    z początku listy, 180
    z wnętrza listy, 182
  struktury, 185
  typu, 87

W

wartości
  opcjonalne, 205, 223
  przyszłe, 244
  zakresów, 168
wartościowanie leniwe, 141, 145, 150, 168
wartownik, 173
wektor, 185, 229, 310
wiązanie, 243
  argumentów, 95–100
widok, 171
  danych, 166
  zakresów, 178
właściwości typu, 258
wskaźnik
  std::shared_ptr, 183
  this, 210
  współdzielony, 280, 281
wskaźniki do funkcji, 67
współbieżne procesy, 130
współbieżność, 33
wydajność drzewa trie, 192
wyjątek, 237
  std::exception_ptr, 291
wyjście std::cerr, 295
wyłuskanie, 168
wyrażenia lambda, 73–76, 101, 279
  podnoszenie, 116
  składnia, 74
  typu mutable, 77
  uogólnione, 79
  zmienne składowe, 77
wysyłanie wiadomości, 280
wyszukiwanie
  elementu, 187
  wartości maksymalnej, 123
wywoływanie obiektów wywoływalnych, 263
wyznaczanie
  odległości Levenshteina, 152
  współrzędnych, 127
wzajemne wykluczenie, 130

Z

zagnieżdżone funktory, 227
zakres posortowanych słów, 177
zakresy, 23, 57, 163, 168, 226
  do obliczania częstości, 175
  mutowalność zmiennych, 170
  nieskończone, 172, 173
  ograniczone, 172
  składane, 231, 233
zapamiętywanie uogólnione, 152
zarządzanie
  pamięcią, 183
  typami w czasie kompilacji, 250
zgodność
  logiczna, 134
  wewnętrzna, 134
zliczanie znaków, 47
złożenie funkcji, 110
zmienne mutowalne, 27
zwięzłość kodu, 32
zwijanie, 45, 61
  od lewej, 48
  od prawej, 48

Ź

źródło wiadomości, 278

Kup ksi k Pole ksi k

PROGRAM PARTNERSKI
GRUPY HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl

GRUPA Helion

Programowanie funkcyjne w C++: twórz najlepsze rozwiązania!

Programowanie jest sztuką, dzięki której możesz stworzyć coś z niczego, przy czym tylko od Ciebie zależy, jak doskonałe będzie to dzieło. Dobrze napisany kod jest wydajny, łatwy w testowaniu, można go używać ponownie i wykazuje mniejszą podatność na błędy. Jednym słowem, taki kod powinien możliwie prosto wyrażać złożoną logikę programu, bezproblemowo obsługiwać błędy i przejrzyście implementować współbieżność. Te wymagania pozwoli Ci spełnić funkcyjny styl programowania. Język C++ umożliwia programowanie funkcyjne dzięki szablonom, wyrażeniom lambda i innym ważnym opcjom. Pomocne też będzie korzystanie z biblioteki STL.

Ta książka jest przeznaczona dla profesjonalnych programistów C++, którzy chcą opanować funkcyjny styl programowania i dzięki temu wykorzystać w nowy sposób potężne zalety tego języka. Po interesującym wprowadzeniu do tej metodologii zamieszczono tu dziesiątki przykładów, schematów i ilustracji wyjaśniających koncepcje programowania funkcyjnego w C++. Pokazano, jak tworzyć bezpieczniejszy kod bez obniżania wydajności pracy programu, jak stosować obiekty funkcyjne, funkcje stosowane, algebraiczne typy danych oraz wiele innych. Nie zabrakło praktycznych przykładów kodu, który stanowi znakomite uzupełnienie prezentowanych treści.

W tej książce między innymi:

- wprowadzenie do programowania funkcyjnego
- funkcje w C++ i funkcje wyższego rzędu oraz ich rozwijanie
- wartościowanie leniwe i wykorzystanie go do optymalizacji
- korzystanie z funktorów i monad
- funkcyjny sposób testowania i debugowania kodu

Ivan Čukić jest wykładowcą na Wydziale Matematyki Uniwersytetu Belgradzkiego, uczy nowoczesnych technik programowania i programowania funkcyjnego. Od ponad dwudziestu lat używa C++. Stosuje techniki programowania funkcyjnego do tworzenia oprogramowania, z którego korzystają setki milionów osób na całym świecie. Jest jednym z głównych programistów w KDE — największym darmowym projekcie C++ opartym na otwartych źródłach.