

Anthony Williams

Odkryj wszystkie tajemnice wielowątkowych aplikacji!

# Język C++

i przetwarzanie  
współbieżne w akcji



Tytuł oryginału: C++ Concurrency in Action: Practical Multithreading

Tłumaczenie: Mikołaj Szczepaniak

Projekt okładki: Anna Mitka

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

ISBN: 978-83-246-5086-6

Original edition copyright © 2012 by Manning Publications Co.

All rights reserved.

Polish edition copyright © 2013 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 jakiegokolwiek 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 Wydawnictwo HELION 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 Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION

ul. Kościuszki 1c, 44-100 GLIWICE

tel. 32 231 22 19, 32 230 98 63

e-mail: [helion@helion.pl](mailto:helion@helion.pl)

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie?gimpbi>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

# Spis treści

---

<i>Słowo wstępne</i> .....	11
<i>Podziękowania</i> .....	13
<i>O tej książce</i> .....	15
<b>Rozdział 1. Witaj, świecie współbieżności w C++!</b> .....	<b>19</b>
1.1. Czym jest współbieżność? .....	20
1.1.1. Współbieżność w systemach komputerowych .....	20
1.1.2. Modele współbieżności .....	22
1.2. Dlaczego warto stosować współbieżność? .....	25
1.2.1. Stosowanie współbieżności do podziału zagadnień .....	25
1.2.2. Stosowanie współbieżności do podniesienia wydajności .....	26
1.2.3. Kiedy nie należy stosować współbieżności .....	27
1.3. Współbieżność i wielowątkowość w języku C++ .....	28
1.3.1. Historia przetwarzania wielowątkowego w języku C++ .....	29
1.3.2. Obsługa współbieżności w nowym standardzie .....	30
1.3.3. Efektywność biblioteki wątków języka C++ .....	30
1.3.4. Mechanizmy związane z poszczególnymi platformami .....	32
1.4. Do dzieła! .....	32
1.4.1. „Witaj świecie współbieżności” .....	32
1.5. Podsumowanie .....	34
<b>Rozdział 2. Zarządzanie wątkami</b> .....	<b>35</b>
2.1. Podstawowe zarządzanie wątkami .....	36
2.1.1. Uruchamianie wątku .....	36
2.1.2. Oczekiwanie na zakończenie wątku .....	39
2.1.3. Oczekiwanie w razie wystąpienia wyjątku .....	39
2.1.4. Uruchamianie wątków w tle .....	42
2.2. Przekazywanie argumentów do funkcji wątku .....	43
2.3. Przenoszenie własności wątku .....	46
2.4. Wybór liczby wątków w czasie wykonywania .....	49
2.5. Identyfikowanie wątków .....	52
2.6. Podsumowanie .....	54
<b>Rozdział 3. Współdzielenie danych przez wątki</b> .....	<b>55</b>
3.1. Problemy związane ze współdzieleniem danych przez wątki .....	56
3.1.1. Sytuacja wyścigu .....	58
3.1.2. Unikanie problematycznych sytuacji wyścigu .....	59
3.2. Ochrona współdzielonych danych za pomocą muteksów .....	60
3.2.1. Stosowanie muteksów w języku C++ .....	60
3.2.2. Projektowanie struktury kodu z myślą o ochronie współdzielonych danych .....	62

3.2.3. Wykrywanie sytuacji wyścigu związanych z interfejsami .....	63
3.2.4. Zakleszczenie: problem i rozwiązanie .....	71
3.2.5. Dodatkowe wskazówki dotyczące unikania zakleszczeń .....	73
3.2.6. Elastyczne blokowanie muteksów za pomocą szablonu <code>std::unique_lock</code> .....	79
3.2.7. Przenoszenie własności muteksu pomiędzy zasięgami .....	80
3.2.8. Dobór właściwej szczególności blokad .....	82
<b>3.3. Alternatywne mechanizmy ochrony współdzielonych danych .....</b>	<b>84</b>
3.3.1. Ochrona współdzielonych danych podczas inicjalizacji .....	84
3.3.2. Ochrona rzadko aktualizowanych struktur danych .....	88
3.3.3. Blokowanie rekurencyjne .....	90
<b>3.4. Podsumowanie .....</b>	<b>91</b>
<b>Rozdział 4. Synchronizacja współbieżnych operacji .....</b>	<b>93</b>
<b>4.1. Oczekiwanie na zdarzenie lub inny warunek .....</b>	<b>94</b>
4.1.1. Oczekiwanie na spełnienie warunku za pomocą zmiennych warunkowych .....	95
4.1.2. Budowa kolejki gwarantującej bezpieczne przetwarzanie wielowątkowe przy użyciu zmiennych warunkowych .....	97
<b>4.2. Oczekiwanie na jednorazowe zdarzenia za pomocą przyszłości .....</b>	<b>102</b>
4.2.1. Zwracanie wartości przez zadania wykonywane w tle .....	103
4.2.2. Wiązanie zadania z przeszłością .....	106
4.2.3. Obietnice (szablon <code>std::promise</code> ) .....	109
4.2.4. Zapisywanie wyjątku na potrzeby przyszłości .....	111
4.2.5. Oczekiwanie na wiele wątków .....	112
<b>4.3. Oczekiwanie z limitem czasowym .....</b>	<b>115</b>
4.3.1. Zegary .....	115
4.3.2. Okresy .....	117
4.3.3. Punkty w czasie .....	118
4.3.4. Funkcje otrzymujące limity czasowe .....	120
<b>4.4. Upraszczenie kodu za pomocą technik synchronizowania operacji .....</b>	<b>121</b>
4.4.1. Programowanie funkcyjne przy użyciu przyszłości .....	122
4.4.2. Synchronizacja operacji za pomocą przesyłania komunikatów .....	127
<b>4.5. Podsumowanie .....</b>	<b>131</b>
<b>Rozdział 5. Model pamięci języka C++ i operacje na typach atomowych .....</b>	<b>133</b>
<b>5.1. Podstawowe elementy modelu pamięci .....</b>	<b>134</b>
5.1.1. Obiekty i miejsca w pamięci .....	134
5.1.2. Obiekty, miejsca w pamięci i przetwarzanie współbieżne .....	135
5.1.3. Kolejność modyfikacji .....	136
<b>5.2. Operacje i typy atomowe języka C++ .....</b>	<b>137</b>
5.2.1. Standardowe typy atomowe .....	138
5.2.2. Operacje na typie <code>std::atomic_flag</code> .....	141
5.2.3. Operacje na typie <code>std::atomic&lt;bool&gt;</code> .....	143
5.2.4. Operacje na typie <code>std::atomic&lt;T*&gt;</code> — arytmetyka wskaźników .....	146
5.2.5. Operacje na standardowych atomowych typach całkowitoliczbowych .....	147
5.2.6. Główny szablon klasy <code>std::atomic&lt;&gt;</code> .....	147
5.2.7. Wolne funkcje dla operacji atomowych .....	150
<b>5.3. Synchronizacja operacji i wymuszanie ich porządku .....</b>	<b>151</b>
5.3.1. Relacja synchronizacji .....	152
5.3.2. Relacja poprzedzania .....	154
5.3.3. Porządkowanie pamięci na potrzeby operacji atomowych .....	155
5.3.4. Sekwencje zwalniania i relacja synchronizacji .....	175

5.3.5. Ogrodzenia .....	178
5.3.6. Porządkowanie operacji nieatomowych za pomocą operacji atomowych .....	180
<b>5.4. Podsumowanie .....</b>	<b>182</b>
<b>Rozdział 6. Projektowanie współbieżnych struktur danych przy użyciu blokad .....</b>	<b>183</b>
<b>6.1. Co oznacza projektowanie struktur danych pod kątem współbieżności? .....</b>	<b>184</b>
6.1.1. Wskazówki dotyczące projektowania współbieżnych struktur danych .....	185
<b>6.2. Projektowanie współbieżnych struktur danych przy użyciu blokad .....</b>	<b>186</b>
6.2.1. Stos gwarantujący bezpieczeństwo przetwarzania wielowątkowego przy użyciu blokad .....	187
6.2.2. Kolejka gwarantująca bezpieczeństwo przetwarzania wielowątkowego przy użyciu blokad i zmiennych warunkowych .....	190
6.2.3. Kolejka gwarantująca bezpieczeństwo przetwarzania wielowątkowego przy użyciu szczegółowych blokad i zmiennych warunkowych .....	194
<b>6.3. Projektowanie złożonych struktur danych przy użyciu blokad .....</b>	<b>207</b>
6.3.1. Implementacja tablicy wyszukiwania gwarantującej bezpieczeństwo przetwarzania wielowątkowego przy użyciu blokad .....	207
6.3.2. Implementacja listy gwarantującej bezpieczeństwo przetwarzania wielowątkowego przy użyciu blokad .....	213
<b>6.4. Podsumowanie .....</b>	<b>218</b>
<b>Rozdział 7. Projektowanie współbieżnych struktur danych bez blokad .....</b>	<b>219</b>
<b>7.1. Definicje i ich praktyczne znaczenie .....</b>	<b>220</b>
7.1.1. Rodzaje nieblokujących struktur danych .....	220
7.1.2. Struktury danych bez blokad .....	221
7.1.3. Struktury danych bez oczekiwania .....	222
7.1.4. Zalety i wady struktur danych bez blokad .....	222
<b>7.2. Przykłady struktur danych bez blokad .....</b>	<b>223</b>
7.2.1. Implementacja stosu gwarantującego bezpieczeństwo przetwarzania wielowątkowego bez blokad .....	224
7.2.2. Eliminowanie niebezpiecznych wycieków — zarządzanie pamięcią w strukturach danych bez blokad .....	228
7.2.3. Wykrywanie węzłów, których nie można odzyskać, za pomocą wskaźników ryzyka .....	233
7.2.4. Wykrywanie używanych węzłów metodą zliczania referencji .....	242
7.2.5. Zmiana modelu pamięci używanego przez operacje na stosie bez blokad .....	247
7.2.6. Implementacja kolejki gwarantującej bezpieczeństwo przetwarzania wielowątkowego bez blokad .....	252
<b>7.3. Wskazówki dotyczące pisania struktur danych bez blokad .....</b>	<b>264</b>
7.3.1. Wskazówka: na etapie tworzenia prototypu należy stosować tryb <code>std::memory_order_seq_cst</code> .....	265
7.3.2. Wskazówka: należy używać schematu odzyskiwania pamięci bez blokad .....	265
7.3.3. Wskazówka: należy unikać problemu ABA .....	266
7.3.4. Wskazówka: należy identyfikować pętle aktywnego oczekiwania i wykorzystywać czas bezczynności na wspieranie innego wątku .....	267
<b>7.4. Podsumowanie .....</b>	<b>267</b>
<b>Rozdział 8. Projektowanie współbieżnego kodu .....</b>	<b>269</b>
<b>8.1. Techniki dzielenia pracy pomiędzy wątki .....</b>	<b>270</b>
8.1.1. Dzielenie danych pomiędzy wątki przed rozpoczęciem przetwarzania .....	271
8.1.2. Rekurencyjne dzielenie danych .....	272
8.1.3. Dzielenie pracy według typu zadania .....	276

8.2.	<b>Czynniki wpływające na wydajność współbieżnego kodu</b>	<b>279</b>
8.2.1.	Liczba procesorów	280
8.2.2.	Współzawodnictwo o dane i ping-pong bufora	281
8.2.3.	Falszywe współdzielenie	284
8.2.4.	Jak blisko należy rozmieścić dane?	285
8.2.5.	Nadsubskrypcja i zbyt intensywne przełączanie zadań	285
8.3.	<b>Projektowanie struktur danych pod kątem wydajności przetwarzania wielowątkowego</b>	<b>286</b>
8.3.1.	Podział elementów tablicy na potrzeby złożonych operacji	287
8.3.2.	Wzorce dostępu do danych w pozostałych strukturach	289
8.4.	<b>Dodatkowe aspekty projektowania współbieżnych struktur danych</b>	<b>291</b>
8.4.1.	Bezpieczeństwo wyjątków w algorytmach równoległych	291
8.4.2.	Skalowalność i prawo Amdahla	298
8.4.3.	Ukrywanie opóźnień za pomocą wielu wątków	300
8.4.4.	Skracanie czasu reakcji za pomocą technik przetwarzania równoległego	301
8.5.	<b>Projektowanie współbieżnego kodu w praktyce</b>	<b>303</b>
8.5.1.	Równoległa implementacja funkcji <code>std::for_each</code>	304
8.5.2.	Równoległa implementacja funkcji <code>std::find</code>	306
8.5.3.	Równoległa implementacja funkcji <code>std::partial_sum</code>	312
8.6.	<b>Podsumowanie</b>	<b>322</b>
<b>Rozdział 9. Zaawansowane zarządzanie wątkami</b>		<b>323</b>
9.1.	<b>Pule wątków</b>	<b>324</b>
9.1.1.	Najprostsza możliwa pula wątków	324
9.1.2.	Oczekiwanie na zadania wysyłane do puli wątków	327
9.1.3.	Zadania oczekujące na inne zadania	330
9.1.4.	Unikanie współzawodnictwa w dostępie do kolejki zadań	333
9.1.5.	Wykradanie zadań	335
9.2.	<b>Przerywanie wykonywania wątków</b>	<b>340</b>
9.2.1.	Uruchamianie i przerywanie innego wątku	340
9.2.2.	Wykrywanie przerywania wątku	342
9.2.3.	Przerywanie oczekiwania na zmienną warunkową	343
9.2.4.	Przerywanie oczekiwania na zmienną typu <code>std::condition_variable_any</code>	346
9.2.5.	Przerywanie pozostałych wywołań blokujących	348
9.2.6.	Obsługa przerwania	349
9.2.7.	Przerywanie zadań wykonywanych w tle podczas zamykania aplikacji	350
9.3.	<b>Podsumowanie</b>	<b>352</b>
<b>Rozdział 10. Testowanie i debugowanie aplikacji wielowątkowych</b>		<b>353</b>
10.1.	<b>Rodzaje błędów związanych z przetwarzaniem współbieżnym</b>	<b>354</b>
10.1.1.	Niechciane blokowanie	354
10.1.2.	Sytuacje wyścigu	355
10.2.	<b>Techniki lokalizacji błędów związanych z przetwarzaniem współbieżnym</b>	<b>357</b>
10.2.1.	Przeglądanie kodu w celu znalezienia ewentualnych błędów	357
10.2.2.	Znajdowanie błędów związanych z przetwarzaniem współbieżnym poprzez testowanie kodu	359
10.2.3.	Projektowanie kodu pod kątem łatwości testowania	361
10.2.4.	Techniki testowania wielowątkowego kodu	363
10.2.5.	Projektowanie struktury wielowątkowego kodu testowego	366
10.2.6.	Testowanie wydajności wielowątkowego kodu	369
10.3.	<b>Podsumowanie</b>	<b>370</b>

<b>Dodatek A Krótki przegląd wybranych elementów języka C++11</b>	<b>371</b>
A.1. Referencje do r-wartości .....	371
A.1.1. Semantyka przenoszenia danych .....	372
A.1.2. Referencje do r-wartości i szablony funkcji .....	375
A.2. Usunięte funkcje .....	376
A.3. Funkcje domyślne .....	377
A.4. Funkcje constexpr .....	381
A.4.1. Wyrażenia constexpr i typy definiowane przez użytkownika .....	382
A.4.2. Obiekty constexpr .....	385
A.4.3. Wymagania dotyczące funkcji constexpr .....	385
A.4.4. Słowo constexpr i szablony .....	386
A.5. Funkcje lambda .....	386
A.5.1. Funkcje lambda odwołujące się do zmiennych lokalnych .....	388
A.6. Szablony zmiennoargumentowe .....	391
A.6.1. Rozwijanie paczki parametrów .....	392
A.7. Automatyczne określanie typu zmiennej .....	395
A.8. Zmienne lokalne wątków .....	396
A.9. Podsumowanie .....	397
<b>Dodatek B Krótkie zestawienie bibliotek przetwarzania współbieżnego</b>	<b>399</b>
<b>Dodatek C Framework przekazywania komunikatów i kompletny przykład implementacji systemu bankomatu</b>	<b>401</b>
<b>Dodatek D Biblioteka wątków języka C++</b>	<b>419</b>
D.1. Nagłówek <code>&lt;chrono&gt;</code> .....	419
D.1.1. Szablon klasy <code>std::chrono::duration</code> .....	420
D.1.2. Szablon klasy <code>std::chrono::time_point</code> .....	429
D.1.3. Klasa <code>std::chrono::system_clock</code> .....	431
D.1.4. Klasa <code>std::chrono::steady_clock</code> .....	433
D.1.5. Definicja typu <code>std::chrono::high_resolution_clock</code> .....	435
D.2. Nagłówek <code>&lt;condition_variable&gt;</code> .....	435
D.2.1. Klasa <code>std::condition_variable</code> .....	436
D.2.2. Klasa <code>std::condition_variable_any</code> .....	444
D.3. Nagłówek <code>&lt;atomic&gt;</code> .....	452
D.3.1. Definicje typów <code>std::atomic_XXX</code> .....	454
D.3.2. Makra <code>ATOMIC_XXX_LOCK_FREE</code> .....	454
D.3.3. Makro <code>ATOMIC_VAR_INIT</code> .....	455
D.3.4. Typ wylczeniowy <code>std::memory_order</code> .....	455
D.3.5. Funkcja <code>std::atomic_thread_fence</code> .....	456
D.3.6. Funkcja <code>std::atomic_signal_fence</code> .....	457
D.3.7. Klasa <code>std::atomic_flag</code> .....	457
D.3.8. Szablon klasy <code>std::atomic</code> .....	460
D.3.9. Specjalizacje szablonu <code>std::atomic</code> .....	471
D.3.10. Specjalizacje szablonu <code>std::atomic&lt;typ-całkowitoliczbowy&gt;</code> .....	472
D.4. Nagłówek <code>&lt;future&gt;</code> .....	489
D.4.1. Szablon klasy <code>std::future</code> .....	490
D.4.2. Szablon klasy <code>std::shared_future</code> .....	495
D.4.3. Szablon klasy <code>std::packaged_task</code> .....	501
D.4.4. Szablon klasy <code>std::promise</code> .....	507
D.4.5. Szablon funkcji <code>std::async</code> .....	513

<b>D.5. Nagłówek &lt;mutex&gt;</b> .....	<b>514</b>
D.5.1. Klasa std::mutex .....	515
D.5.2. Klasa std::recursive_mutex .....	518
D.5.3. Klasa std::timed_mutex .....	520
D.5.4. Klasa std::recursive_timed_mutex .....	524
D.5.5. Szablon klasy std::lock_guard .....	529
D.5.6. Szablon klasy std::unique_lock .....	530
D.5.7. Szablon funkcji std::lock .....	540
D.5.8. Szablon funkcji std::try_lock .....	541
D.5.9. Klasa std::once_flag .....	541
D.5.10. Szablon funkcji std::call_once .....	542
<b>D.6. Nagłówek &lt;ratio&gt;</b> .....	<b>543</b>
D.6.1. Szablon klasy std::ratio .....	544
D.6.2. Alias szablonu std::ratio_add .....	544
D.6.3. Alias szablonu std::ratio_subtract .....	545
D.6.4. Alias szablonu std::ratio_multiply .....	545
D.6.5. Alias szablonu std::ratio_divide .....	546
D.6.6. Szablon klasy std::ratio_equal .....	547
D.6.7. Szablon klasy std::ratio_not_equal .....	547
D.6.8. Szablon klasy std::ratio_less .....	547
D.6.9. Szablon klasy std::ratio_greater .....	548
D.6.10. Szablon klasy std::ratio_less_equal .....	548
D.6.11. Szablon klasy std::ratio_greater_equal .....	548
<b>D.7. Nagłówek &lt;thread&gt;</b> .....	<b>549</b>
D.7.1. Klasa std::thread .....	549
D.7.2. Przestrzeń nazw std::this_thread .....	558
<b>Materiały dodatkowe</b> .....	<b>561</b>
<b>Skorowidz</b> .....	<b>563</b>



# 4

## Synchronizacja współbieżnych operacji

---

### ***W tym rozdziale zostaną omówione następujące zagadnienia:***

- oczekiwanie na zdarzenie;
- oczekiwanie na jednorazowe zdarzenia za pomocą przyszłości;
- oczekiwanie z limitem czasowym;
- upraszczanie kodu za pomocą technik synchronizowania operacji.

W poprzednim rozdziale przeanalizowaliśmy rozmaite sposoby ochrony danych współdzielonych przez wiele wątków. Okazuje się jednak, że w pewnych przypadkach jest potrzebna nie tyle ochrona danych, co synchronizacja działań podejmowanych przez różne wątki. Wątek może na przykład czekać z realizacją własnej operacji na zakończenie pewnego zadania przez inny wątek. Ogólnie w wielu przypadkach wątek oczekujący na określone zdarzenie lub spełnienie pewnego warunku jest najwygodniejszym rozwiązaniem. Mimo że analogiczne rozwiązanie można zaimplementować w formie mechanizmu okresowego sprawdzania flagi zakończonego zadania lub innej wartości zapisanej we współdzielonych danych, taki model byłby daleki od ideału. Konieczność synchronizacji operacji wykonywanych przez różne wątki jest dość typowym scenariuszem, zatem biblioteka standardowa języka C++ oferuje mechanizmy ułatwiające obsługę tego modelu, w tym **zmiennie warunkowe** i tzw. **przyszłości**.

W tym rozdziale omówię techniki oczekiwania na zdarzenia przy użyciu zmiennych warunkowych oraz sposoby upraszczania synchronizacji operacji za pomocą przyszłości.

## 4.1. Oczekiwanie na zdarzenie lub inny warunek

Przypuśćmy, że podróżujemy nocnym pociągiem. Jednym ze sposobów zagwarantowania, że wysiadziemy na właściwej stacji, jest unikanie snu i sprawdzanie wszystkich stacji, na których zatrzymuje się nasz pociąg. W ten sposób nie przegapimy naszej stacji, jednak po dotarciu na miejsce będziemy bardzo zmęczeni. Alternatywnym rozwiązaniem jest sprawdzenie rozkładu jazdy pod kątem godziny przyjazdu, ustawienie budzika z pewnym wyprzedzeniem względem tej godziny i pójście spać. To rozwiązanie jest dość bezpieczne — nie przegapimy naszej stacji, ale jeśli pociąg się spóźni, wstaniemy zbyt wcześnie. Nie można też wykluczyć sytuacji, w której wyczerpią się baterie w budziku — w takim przypadku możemy zaspać i przegapić swoją stację. Idealnym rozwiązaniem byłaby możliwość pójścia spać i skorzystania z pomocy czegoś (lub kogoś), co obudziłoby nas bezpośrednio przed osiągnięciem stacji docelowej.

Jaki to ma związek z wątkami? Jeśli jeden wątek czeka, aż inny wątek zakończy jakieś zadanie, ma do wyboru kilka możliwych rozwiązań. Po pierwsze, może stale sprawdzać odpowiednią flagę we współdzielonych danych (chronionych przez muteks); flaga zostanie ustawiona przez drugi wątek w momencie zakończenia zadania. Takie rozwiązanie jest nieefektywne z dwóch powodów: wątek, który wielokrotnie sprawdza wspomnianą flagę, zajmuje cenny czas procesora, a muteks zablokowany przez oczekujący wątek nie jest dostępny dla żadnego innego wątku. Oba te czynniki działają na niekorzyść oczekującego wątku, ponieważ ten wątek zajmuje zasoby potrzebne także do działania wątku, na który czeka, co opóźnia wykonanie zadania i ustawienie odpowiedniej flagi. Sytuacja przypomina unikanie snu przez całą podróż pociągiem i prowadzenie rozmowy z maszynistą — maszynista zajęty rozmową musi prowadzić pociąg nieco wolniej, zatem później dotrzemy na swoją stację. Podobnie wątek oczekujący zajmuje zasoby, które mogłyby być używane przez pozostałe wątki w systemie, przez co czas oczekiwania może być dłuższy, niż to konieczne.

Druga opcja polega na przechodzeniu wątku oczekującego w stan uśpienia na krótkie momenty i okresowym wykonywaniu testów za pomocą funkcji `std::this_thread::sleep_for()` (patrz punkt 4.3):

```
bool flag;
std::mutex m;

void wait_for_flag()
{
    std::unique_lock<std::mutex> lk(m);
    while(!flag)
    {
        lk.unlock();           ← ❶ Odblokowuje muteks
        std::this_thread::sleep_for(std::chrono::milliseconds(100)); ← ❷ Czeka 100 ms
        lk.lock();           ← ❸ Ponownie blokuje muteks
    }
}
```

Wywołanie funkcji w pętli odblokowuje muteks ❶ przed przejściem w stan uśpienia ❷ i ponownie blokuje ten muteks po wyjściu z tego stanu ❸ — dzięki temu drugi wątek ma szansę uzyskania dostępu do flagi i jej ustawienia.

Opisane rozwiązanie jest o tyle dobre, że uśpiiony wątek nie zajmuje bezproduktywnie czasu procesora. Warto jednak pamiętać, że dobór właściwego czasu uśpienia

jest dość trudny. Zbyt krótki czas przebywania w tym stanie spowoduje, że wątek będzie tracił czas procesora na zbyt częste testy; zbyt długi czas uśpienia będzie oznaczał, że wątek będzie przebywał w tym stanie nawet po zakończeniu zadania, na które oczekuje, zatem opóźnienie w działaniu wątku oczekującego będzie zbyt duże. Takie „zaspanie” wątku rzadko ma bezpośredni wpływ na wynik operacji wykonywanych przez program, ale już w przypadku szybkiej gry może powodować pominięcie niektórych klatek animacji, a w przypadku aplikacji czasu rzeczywistego może oznaczać pominięcie przydziału czasu procesora.

Trzecim, najlepszym rozwiązaniem jest użycie gotowych elementów biblioteki standardowej języka C++ umożliwiających oczekiwanie na określone zdarzenie. Najprostszym mechanizmem oczekiwania na zdarzenie generowane przez inny wątek (na przykład zdarzenie polegające na umieszczeniu dodatkowego zadania w potoku) jest tzw. **zmienna warunkowa**. Zmienna warunkowa jest powiązana z pewnym zdarzeniem lub **warunkiem** oraz co najmniej jednym wątkiem, który **czeka** na spełnienie tego warunku. Wątek, który odkrywa, że warunek jest spełniony, może **powiadomić** pozostałe wątki oczekujące na tę zmienną warunkową, aby je obudzić i umożliwić im dalsze przetwarzanie.

#### 4.1.1. Oczekiwanie na spełnienie warunku za pomocą zmiennych warunkowych

Biblioteka standardowa języka C++ udostępnia **dwie** implementacje mechanizmu zmiennych warunkowych w formie klas `std::condition_variable` i `std::condition_variable_any`. Obie klasy zostały zadeklarowane w pliku nagłówkowym `<condition_variable>`. W obu przypadkach zapewnienie właściwej synchronizacji wymaga użycia muteksu — pierwsza klasa jest przystosowana tylko do obsługi muteksów typu `std::mutex`, natomiast druga klasa obsługuje wszystkie rodzaje muteksów spełniających pewien minimalny zbiór kryteriów (stąd przyrostek `_any`). Ponieważ klasa `std::condition_variable_any` jest bardziej uniwersalna, z jej stosowaniem wiążą się dodatkowe koszty w wymiarze wielkości, wydajności i zasobów systemu operacyjnego. Jeśli więc nie potrzebujemy dodatkowej elastyczności, powinniśmy stosować klasę `std::condition_variable`.

Jak należałoby użyć klasy `std::condition_variable` do obsługi przykładu opisanego na początku tego podrozdziału — jak sprawić, że wątek oczekujący na wykonanie jakiegoś zadania będzie uśpiony do momentu, w którym będą dostępne dane do przetworzenia? Na listingu 4.1 pokazano przykład kodu implementującego odpowiednie rozwiązanie przy użyciu zmiennej warunkowej.

**Listing 4.1. Oczekiwanie na dane do przetworzenia za pomocą klasy `std::condition_variable`**

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread()
{
    while(more_data_to_prepare())
    {
        data_chunk const data=prepare_data();
```



```

std::lock_guard<std::mutex> lk(mut);
data_queue.push(data); ← ❷
data_cond.notify_one(); ← ❸
}
}

void data_processing_thread()
{
    while(true)
    {
        std::unique_lock<std::mutex> lk(mut); ← ❹
        data_cond.wait(
            lk, []{return !data_queue.empty();}); ← ❺
        data_chunk data=data_queue.front();
        data_queue.pop();
        lk.unlock(); ← ❻
        process(data);
        if(is_last_chunk(data))
            break;
    }
}

```

Na początku kodu zdefiniowano kolejkę ❶, która będzie używana do przekazywania danych pomiędzy dwoma wątkami. Kiedy dane są gotowe do przetworzenia, wątek, który je przygotował, blokuje muteks chroniący kolejkę za pomocą klasy `std::lock_guard` i umieszcza nowe dane w kolejce ❷. Wątek wywołuje następnie funkcję składową `notify_one()` dla obiektu klasy `std::condition_variable`, aby powiadomić oczekujący wątek (jeśli taki istnieje) o dostępności nowych danych ❸.

W tym modelu drugą stroną komunikacji jest wątek przetwarzający te dane. Wątek przetwarzający najpierw blokuje muteks, jednak tym razem użyto do tego celu klasy `std::unique_lock` zamiast klasy `std::lock_guard` ❹ — przyczyny tej decyzji zostaną wyjaśnione za chwilę. Wątek wywołuje następnie funkcję `wait()` dla obiektu klasy `std::condition_variable`. Na wejściu tego wywołania wątek przekazuje obiekt blokady i funkcję lambda reprezentującą warunek, który musi zostać spełniony przed przystąpieniem do dalszego przetwarzania ❺. Funkcje lambda to stosunkowo nowy element (wprowadzony w standardzie C++11), który umożliwia pisanie funkcji anonimowych w ramach innych wyrażeń. Wspomniane rozwiązanie wprost idealnie nadaje się do wskazywania predykatów w wywołaniach takich funkcji biblioteki standardowej jak `wait()`. W tym przypadku prosta funkcja lambda `[]{return !data_queue.empty();}` sprawdza, czy struktura reprezentowana przez zmienną `data_queue` nie jest pusta, tj. czy kolejka zawiera jakieś dane gotowe do przetworzenia. Funkcje lambda zostaną szczegółowo omówione w części A.5 dodatku A.

Implementacja funkcji `wait()` sprawdza warunek (wywołując przekazaną funkcję lambda), po czym zwraca sterowanie, jeśli ten warunek jest spełniony (jeśli funkcja lambda zwróciła wartość `true`). Jeśli warunek nie jest spełniony (jeśli funkcja lambda zwróciła wartość `false`), funkcja `wait()` odblokowuje muteks i wprowadza bieżący wątek w stan blokady (oczekiwania). Kiedy zmienna warunkowa jest powiadamiana za pomocą funkcji `notify_one()` wywołanej przez wątek przygotowujący dane, wątek oczekujący jest budzony (odblokowywany), ponownie uzyskuje blokadę muteksu i jeszcze raz sprawdza warunek. Jeśli warunek dalszego przetwarzania jest spełniony, funkcja `wait()` zwraca

sterowanie z zachowaniem blokady muteksu. Jeśli warunek nie jest spełniony, wątek odblokowuje muteks i ponownie przechodzi w stan oczekiwania. Właśnie dlatego w przykładzie należało użyć klasy `std::unique_lock` zamiast klasy `std::lock_guard` — wątek oczekujący musi odblokować muteks na czas oczekiwania i zablokować go ponownie po otrzymaniu powiadomienia, a klasa `std::lock_guard` nie zapewnia takiej elastyczności. Gdyby muteks pozostał zablokowany przez cały czas uśpienia tego wątku, wątek przygotowujący dane nie mógłby zablokować tego muteksu i dodać elementu do kolejki, zatem warunek budzenia wątku oczekującego nigdy nie zostałby spełniony.

Na listingu 4.1 użyłem prostej funkcji lambda 5, która sprawdza, czy struktura kolejki nie jest pusta. Okazuje się, że w tej roli również dobrze można by użyć dowolnej funkcji lub obiektu wywoływalnego. Jeśli programista dysponuje już funkcją sprawdzającą odpowiedni warunek (funkcja może oczywiście być nieporównanie bardziej złożona niż prosty test z powyższego przykładu), może przekazać tę funkcję bezpośrednio na wejściu funkcji `wait()`, bez konieczności opakowywania jej w ramach wyrażenia lambda. Po wywołaniu funkcji `wait()` zmienna warunkowa może sprawdzić wskazany warunek na wiele różnych sposobów, jednak podczas tego testu muteks zawsze jest zablokowany, a funkcja `wait()` natychmiast zwraca sterowanie, pod warunkiem że przekazana funkcja sprawdzająca ten warunek zwróciła wartość `true`. Jeśli wątek oczekujący ponownie uzyskuje muteks i sprawdza warunek, mimo że nie otrzymał powiadomienia od innego wątku i jego działania nie są bezpośrednią odpowiedzią na takie powiadomienie, mamy do czynienia z tzw. **pozornym budzeniem** (ang. *spurious wake*). Ponieważ optymalna liczba i częstotliwość takich pozornych budzeń są z definicji trudne do oszacowania, funkcja sprawdzająca prawdziwość warunku nie powinna powodować żadnych skutków ubocznych. Gdyby ta funkcja powodowała skutki uboczne, programista musiałby przygotować swój kod na wielokrotne występowanie tych skutków przed spełnieniem warunku.

Możliwość odblokowania obiektu klasy `std::unique_lock` nie jest używana tylko dla wywołania funkcji `wait()` — analogiczne rozwiązanie zastosowaliśmy po uzyskaniu danych do przetworzenia, ale przed przystąpieniem do właściwego przetwarzania 6. Przetwarzanie danych może być czasochłonną operacją, a jak wiemy z rozdziału 3., utrzymywanie blokady muteksu dłużej, niż to konieczne, nie jest dobrym rozwiązaniem.

Stosowanie struktury kolejki do przekazywania danych pomiędzy wątkami (jak na listingu 4.1) jest dość typowym rozwiązaniem. Jeśli projekt aplikacji jest właściwy, synchronizacja powinna dotyczyć samej kolejki, co znacznie ogranicza liczbę potencjalnych problemów i problematycznych sytuacji wyścigu. Spróbujmy więc wyodrębnić z listingu 4.1 uniwersalną kolejkę gwarantującą bezpieczne przetwarzanie wielowątkowe.

#### 4.1.2. **Budowa kolejki gwarantującej bezpieczne przetwarzanie wielowątkowe przy użyciu zmiennych warunkowych**

Przed przystąpieniem do projektowania uniwersalnej kolejki warto poświęcić kilka minut analizie operacji, które trzeba będzie zaimplementować dla tej struktury danych (podobnie jak w przypadku stosu gwarantującego bezpieczeństwo przetwarzania wielowątkowego z punktu 3.2.3). Przyjrzyjmy się kontenerowi `std::queue<>` dostępnemu w bibliotece standardowej języka C++ (patrz listing 4.2), który będzie stanowił punkt wyjścia dla naszej implementacji.

Listing 4.2. Interfejs kontenera `std::queue`

```

template <class T, class Container = std::deque<T> >
class queue {
public:
    explicit queue(const Container&);
    explicit queue(Container&& = Container());

    template <class Alloc> explicit queue(const Alloc&);
    template <class Alloc> queue(const Container&, const Alloc&);
    template <class Alloc> queue(Container&&, const Alloc&);
    template <class Alloc> queue(queue&&, const Alloc&);

    void swap(queue& q);

    bool empty() const;
    size_type size() const;

    T& front();
    const T& front() const;
    T& back();
    const T& back() const;

    void push(const T& x);
    void push(T&& x);
    void pop();
    template <class... Args> void emplace(Args&&... args);
};

```

Jeśli pominiemy operacje konstruowania, przypisywania i wymiany, pozostaną nam zaledwie trzy grupy operacji: operacje zwracające stan całej kolejki (`empty()` i `size()`), operacje zwracające pojedyncze elementy kolejki (`front()` i `back()`) oraz operacje modyfikujące kolejkę (`push()`, `pop()` i `emplace()`). Mamy więc do czynienia z sytuacją analogiczną do tej opisanej w punkcie 3.2.3 (gdzie omawialiśmy strukturę stosu), zatem opisany interfejs jest narażony na te same problemy związane z sytuacjami wyścigów. W tym przypadku należy połączyć funkcje `front()` i `pop()` w jedno wywołanie, tak jak wcześniej połączyliśmy funkcje `top()` i `pop()` dla struktury stosu. Warto jeszcze zwrócić uwagę na pewien nowy element w kodzie z listingu 4.1 — podczas używania kolejki do przekazywania danych pomiędzy wątkami wątek docelowy zwykle musi czekać na te dane. Warto więc zaimplementować funkcję `pop()` w dwóch wersjach — pierwsza funkcja, `try_pop()`, próbuje pobrać wartość z kolejki, ale zawsze zwraca sterowanie bezpośrednio po wywołaniu, nawet jeśli kolejka nie zawierała żadnej wartości (wtedy funkcja sygnalizuje błąd); druga funkcja, `wait_and_pop()`, czeka na pojawienie się w kolejce wartości do pobrania. Po wprowadzeniu zmian zgodnie ze schematem opisanym już przy okazji przykładu stosu interfejs struktury kolejki powinien wyglądać tak jak na listingu 4.3.

Listing 4.3. Interfejs struktury danych `threadsafe_queue`

```

#include <memory> ← Dla typu std::shared_ptr

template<typename T>
class threadsafe_queue

```

```

{
public:
    threadsafe_queue();
    threadsafe_queue(const threadsafe_queue&);
    threadsafe_queue& operator=(
        const threadsafe_queue&) = delete; ← Dla uproszczenia wyklucza możliwość
                                                przypisywania

    void push(T new_value);

    bool try_pop(T& value); ← ❶
    std::shared_ptr<T> try_pop(); ← ❷

    void wait_and_pop(T& value);
    std::shared_ptr<T> wait_and_pop();

    bool empty() const;
};

```

Podobnie jak w przypadku stosu, na listingu 4.3 usunięto konstruktory i operator przypisania, aby uprościć analizowany kod. Tak jak wcześniej, także tym razem funkcje `try_pop()` i `wait_for_pop()` występują w dwóch wersjach. Pierwsza przeciążona wersja funkcji `try_pop()` ❶ zapisuje pobraną wartość we wskazywanej zmiennej, tak aby można było użyć tej wartości w roli statusu; funkcja zwraca wartość `true`, jeśli uzyskała jakąś wartość — w przeciwnym razie funkcja zwraca wartość `false` (patrz część A.2 dodatku A). Druga przeciążona wersja ❷ nie może działać w ten sam sposób, ponieważ natychmiast zwraca uzyskaną wartość. Jeśli jednak funkcja nie uzyskała żadnej wartości, może zwrócić wskaźnik równy `NULL`.

Jaki to ma związek z listingiem 4.1? Okazuje się, że możemy wyodrębnić kod funkcji `push()` i `wait_and_pop()` z tamtego listingu i na tej podstawie przygotować nową implementację (patrz listing 4.4).

#### Listing 4.4. Funkcje `push()` i `wait_and_pop()` wyodrębnione z listingu 4.1

```

#include <queue>
#include <mutex>
#include <condition_variable>

template<typename T>
class threadsafe_queue
{
private:
    std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    void push(T new_value)
    {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(new_value);
        data_cond.notify_one();
    }

    void wait_and_pop(T& value)
    {

```

```

        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        value=data_queue.front();
        data_queue.pop();
    }
};

threadsafe_queue<data_chunk> data_queue; ← ❶

void data_preparation_thread()
{
    while(more_data_to_prepare())
    {
        data_chunk const data=prepare_data();
        data_queue.push(data); ← ❷
    }
}

void data_processing_thread()
{
    while(true)
    {
        data_chunk data;
        data_queue.wait_and_pop(data); ← ❸
        process(data);
        if(is_last_chunk(data))
            break;
    }
}

```

Muteks i zmienna warunkowa są teraz elementami składowymi obiektu klasy `threadsafe_queue`, zatem nie jest potrzebne stosowanie odrębnych zmiennych ❶, a wywołanie funkcji `push()` nie wymaga zewnętrznych mechanizmów synchronizacji ❷. Jak widać, także funkcja `wait_and_pop()` uwzględnia stan zmiennej warunkowej ❸.

Napisanie drugiej wersji przeciążonej funkcji `wait_and_pop()` nie stanowi żadnego problemu; także pozostałe funkcje można niemal skopiować z przykładu stosu pokazanego na listingu 3.5. Ostateczną wersję implementacji kolejki pokazano na listingu 4.5.

**Listing 4.5. Kompletna definicja klasy kolejki gwarantującej bezpieczeństwo przetwarzania wielowątkowego (dzięki użyciu zmiennych warunkowych)**

```

#include <queue>
#include <memory>
#include <mutex>
#include <condition_variable>

template<typename T>
class threadsafe_queue
{
private:
    mutable std::mutex mut; ← ❶ Muteks musi być modyfikowalny
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue()
    {}

```



```

threadsafe_queue(threadsafe_queue const& other)
{
    std::lock_guard<std::mutex> lk(other.mut);
    data_queue=other.data_queue;
}

void push(T new_value)
{
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(new_value);
    data_cond.notify_one();
}

void wait_and_pop(T& value)
{
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[this]{return !data_queue.empty();});
    value=data_queue.front();
    data_queue.pop();
}

std::shared_ptr<T> wait_and_pop()
{
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[this]{return !data_queue.empty();});
    std::shared_ptr<T> res(std::make_shared<T>(data_queue.front()));
    data_queue.pop();
    return res;
}

bool try_pop(T& value)
{
    std::lock_guard<std::mutex> lk(mut);
    if(data_queue.empty())
        return false;
    value=data_queue.front();
    data_queue.pop();
    return true;
}

std::shared_ptr<T> try_pop()
{
    std::lock_guard<std::mutex> lk(mut);
    if(data_queue.empty())
        return std::shared_ptr<T>();
    std::shared_ptr<T> res(std::make_shared<T>(data_queue.front()));
    data_queue.pop();
    return res;
}

bool empty() const
{
    std::lock_guard<std::mutex> lk(mut);
    return data_queue.empty();
}
};

```

Mimo że `empty()` jest stałą funkcją składową i mimo że parametr `other` konstruktora kopiującego jest stałą referencją, pozostałe wątki mogą dysponować niestałymi referencjami do tego obiektu i wywoływać funkcje składowe zmieniające jego stan, zatem blokowanie muteksu wciąż jest konieczne. Ponieważ blokowanie muteksu jest operacją zmieniającą stan obiektu, obiekt muteksu należy oznaczyć jako modyfikowalny (ang. *mutable*) **1**, tak aby można było blokować ten muteks w ciele funkcji `empty()` i konstruktora kopiującego.

Zmienne warunkowe są przydatne także w sytuacji, w której wiele wątków czeka na to samo zdarzenie. Jeśli celem stosowania wątków jest dzielenie obciążenia i jeśli tylko jeden wątek powinien reagować na powiadomienie, można zastosować dokładnie taką samą strukturę jak ta z listingu 4.1 — wystarczy uruchomić wiele instancji wątku przetwarzającego dane. Po przygotowaniu nowych danych wywołanie funkcji `notify_one()` spowoduje, że jeden z wątków aktualnie wykonujących funkcję `wait()` sprawdzi warunek. Ponieważ do struktury `data_queue` właśnie dodano nowe dane, funkcja `wait()` zwróci sterowanie. Nie wiadomo, do którego wątku trafi powiadomienie ani nawet czy istnieje wątek oczekujący na to powiadomienie (nie można przecież wykluczyć, że wszystkie wątki w danej chwili przetwarzają swoje dane).

Warto też pamiętać o możliwości oczekiwania na to samo zdarzenie przez wiele wątków, z których każdy musi zareagować na powiadomienie. Opisany scenariusz może mieć związek z inicjalizacją współdzielonych danych, gdzie wszystkie wątki przetwarzające operują na tych samych danych i muszą czekać albo na ich inicjalizację (w takim przypadku istnieją lepsze mechanizmy — patrz punkt 3.3.1 w rozdziale 3.), albo na ich aktualizację (na przykład w ramach okresowej, wielokrotnej inicjalizacji). W opisanych przypadkach wątek przygotowujący dane może wywołać funkcję składową `notify_all()` dla zmiennej warunkowej (zamiast funkcji `notify_one()`). Jak nietrudno się domyślić, funkcja powoduje, że **wszystkie** wątki aktualnie wykonujące funkcję `wait()` sprawdzą warunek, na który czekają.

Jeśli wątek wywołujący w założeniu ma oczekiwać na dane zdarzenie tylko raz, czyli jeśli po spełnieniu warunku wątek nie będzie ponownie czekał na tę samą zmienną warunkową, być może warto zastosować inny mechanizm synchronizacji niż zmienna warunkowa. Zmienne warunkowe są szczególnie nieefektywne w sytuacji, gdy warunkiem, na który oczekują wątki, jest dostępność określonego elementu danych. W takim przypadku lepszym rozwiązaniem jest użycie mechanizmu **przyszłości**.

## 4.2. Oczekiwanie na jednorazowe zdarzenia za pomocą przyszłości

Przypuśćmy, że planujemy podróż samolotem. Po przyjeździe na lotnisko i przejściu rozmaitych procedur wciąż musimy czekać na komunikat dotyczący gotowości naszego samolotu na przyjęcie pasażerów (zdarza się, że pasażerowie muszą czekać wiele godzin). Możemy oczywiście znaleźć sposób, aby ten czas minął nieco szybciej (możemy na przykład czytać książkę, przeglądać strony internetowe lub udać się na posiłek do drogiej lotniskowej kawiarni), jednak niezależnie od sposobu spędzania czasu czekamy na jedno — sygnał wzywający do udania się na pokład samolotu. Co więcej, interesujący nas lot odbędzie się tylko raz, zatem przy okazji następnego wyjazdu na wakacje będziemy czekali na inny lot.

Twórcy biblioteki standardowej języka C++ rozwiązali problem jednorazowych zdarzeń za pomocą mechanizmu nazwanego **przyszłością** (ang. *future*). Wątek, który musi czekać na określone jednorazowe zdarzenie, powinien uzyskać przyszłość reprezentującą to zdarzenie. Wątek oczekujący na tę przyszłość może następnie okresowo sprawdzać, czy odpowiednio zdarzenie nie nastąpiło (tak jak pasażerowie co jakiś czas zerkają na tablicę odlotów), i jednocześnie pomiędzy tymi testami wykonywać inne zadanie (spożywać drogi deser w lotniskowej kawiarni). Alternatywnym rozwiązaniem jest wykonywanie innego zadania do momentu, w którym dalsze działanie nie jest możliwe bez określonego zdarzenia, i przejście w stan **gotowości** na przyszłość. Przyszłość może, ale nie musi być powiązana z danymi (tak jak tablica odlotów może wskazywać rękawy prowadzące do właściwych samolotów). Po wystąpieniu zdarzenia (po osiągnięciu **gotowości** przez przyszłość) nie jest możliwe wyzerowanie tej przyszłości.

W bibliotece standardowej języka C++ istnieją dwa rodzaje przyszłości zaimplementowane w formie dwóch szablonów klas zadeklarowanych w nagłówku biblioteki `<future>`: **przyszłości unikatowe** (`std::future<>`) oraz **przyszłości współdzielone** (`std::shared_future<>`). Wymienione klasy opracowano na bazie typów `std::unique_ptr` i `std::shared_ptr`. Obiekt typu `std::future` jest jedyną instancją odwołującą się do powiązanego zdarzenia, natomiast do jednego zdarzenia może się odwoływać wiele instancji typu `std::shared_future`. W drugim przypadku wszystkie instancje są **gotowe** jednocześnie i wszystkie mogą uzyskiwać dostęp do dowolnych danych powiązanych z danym zdarzeniem. Właśnie z myślą o powiązanych danych zaprojektowano te szablony klas — tak jak w przypadku szablonów `std::unique_ptr` i `std::shared_ptr`, parametry szablonów `std::future<>` i `std::shared_future<>` reprezentują właśnie typy powiązanych danych. W razie braku powiązanych danych należy stosować następujące specjalizacje tych szablonów: `std::future<void>` i `std::shared_future<void>`. Mimo że przyszłości służą do komunikacji pomiędzy wątkami, same obiekty przyszłości nie oferują mechanizmów synchronizowanego dostępu. Jeśli wiele wątków potrzebuje dostępu do jednego obiektu przyszłości, należy chronić ten dostęp za pomocą muteksu lub innego mechanizmu synchronizacji (patrz rozdział 3.). Jak napiszę w punkcie 4.2.5 w dalszej części tego podrozdziału, wiele wątków może uzyskiwać dostęp do własnej kopii obiektu typu `std::shared_future<>` bez konieczności dodatkowej synchronizacji, nawet jeśli wszystkie te kopie odwołują się do tego samego asynchronicznego wyniku.

Najprostszym przykładem jednorazowego zdarzenia jest wynik obliczeń wykonywanych w tle. Już w rozdziale 2. napisałem, że klasa `std::thread` nie udostępnia prostych mechanizmów zwracania wartości wynikowych dla tego rodzaju zadań, i zapowiedziałem wprowadzenie odpowiednich rozwiązań w rozdziale 4. przy okazji omawiania przyszłości — czas zapoznać się z tymi rozwiązaniami.

#### 4.2.1. Zwracanie wartości przez zadania wykonywane w tle

Przypuśćmy, że nasza aplikacja wykonuje czasochłonne obliczenia, które ostatecznie pozwolą uzyskać oczekiwany wynik. Załóżmy, że wartość wynikowa nie jest potrzebna na tym etapie działania programu. Być może udało nam się wymyślić sposób poszukiwania odpowiedzi na pytanie o życie, wszechświat i całą resztę stawiane w książkach

Douglasa Adamsa<sup>1</sup>. Moglibyśmy oczywiście uruchomić nowy wątek, który wykona niezbędne obliczenia, jednak takie rozwiązanie wiązałoby się z koniecznością przekazania wyników z powrotem do wątku głównego, ponieważ klasa `std::thread` nie oferuje alternatywnego mechanizmu zwracania wartości wynikowych. W takim przypadku sporym ułatwieniem jest użycie szablonu funkcji `std::async` (zadeklarowanego w pliku nagłówkowym `<future>`).

**Asynchroniczne zadanie**, którego wynik nie jest potrzebny na bieżącym etapie działania programu, można rozpocząć za pomocą funkcji `std::async`. Zamiast zwracania obiektu klasy `std::thread`, który umożliwia oczekiwanie na zakończenie danego wątku, funkcja `std::async` zwraca obiekt klasy `std::future`, który w przyszłości będzie zawierał wartość wynikową. W miejscu, w którym aplikacja będzie potrzebowała tej wartości, należy wywołać funkcję `get()` dla obiektu przyszłości — wywołanie tej funkcji zablokuje wykonywanie bieżącego wątku do momentu osiągnięcia **gotowości** przez przyszłość, po czym zwróci uzyskaną wartość. Prosty przykład użycia tych elementów pokazano na listingu 4.6.

**Listing 4.6. Przykład użycia szablonu klasy `std::future` do uzyskania wartości wynikowej asynchronicznego zadania**

```
#include <future>
#include <iostream>

int find_the_answer_to_ltuae();
void do_other_stuff();
int main()
{
    std::future<int> the_answer=std::async(find_the_answer_to_ltuae);
    do_other_stuff();
    std::cout<<"Odpowiedź brzmi "<<the_answer.get()<<std::endl;
}
```

Szablon funkcji `std::async` umożliwia przekazywanie dodatkowych argumentów na wejściu wywoływanej funkcji — wystarczy dodać te argumenty do wywołania (podobnie jak w przypadku klasy `std::thread`). Jeśli pierwszy argument reprezentuje wskaźnik do funkcji składowej, drugi argument zawiera obiekt, dla którego ma zostać wywołana ta funkcja składowa (bezpośrednio, za pośrednictwem wskaźnika lub poprzez opakowanie `std::ref`), a pozostałe argumenty są przekazywane na wejściu tej funkcji składowej. W przeciwnym razie drugi i kolejne argumenty są przekazywane na wejściu funkcji składowej lub wywoływalnego obiektu wskazanego za pośrednictwem pierwszego argumentu. Tak jak w przypadku klasy `std::thread`, jeśli argumenty mają postać r-wartości, zostaną utworzone kopie poprzez **przeniesienie** oryginalnych wartości. Dzięki temu możemy stosować typy oferujące tylko możliwość przenoszenia zarówno w roli obiektów funkcji, jak i w roli argumentów. Przykład takiego rozwiązania pokazano na listingu 4.7.

<sup>1</sup> W książce *Autostopem przez Galaktykę* zbudowano komputer Deep Thought, który miał odpowiedzieć na pytanie o życie, wszechświat i całą resztę. Odpowiedzią na to pytanie była liczba 42.

Listing 4.7. Przekazywanie argumentów na wejściu funkcji wątku `std::async`

```

#include <string>
#include <future>

struct X
{
    void foo(int,std::string const&);
    std::string bar(std::string const&);
};
X x;
auto f1=std::async(&X::foo,&x,42,"witaj");
auto f2=std::async(&X::bar,x,"żegnaj");
struct Y
{
    double operator()(double);
};
Y y;
auto f3=std::async(Y(),3.141);
auto f4=std::async(std::ref(y),2.718);
X baz(X&);
std::async(baz,std::ref(x));
class move_only
{
public:
    move_only();
    move_only(move_only&&);
    move_only(move_only const&) = delete;
    move_only& operator=(move_only&&);
    move_only& operator=(move_only const&) = delete;

    void operator()();
};
auto f5=std::async(move_only());

```

**Wywołuje p->foo(42,"witaj"),  
gdzie p jest reprezentowane przez &x**

**Wywołuje tmpx.bar("żegnaj"),  
gdzie tmpx jest kopią x**

**Wywołuje tmpy(3.141), gdzie tmpy  
jest tworzone za pomocą konstruktora  
przenoszącego Y()**

**Wywołuje y(2.718)**

**Wywołuje baz(x)**

**Wywołuje tmp(), gdzie tmp jest konstruowany  
na podstawie wywołania std::move(move\_only())**

Domyślnie to od stosowanej implementacji zależy, czy funkcja `std::async` uruchamia nowy wątek, czy wskazane zadanie będzie wykonywane w sposób synchroniczny (wówczas bieżący wątek będzie czekał na osiągnięcie gotowości przez przyszłość). W większości przypadków standardowe rozwiązanie jest wystarczające, jednak programista może wybrać właściwy tryb za pomocą dodatkowego parametru funkcji `std::async` przekazywanego przed funkcją do wywołania. Wspomniany parametr typu `std::launch` może mieć albo wartość `std::launch::deferred` (wówczas wywołanie funkcji jest odkładane do momentu wywołania funkcji `wait()` lub `get()` dla danej przyszłości), albo wartość `std::launch::async` (wówczas funkcja musi być wykonywana w odrębnym wątku), albo wartość `std::launch::deferred | std::launch::async` (wówczas decyzja należy do implementacji). Ostatnia opcja jest stosowana w roli wartości domyślnej. Jeśli wywołanie funkcji jest odkładane na przyszłość, może nigdy nie nastąpić. Na przykład:

```

auto f6=std::async(std::launch::async,Y(),1.2);
auto f7=std::async(std::launch::deferred,baz,std::ref(x));
auto f8=std::async(
    std::launch::deferred | std::launch::async,
    baz,std::ref(x));
auto f9=std::async(baz,std::ref(x));
f7.wait();

```

**Wykonywane w nowym wątku**

**Wykonywane w ramach funkcji wait() lub get()**

**Wybór implementacji**

**Wywołanie odroczonej funkcji**

Jak się przekonasz w dalszej części tego rozdziału (i ponownie w rozdziale 8.), funkcja `std::async` ułatwia dzielenie algorytmów na współbieżnie wykonywane zadania. Okazuje się jednak, że nie jest to jedyny sposób kojarzenia obiektu typu `std::future` z zadaniem — alternatywnym rozwiązaniem jest opakowanie zadania w ramach instancji szablonu klasy `std::packaged_task<>` lub napisanie kodu bezpośrednio ustawiającego wartości za pomocą szablonu klasy `std::promise<>`. Szablon klasy `std::packaged_task` jest abstrakcją wyższego poziomu (w porównaniu z szablonem `std::promise`), zatem właśnie ten szablon omówimy jako pierwszy.

#### 4.2.2. Wiązanie zadania z przyszłością

Szablon klasy `std::packaged_task<>` wiąże przyszłość z funkcją lub wywoływalnym obiektem. W momencie wywołania obiektu typu `std::packaged_task<>` wywołana zostaje powiązana funkcja lub wywoływalny obiekt, a sama przyszłość przechodzi w stan **gotowości** (wartość wynikowa zostaje umieszczona w powiązanych danych). Opisaną strukturę można wykorzystać w roli elementu składowego podczas budowy puli wątków (patrz rozdział 9.) lub dowolnego innego schematu zarządzania zadaniami polegającego na przykład na wykonywaniu każdego zadania w osobnym wątku lub sekwencyjnym wykonywaniu zadań w jednym wątku działającym w tle. Jeśli jedną większą operację można podzielić na wiele autonomicznych podzadań, każde z tych podzadań można opakować w ramach obiektu klasy `std::packaged_task<>`, aby następnie przekazać ten obiekt do mechanizmu szeregowania zadań lub do puli wątków. W ten sposób można skutecznie ukryć szczegóły związane z poszczególnymi zadaniami — mechanizm szeregowania zadań operuje na obiektach klasy `std::packaged_task<>`, nie na poszczególnych funkcjach.

Parametr szablonu klasy `std::packaged_task<>` reprezentuje sygnaturę funkcji — na przykład dla funkcji, która nie otrzymuje żadnych parametrów i nie zwraca wartości, należałoby użyć sygnatury `void()`, natomiast dla funkcji otrzymującej niestałą referencję do wartości typu `std::string` i wskaźnik do wartości typu `double` oraz zwracającej wartość typu `int` należałoby użyć sygnatury `int(std::string&, double*)`. Podczas konstruowania obiektu klasy `std::packaged_task` należy przekazać funkcję (lub wywoływalny obiekt) otrzymującą na wejściu wskazane parametry i zwracającą typ, który można przekonwertować na wskazany typ danych. Dokładne dopasowanie typów nie jest wymagane — istnieje możliwość skonstruowania obiektu klasy `std::packaged_task` `↳<double(double)>` na podstawie funkcji otrzymującej na wejściu wartość typu `int` i zwracającej wartość typu `float`, ponieważ wymienione typy mogą być automatycznie konwertowane.

Typ wartości zwracanych przez wskazaną funkcję identyfikuje typ zwracany przez funkcję składową `get_future()` konstruowanego obiektu klasy `std::future<>`, natomiast lista argumentów zdefiniowana w ramach sygnatury funkcji jest używana do wyznaczenia sygnatury operatora wywołania funkcji zadania reprezentowanego przez ten obiekt. Przykład częściowej definicji klasy `std::packaged_task<std::string(std::vector<char>*, int)>` pokazano na listingu 4.8.

Instancja klasy `std::packaged_task` jest obiektem wywoływalnym i jako taka może być opakowana w ramach obiektu klasy `std::function`, przekazana do obiektu klasy `std::thread` w roli funkcji wątku, przekazana do dowolnej innej funkcji oczekującej wywoływalnego obiektu, a nawet bezpośrednio wywołana. W momencie wywołania

**Listing 4.8. Częściowa definicja specjalizacji szablonu klasy `std::packaged_task<>`**

```

template<>
class packaged_task<std::string(std::vector<char>*,int)>
{
public:
    template<typename Callable>
    explicit packaged_task(Callable&& f);
    std::future<std::string> get_future();
    void operator()(std::vector<char>*,int);
};

```

obiektu klasy `std::packaged_task` jako obiektu funkcji argumenty przekazane na wejściu operatora wywołania są przekazywane do opakowanej funkcji, a zwracana wartość jest zapisywana jako asynchroniczny wynik w obiekcie typu `std::future` (obiekt można następnie uzyskać za pomocą funkcji `get_future()`). Oznacza to, że możemy opakować zadanie w obiekcie klasy `std::packaged_task` i uzyskać przyszłość przed przekazaniem tego obiektu do miejsca, gdzie zostanie wywołany. W momencie, w którym program będzie potrzebował wyniku, wystarczy poczekać na osiągnięcie gotowości przez tę przyszłość. Praktyczny przykład takiego rozwiązania opisano w następnym podpunkcie.

### PRZEKAZYWANIE ZADAŃ POMIĘDZY WĄTKAMI

Wiele frameworków graficznego interfejsu użytkownika wymaga, aby aktualizacje tego interfejsu były wykonywane przez określone wątki. Oznacza to, że jeśli jakiś inny wątek musi zaktualizować graficzny interfejs użytkownika, powinien wysłać komunikat do właściwego wątku, aby wyznaczony wątek wykonał to zadanie w jego imieniu. Szablon klasy `std::packaged_task` oferuje odpowiednie rozwiązania bez konieczności stosowania niestandardowych komunikatów dla każdego zadania związanego z działaniem graficznego interfejsu użytkownika (patrz listing 4.9).

**Listing 4.9. Uruchamianie kodu w wątku graficznego interfejsu użytkownika za pomocą szablonu klasy `std::packaged_task`**

```

#include <deque>
#include <mutex>
#include <future>
#include <thread>
#include <utility>

std::mutex m;
std::deque<std::packaged_task<void()> > tasks;

bool gui_shutdown_message_received();
void get_and_process_gui_message();

void gui_thread() ← ❶
{
    while(!gui_shutdown_message_received()) ← ❷
    {
        get_and_process_gui_message(); ← ❸
        std::packaged_task<void()> task;
    }
}

```

```

std::lock_guard<std::mutex> lk(m);
if(tasks.empty()) ← 4
    continue;
task=std::move(tasks.front()); ← 5
tasks.pop_front();
}
task(); ← 6
}
}

std::thread gui_bg_thread(gui_thread);

template<typename Func>
std::future<void> post_task_for_gui_thread(Func f)
{
    std::packaged_task<void()> task(f); ← 7
    std::future<void> res=task.get_future(); ← 8
    std::lock_guard<std::mutex> lk(m);
    tasks.push_back(std::move(task)); ← 9
    return res; ← 10
}

```

Powyższy kod jest bardzo prosty: wątek graficznego interfejsu użytkownika ❶ działa w pętli do momentu otrzymania komunikatu sygnalizującego konieczność zamknięcia tego interfejsu ❷. W ciele tej pętli wątek sprawdza komunikaty dotyczące graficznego interfejsu użytkownika ❸ (na przykład tego, że użytkownik kliknął jakiś element interfejsu) oraz ewentualne zadania w kolejce zadań. Jeśli kolejka nie zawiera żadnych zadań ❹, wątek przechodzi do następnej iteracji pętli; w przeciwnym razie wątek odczytuje zadanie z kolejki ❺, zwalnia blokadę tej kolejki, po czym uruchamia to zadanie ❻. W momencie zakończenia zadania powiązana z nim przyszłość przechodzi w stan gotowości.

Umieszczenie zadania w kolejce jest równie proste: nowe, opakowane zadanie jest tworzone na podstawie wskazanej funkcji ❷, przyszłość jest uzyskiwana z obiektu zadania ❸ za pomocą funkcji składowej `get_future()` i wreszcie zadanie jest umieszczane na liście ❹ przed zwróceniem przyszłości do kodu wywołującego ❺. Kod, który wysłał komunikat do wątku interfejsu użytkownika, może albo poczekać na przyszłość (jeśli wykonanie zadania jest niezbędne do dalszego działania), albo porzucić tę przyszłość (jeśli nie potrzebuje wyniku przetwarzania).

W tym przykładzie użyliśmy do reprezentacji zadań klasy `std::packaged_task`  $\hookrightarrow$  `<void()>`. Klasa opakowuje funkcję (lub inny obiekt wywołalny), która nie otrzymuje żadnych parametrów i zwraca `void` (jeśli wskazana funkcja zwraca inną wartość, wynik zostanie porzucony). W tym przypadku zastosowano najprostsze możliwe zadanie, jednak (jak już wiemy) szablon klasy `std::packaged_task` może być równie dobrze stosowany w implementacjach bardziej złożonych rozwiązań — wystarczy w roli parametru szablonu użyć innej sygnatury funkcji, zmienić typ zwracanych wartości (a więc także typ danych przechowywanych w ramach stanu przyszłości) i zmienić typy argumentów operatora wywołania funkcji. Przedstawiony przykład można by łatwo rozszerzyć o możliwość przekazywania argumentów do zadań, które mają być wykonywane przez wątek graficznego interfejsu użytkownika, i zwracania wartości w ramach obiektu typu `std::future` (zamiast samego sygnału o zakończeniu zadania).



Co należy zrobić z zadaniami, których nie można wyrazić w formie prostych wywołań funkcji, i zadaniami, których wyniki mogą pochodzić z wielu różnych miejsc? Obsługa takich przypadków wymaga jeszcze innego sposobu tworzenia przyszłości — bezpośredniego ustawiania wartości za pomocą szablonu `std::promise`.

### 4.2.3. Obietnice (szablon `std::promise`)

Programiści pracujący nad aplikacjami, które muszą obsługiwać wiele połączeń sieciowych, często ulegają pokusie obsługi każdego połączenia w osobnym wątku, ponieważ takie rozwiązanie ułatwia zrozumienie i zaimplementowanie mechanizmów komunikacji sieciowej. Takie rozwiązanie sprawdza się w przypadku niewielkiej liczby połączeń (a więc także niewielkiej liczby wątków). Okazuje się jednak, że w razie wzrostu liczby połączeń opisany model staje się nieefektywny, ponieważ duża liczba wątków zajmuje zbyt wiele zasobów systemu operacyjnego, a częste przełączanie kontekstu (jeśli liczba wątków przekracza współbieżność sprzętową) ma negatywny wpływ na wydajność aplikacji. W skrajnych przypadkach aplikacja uruchamiająca dużo nowych wątków może wyczerpać zasoby systemu operacyjnego przed osiągnięciem limitu połączeń sieciowych. Właśnie dlatego nawet w aplikacjach obsługujących bardzo dużo połączeń sieciowych stosuje się stosunkowo niewiele wątków (czasem tylko jeden wątek) odpowiedzialnych za obsługę tych połączeń, zatem każdy wątek musi obsługiwać wiele połączeń jednocześnie.

Przeanalizujmy przykład wątku obsługującego połączenia. Pakiety danych przychodzą za pośrednictwem różnych połączeń w przypadkowej kolejności; podobnie pakiety danych przeznaczone do wysłania są kolejgowane w przypadkowej kolejności. W wielu przypadkach pozostałe elementy aplikacji będą oczekiwały albo na wysłanie danych, albo na otrzymanie nowego pakietu danych za pośrednictwem określonego połączenia sieciowego.

Szablon klasy `std::promise<T>` umożliwia ustawienie wartości (typu `T`), którą w przyszłości będzie można odczytać za pośrednictwem powiązanego obiektu klasy `std::future<T>`. Para klas `std::promise` i `std::future` to jeden z mechanizmów umożliwiających implementację interesującego nas rozwiązania — wątek oczekujący może wstrzymać działanie w oczekiwaniu na przyszłość, natomiast wątek udostępniający dane może użyć obiektu obietnicy do ustawienia powiązanej wartości, tak aby odpowiednia przyszłość przeszła w stan **gotowości**.

Obiekt klasy `std::future` powiązany z danym obiektem klasy `std::promise` można uzyskać za pomocą funkcji składowej `get_future()`, a więc tak samo jak w przypadku obiektu klasy `std::packaged_task`. W momencie ustawienia wartości obiektu obietnicy (za pomocą funkcji składowej `set_value()`) obiekt przyszłości przechodzi w stan **gotowości** i jako taki może zostać użyty do pobrania zapisanej wartości. Jeśli nastąpi zniszczenie obiektu klasy `std::promise` bez wcześniejszego ustawienia wartości, zamiast oczekiwanej wartości zostanie ustawiony stosowny wyjątek. Sposób przekazywania wyjątków pomiędzy wątkami zostanie opisany w punkcie 4.2.4.

Na listingu 4.10 pokazano przykład kodu wątku, który przetwarza połączenia w opisany powyżej sposób. W prezentowanym przykładzie użyliśmy pary klas `std::promise` i `std::future<bool>` do identyfikacji udanej transmisji bloku danych wychodzących; wartość powiązana z obiektem przyszłości ma postać prostej flagi sukcesu

lub niepowodzenia. W przypadku pakietów przychodzących funkcję danych powiązanych z obiektem przyszłości pełni właściwa treść tych pakietów.

**Listing 4.10. Obsługa wielu połączeń w jednym wątku przy użyciu obiektów obietnic**

```
#include <future>

void process_connections(connection_set& connections)
{
    while(!done(connections)) ← ❶
    {
        for(connection_iterator ← ❷
            connection=connections.begin(),end=connections.end();
            connection!=end;
            ++connection)
        {
            if(connection->has_incoming_data()) ← ❸
            {
                data_packet data=connection->incoming();
                std::promise<payload_type>& p=
                    connection->get_promise(data.id); ← ❹
                p.set_value(data.payload);
            }
            if(connection->has_outgoing_data()) ← ❺
            {
                outgoing_packet data=
                    connection->top_of_outgoing_queue();
                connection->send(data.payload);
                data.promise.set_value(true); ← ❻
            }
        }
    }
}
```

Funkcja `process_connections()` wykonuje pętlę do momentu, w którym funkcja `done()` zwróci wartość `true` ❶. W każdej iteracji tej pętli kod aplikacji sprawdza kolejno każde połączenie ❷ i pobiera dane przychodzące (jeśli istnieją) ❸ lub wysyła kolejkwane dane wychodzące ❺. Zakładamy, że pakiet przychodzący zawiera jakiś identyfikator i właściwe dane. Identyfikator jest odwzorowywany na odpowiedni obiekt klasy `std::promise` (na przykład metodą odnajdywania w kontenerze asocjacyjnym) ❹, natomiast wartość jest przypisywana do ciała pakietu. W przypadku pakietów wychodzących zastosowano mechanizm kolejki pakietów oczekujących na wysłanie — program sprawdza stan kolejki i wysyła ewentualne pakiety dla danego połączenia. Po wysłaniu pakietu w obiekcie obietnicy powiązanej z tymi danymi wychodzącymi jest ustawiana wartość `true`, która oznacza pomyślną transmisję danych ❻. Zgodność opisanego modelu z rzeczywistymi protokołami komunikacji sieciowej zależy tylko od tych protokołów. Struktura na bazie obietnicy i przyszłości nie pasuje co prawda do każdego scenariusza, ale pod wieloma względami przypomina model asynchronicznych operacji wejścia-wyjścia stosowany w niektórych systemach operacyjnych.

W dotychczas prezentowanym kodzie całkowicie ignorowaliśmy problem wyjątków. Wyobrażenie świata, w którym wszystko działa, jak należy, jest być może kuszące, ale nie ma wiele wspólnego z rzeczywistością. Nie można wykluczyć, że dysk zostanie

zapełniony, że program nie będzie mógł znaleźć potrzebnych danych, że nastąpi awaria połączenia sieciowego lub że w wyniku błędu nie będzie dostępna baza danych. Jeśli operacja wykonywana w jednym wątku potrzebuje do działania wyniku innego wątku, warto uwzględnić możliwość zasygnalizowania błędu w formie wyjątku — zakładanie, że w kodzie stosującym obiekty klasy `std::packaged_task` czy `std::promise` wszystko zawsze będzie działało prawidłowo, byłoby zbyt optymistyczne. Biblioteka standardowa języka C++ oferuje wygodne mechanizmy obsługi wyjątków w tego rodzaju scenariuszach i umożliwia zapisywanie wyjątków w ramach wyników powiązanych z tymi obiektami.

#### 4.2.4. Zapisywanie wyjątku na potrzeby przyszłości

Przeanalizujmy następujący fragment kodu. Jeśli na wejściu funkcji `square_root()` prześlemy wartość `-1`, zgłoszony zostanie wyjątek (to on trafi do kodu wywołującego tę funkcję):

```
double square_root(double x)
{
    if(x<0)
    {
        throw std::out_of_range("x<0");
    }
    return sqrt(x);
}
```

Przypuśćmy teraz, że zamiast wywołać funkcję `square_root()` w bieżącym wątku, jak w poniższym wierszu:

```
double y=square_root(-1);
```

użyjemy wywołania asynchronicznego w następującej formie:

```
std::future<double> f=std::async(square_root,-1);
double y=f.get();
```

Idealnym rozwiązaniem byłoby zapewnienie dokładnie takiego samego zachowania jak w przypadku kodu jednowątkowego — skoro zmiennej `y` w obu przypadkach jest przypisywany wynik funkcji, wątek wywołujący funkcję `f.get()` powinien mieć dostęp także do ewentualnych wyjątków (tak jak odpowiedni kod jednowątkowy).

Okazuje się, że właśnie tak działa prezentowane rozwiązanie: jeśli funkcja `square_root` wywołana za pośrednictwem funkcji `std::async` zgłosi jakiś wyjątek, wyjątek ten zostanie zapisany w obiekcie przyszłości (w miejscu dla wartości wynikowej), przyszłość przejdzie w stan **gotowości**, a funkcja `get()` spowoduje ponowne zgłoszenie zapisanego wyjątku. (Uwaga: standard języka C++ nie określa, czy ponowne zgłoszenie dotyczy oryginalnego obiektu wyjątku, czy jego kopii; różne kompilatory i biblioteki stosują w tym względzie odmienne rozwiązania). To samo dotyczy funkcji opakowanej w ramach obiektu klasy `std::packaged_task` — jeśli po wywołaniu zadania opakowana funkcja zgłosi jakiś wyjątek, wyjątek jest zapisywany w obiekcie przyszłości zamiast właściwego wyniku. Aby ponownie zgłosić ten wyjątek, wystarczy wywołać funkcję `get()`.

Szablon klasy `std::promise` oferuje oczywiście analogiczne rozwiązanie, które wymaga bezpośredniego wywołania funkcji. Aby zapisać wyjątek zamiast wartości wynikowej, wystarczy wywołać funkcję składową `set_exception()` zamiast funkcji `set_value()`.

Wspomniana funkcja jest zwykle stosowana w bloku `catch` odpowiedzialnym za obsługę wyjątku zgłoszonego w trakcie działania algorytmu — wyjątek jest umieszczany w obiekcie obietnicy:

```
extern std::promise<double> some_promise;

try
{
    some_promise.set_value(calculate_value());
}
catch(...)
{
    some_promise.set_exception(std::current_exception());
}
```

W powyższym kodzie użyto funkcji `std::current_exception()` do pobrania zgłoszonego wyjątku; alternatywnym rozwiązaniem byłoby wywołanie funkcji `std::copy_exception()` w celu zapisania nowego wyjątku bez jego bezpośredniego zgłaszania:

```
some_promise.set_exception(std::copy_exception(std::logic_error("foo ")));
```

Opisane rozwiązanie jest nieporównanie bardziej czytelne niż stosowanie bloku `try-catch`, jeśli tylko potencjalny wyjątek jest znany z wyprzedzeniem. W ten sposób można nie tylko uprościć kod, ale też ułatwić optymalizację tego kodu przez kompilator.

Innym sposobem zapisywania wyjątku w przyszłości jest zniszczenie obiektu klasy `std::promise` lub `std::packaged_task` powiązanego z obiektem przyszłości bez uprzedniego wywołania funkcji ustawiającej (w przypadku obiektu obietnicy) lub uruchomienia opakowanego zadania. Jeśli obiekt przyszłości nie będzie **gotowy**, w obu przypadkach destruktor klasy `std::promise` lub `std::packaged_task` zapisze w powiązanim stanie wyjątek typu `std::future_error` z kodem błędu `std::future_errc::broken_promise`. Tworząc przyszłość, zapowiadamy (składamy obietnicę), że udostępnimy jakąś wartość lub jakiś wyjątek; zniszczenie źródła tej wartości lub tego wyjątku bez uprzedniego dostarczenia zapowiedzianego zasobu łamie tę obietnicę. Gdyby w opisanym przypadku kompilator niczego nie zapisał w obiekcie przyszłości, wątki oczekujące mogłyby czekać w nieskończoność.

Do tej pory we wszystkich przykładach stosowałem szablon klasy `std::future`. Warto jednak pamiętać o pewnych ograniczeniach szablonu `std::future`, w tym o możliwości oczekiwania na wynik przez zaledwie jeden wątek. W razie konieczności zaimplementowania modelu, w którym na jedno zdarzenie będzie oczekiwało wiele wątków, należy użyć raczej szablonu klasy `std::shared_future`.

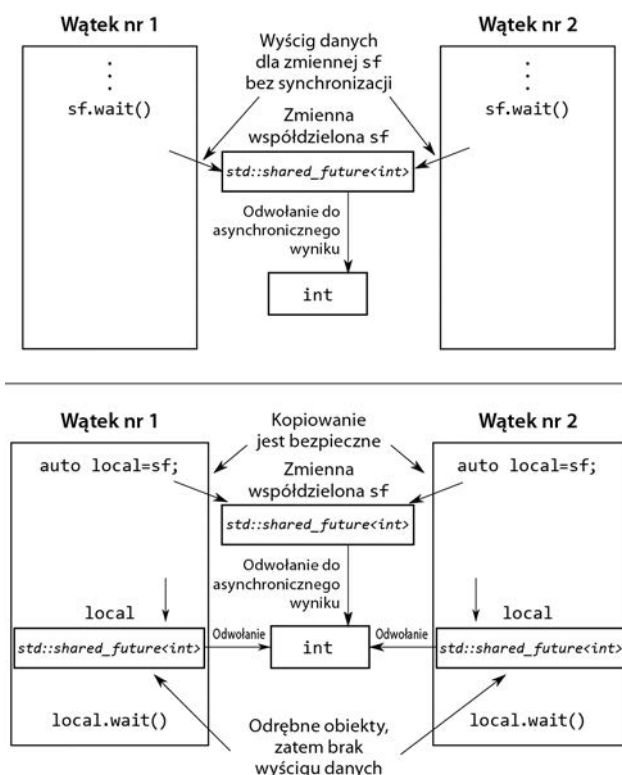
#### 4.2.5. Oczekiwanie na wiele wątków

Mimo że szablon klasy `std::future` obsługuje wszystkie mechanizmy synchronizacji potrzebne do przesyłania danych pomiędzy wątkami, wywołania funkcji składowych określonego obiektu klasy `std::future` nie są synchronizowane z wywołaniami funkcji pozostałych obiektów tej klasy. Jeśli wiele wątków uzyskuje dostęp do jednego obiektu klasy `std::future` bez stosowania dodatkowych mechanizmów synchronizacji, aplikacja jest narażona na **wyścig danych** i niezdefiniowane zachowania. Problem wynika z projektu tego rozwiązania — szablon klasy `std::future` modeluje unikatową własność asynchronicznego wyniku, a jednorazowy charakter funkcji `get()` i tak wyklucza sens

współbieżnego dostępu. Skoro po pierwszym wywołaniu funkcji `get()` nie można już pobrać żadnych danych, z natury rzeczy dane mogą być pobrane tylko przez jeden wątek.

Jeśli jednak projekt naszej aplikacji współbieżnej wymaga, aby wiele wątków mogło czekać na to samo zdarzenie, nie wszystko stracone — wystarczy użyć szablonu klasy `std::shared_future`. O ile szablon klasy `std::future` oferuje tylko możliwość **przeniesienia**, zatem własność przyszłości można przenosić pomiędzy różnymi obiektami, ale tylko jeden obiekt może się jednocześnie odwoływać do jednego asynchronicznego wyniku, o tyle szablon klasy `std::shared_future` oferuje możliwość **kopiowania**, zatem może istnieć wiele obiektów odwołujących się do tego samego stanu.

W przypadku szablonu `std::shared_future` funkcje składowe wywoływane dla pojedynczego obiektu wciąż nie są synchronizowane, zatem warunkiem unikania wyścigów danych w związku z dostępem do tego samego obiektu z poziomu wielu wątków jest ochrona tego dostępu za pomocą blokady. Najlepszym sposobem jest kopiowanie tego obiektu, tak aby każdy wątek uzyskiwał dostęp do własnej kopii. Dostęp do współdzielonego, asynchronicznego stanu z poziomu wielu wątków jest bezpieczny, jeśli tylko każdy z tych wątków uzyskuje dostęp do stanu za pośrednictwem własnego obiektu klasy `std::shared_future`. Przykład takiego rozwiązania pokazano na rysunku 4.1.



**Rysunek 4.1. Użycie wielu obiektów klasy `std::shared_future` w celu uniknięcia wyścigów danych**

Jednym z możliwych zastosowań szablonu klasy `std::shared_future` jest implementacja równoległego wykonywania jakiejś operacji w modelu zbliżonym do złożonego arkusza kalkulacyjnego, gdzie każda komórka zawiera wartość, która może być używana

we wzorach w wielu pozostałych komórkach. Wzory potrzebne do obliczania wyników w komórkach zależnych mogą używać obiektu klasy `std::shared_future` podczas odwoływania się do pierwszej komórki. Jeśli wzory we wszystkich komórkach będą przetwarzane równolegle, zadania odwołujące się do gotowych wartości zostaną zrealizowane natychmiast, natomiast zadania zależne od innych, jeszcze przetwarzanych komórek będą musiały poczekać na osiągnięcie gotowości przez tamte komórki. Takie rozwiązanie umożliwia maksymalne wykorzystanie dostępnej współbieżności sprzętowej.

Obiekty klasy `std::shared_future`, które wskazują pewien asynchroniczny stan, są konstruowane na podstawie obiektów klasy `std::future` odwołujących się do tego stanu. Ponieważ obiekty klasy `std::future` nie współdzielą własności tego asynchronicznego stanu z żadnymi innymi obiektami, własność należy przenieść do obiektu klasy `std::shared_future` za pomocą funkcji `std::move`, pozostawiając oryginalny obiekt klasy `std::future` z pustym stanem (jak w przypadku użycia konstruktora domyślnego):

```
std::promise<int> p;
std::future<int> f(p.get_future());
assert(f.valid());           ← ❶ Przyszłość f jest prawidłowa
std::shared_future<int> sf(std::move(f));
assert(!f.valid());         ← ❷ Przyszłość f już nie jest prawidłowa
assert(sf.valid());         ← ❸ Przyszłość sf jest teraz prawidłowa
```

Obiekt przyszłości `f` jest początkowo prawidłowy ❶, ponieważ odwołuje się do asynchronicznego stanu obietnicy `p`, jednak po przeniesieniu tego stanu do obiektu `sf` to obiekt `sf` jest prawidłowy ❸, natomiast obiekt `f` jest już nieprawidłowy ❷.

Jak w przypadku wszystkich obiektów z możliwością przenoszenia, przeniesienie własności jest wykonywane automatycznie dla `r`-wartości, zatem możemy skonstruować obiekt klasy `std::shared_future` bezpośrednio na podstawie wartości zwróconej przez funkcję składową `get_future()` obiektu klasy `std::promise`:

```
std::promise<std::string> p;
std::shared_future<std::string> sf(p.get_future()); ← Automatyczne
                                                    ❶ przeniesienie własności
```

W powyższym kodzie własność jest przenoszona automatycznie — obiekt klasy `std::shared_future<string>` jest konstruowany na podstawie `r`-wartości typu `std::future<std::string>` ❶.

Szablon klasy `std::future` oferuje jeszcze inne rozwiązanie ułatwiające stosowanie obiektów klasy `std::shared_future` przy użyciu nowego mechanizmu automatycznego określania typu zmiennej na podstawie inicjalizatora (patrz część A.6 dodatku A). Szablon klasy `std::future` definiuje funkcję składową `share()`, która tworzy nowy obiekt klasy `std::shared_future` i bezpośrednio przenosi własność do tego obiektu. Użycie tego rozwiązania może nam oszczędzić sporo pisania i znacznie ułatwia modyfikowanie kodu:

```
std::promise< std::map< SomeIndexType, SomeDataType, SomeComparator,
SomeAllocator>::iterator> p;
auto sf=p.get_future().share();
```

W tym przypadku typ zmiennej `sf` jest identyfikowany jako `std::shared_future<std::map< SomeIndexType, SomeDataType, SomeComparator, SomeAllocator>::iterator>`, czyli konstrukcja, której wielokrotne stosowanie w kodzie byłoby dość kłopotliwe.

W razie zmiany komparatora lub alokatora wystarczy zmodyfikować typ obiektu obietnicy; typ obiektu przyszłości zostanie automatycznie zaktualizowany i dostosowany do tej zmiany.

W pewnych przypadkach dobrym rozwiązaniem jest ograniczanie maksymalnego czasu oczekiwania na zdarzenie (z uwagi na ograniczony czas działania określonej sekcji kodu lub ze względu na istnienie innych ważnych zadań, którymi dany wątek może się zająć, jeśli oczekiwane zdarzenie nie wystąpi odpowiednio wcześniej). Z myślą o takich przypadkach wiele funkcji oczekiwania oferuje wersje z możliwością określenia limitu czasowego.

### 4.3. Oczekiwanie z limitem czasowym

Wszystkie wywołania blokujące, które stosowaliśmy w dotychczasowych przykładach, blokowały wykonywanie wątków przez nieokreślony czas, tj. do momentu wystąpienia oczekiwanego zdarzenia. W wielu przypadkach takie rozwiązanie jest wystarczające, jednak w niektórych sytuacjach lepszym wyjściem jest określenie maksymalnego czasu oczekiwania. Stosowanie takich limitów czasowych może mieć na celu potwierdzenie prawidłowego działania aplikacji (w formie komunikatu dla użytkownika lub innego procesu) lub przerwanie oczekiwania, jeśli na przykład użytkownik kliknął przycisk *Anuluj*.

Istnieją dwa rodzaje limitów czasowych stosowanych dla operacji blokujących: limity określające maksymalny **czas blokowania** wątku (na przykład 30 milisekund) oraz limity **bezwzględne**, gdzie oczekiwanie nie może trwać dłużej niż do określonego punktu w czasie (na przykład do godziny 17:30:15.045987023 dnia 30 listopada 2012 roku). Większość funkcji oczekujących występuje w wersjach obsługujących obie formy limitów czasowych. Wersje obsługujące względne limity czasowe (określające czas trwania operacji) są oznaczane przyrostkiem `_for`, natomiast bezwzględne limity czasowe oznaczają się przyrostkiem `_until`.

Na przykład klasa `std::condition_variable` definiuje dwie przeciążone wersje funkcji składowej `wait_for()` i dwie przeciążone wersje funkcji składowej `wait_until()`, czyli odpowiedniki obu wersji funkcji `wait()` uzupełnione o obsługę względnych i bezwzględnych limitów czasowych — pierwsza wersja czeka na sygnał, upłynięcie limitu czasowego lub bezpośrednio budzenie; druga wersja w momencie budzenia sprawdza przekazany predykat i zwraca sterowanie, pod warunkiem że albo ten predykat jest prawdziwy (w wyniku sygnału umieszczonego w zmiennej warunkowej), albo osiągnięto limit czasowy.

Zanim przeanalizujemy szczegółowe aspekty stosowania funkcji uwzględniających limity czasowe, warto poświęcić chwilę na omówienie sposobu określania czasu w języku C++, w tym dostępnych zegarów.

#### 4.3.1. Zegary

W kontekście elementów biblioteki standardowej języka C++ zegar jest dla programu źródłem informacji o bieżącej godzinie. W szczególności zegar jest klasą udostępniającą cztery odrębne informacje:

- **bieżąca** godzina;
- typ wartości używanych do reprezentowania godzin uzyskiwanych z obiektu zegara;
- okres reprezentowany przez jeden takt zegara;
- to, czy takty zegara mają stałą długość, czyli możliwość traktowania zegara jako **stabilnego**.

Bieżącą godzinę reprezentowaną przez zegar można uzyskać, wywołując statyczną funkcję składową `now()` klasy zegara; na przykład funkcja `std::chrono::system_clock::now()` zwróci bieżącą godzinę reprezentowaną przez zegar systemowy. Typ punktów w czasie dla poszczególnych zegarów jest reprezentowany przez składową definicję typu `time_point`, zatem każda funkcja `zegar::now()` zwraca wartość typu `zegar::time_point`.

Okres taktu zegara jest wyrażany w formie ułamka sekundy reprezentowanego przez składową definicję typu `period` — w przypadku zegara wykonującego 25 taktów w ciągu sekundy `period` definiuje typ `std::ratio<1,25>`, natomiast w przypadku zegara wykonującego jeden takt na 2,5 sekundy składowa `period` definiuje typ `std::ratio<5,2>`. Jeśli określenie okresu taktu nie jest możliwe do momentu uruchomienia programu lub jeśli ten okres może się zmieniać w czasie działania aplikacji, okres można zdefiniować w formie średniego czasu trwania taktu, najkrótszego możliwego taktu lub innej wartości przewidzianej przez twórców biblioteki. Nie można jednak zakładać, że obserwowany okres taktu zegara podczas jednej próby uruchomienia programu będzie odpowiadał rzeczywistemu okresowi danego zegara.

Jeśli **takty zegara mają stałą częstotliwość** (niezależnie od tego, czy ta częstotliwość pasuje do przyjętego okresu) i jeśli **nie możemy zmienić długości taktu**, mamy do czynienia z tzw. **stabilnym zegarem** (ang. *steady clock*). Składowa statyczna `is_steady` klasy stabilnego zegara ma wartość `true` (w przypadku niestabilnego zegara ta sama składowa ma wartość `false`). Zegar systemowy (reprezentowany przez klasę `std::chrono::system_clock`) zwykle **nie** jest stabilny, ponieważ można dostosowywać jego częstotliwość (nawet jeśli zmiany częstotliwości są wprowadzane automatycznie z uwzględnieniem przesunięć względem zegara lokalnego). Każda taka zmiana może spowodować, że wywołanie funkcji `now()` zwróci wartość wcześniejszą niż zwrócona przez poprzednie wywołanie tej funkcji, co oczywiście narusza wymaganie stałej częstotliwości zegara (i długości taktu). Jak się za chwilę przekonasz, stabilne zegary są szczególnie przydatne podczas obliczeń z uwzględnieniem limitów czasowych — z myślą o tych i podobnych zastosowaniach twórcy biblioteki standardowej udostępnili taki zegar w formie klasy `std::chrono::steady_clock`. Biblioteka standardowa języka C++ zawiera też inne klasy zegarów: wspomnianą wcześniej klasę `std::chrono::system_clock`, która reprezentuje zegar „czasu rzeczywistego” w danym systemie i która udostępnia funkcję konwersji punktów w czasie na i z wartości typu `time_t`, oraz klasę `std::chrono::high_resolution_clock`, która zapewnia najkrótszy możliwy takt zegara (a więc także najwyższą możliwą częstotliwość) spośród wszystkich zegarów tej biblioteki. Drugi z zegarów można wykorzystać w roli punktu wyjścia dla definicji alternatywnych rozwiązań. Wymienione zegary zdefiniowano (wraz z pozostałymi elementami związanymi z obsługą czasu) w pliku nagłówkowym `<chrono>`.

Zanim przystąpimy do omawiania metod reprezentowania punktów w czasie, warto poświęcić chwilę analizie technik reprezentowania okresów.



### 4.3.2. Okresy

Okres (czas trwania) to najprostszy aspekt obsługi czasu. Okres zaimplementowano w szablonie klasy `std::chrono::duration<>` (wszystkie elementy języka C++ związane z obsługą czasu i używane przez bibliotekę wątków należą do przestrzeni nazw `std::` ↪ `chrono`). Pierwszy parametr tego szablonu określa typ reprezentacji (na przykład `int`, `long` lub `double`); drugi parametr jest ułamkiem określającym liczbę sekund reprezentowanych przez jednostkę okresu. Na przykład liczba minut przechowywana w wartości typu `short` jest reprezentowana przez klasę `std::chrono::duration<short, std::ratio<60, 1>>`, ponieważ minuta składa się z 60 sekund. Liczba milisekund przechowywanych w wartości typu `double` jest reprezentowana przez klasę `std::chrono::duration<double, std::ratio<1, 1000>>`, ponieważ każda milisekunda trwa jedną tysięczną część sekundy.

Biblioteka standardowa oferuje zbiór predefiniowanych definicji typów w przestrzeni nazw `std::chrono` dla różnych okresów (wyrażanych w nanosekundach, mikrosekundach, milisekundach, sekundach, minutach i godzinach). Wszystkie te definicje stosują na tyle elastyczne typy całkowitoliczbowe, że mogą reprezentować na przykład okresy ponad 500-letnie w wybranych jednostkach czasu. Przestrzeń nazw zawiera także definicje typów dla rzędów wielkości układu SI: od `std::atto` ( $10^{-18}$ ) do `std::exa` ( $10^{18}$ ) (i więcej, jeśli tylko dana platforma obsługuje 128-bitowe typy całkowitoliczbowe). Za pomocą tych typów można operować na niestandardowych okresach, na przykład klasa `std::duration<double, std::centi>` obsługuje liczbę setnych części sekundy reprezentowanych przez liczbę typu `double`.

Konwersja pomiędzy okresami jest wykonywana automatycznie, pod warunkiem że nie wymaga obciążenia wartości źródłowej — oznacza to, że konwersja godzin na sekundy jest możliwa, ale już konwersja sekund na godziny nie zostanie wykonana automatycznie. Konwersję można też wykonać jawnie za pomocą funkcji `std::chrono::duration_cast<>`:

```
std::chrono::milliseconds ms(54802);
std::chrono::seconds s=
    std::chrono::duration_cast<std::chrono::seconds>(ms);
```

Ponieważ wynik jest obcinany (nie zaokrąglany), zmienna `s` będzie zawierała wartość 54.

Okresy obsługują działania arytmetyczne, zatem możemy dodawać i odejmować okresy, aby otrzymywać nowe okresy, bądź mnożyć lub dzielić okresy przez stałe wybranego typu danych (czyli pierwszego parametru szablonu klasy). Oznacza to, że wyrażenie `5*seconds(1)` ma taką samą wartość jak wyrażenia `seconds(5)` i `minutes(1) - seconds(55)`. Liczbę jednostek składających się na dany okres można uzyskać za pomocą funkcji składowej `count()`. Oznacza to, że wywołanie `std::chrono::milliseconds(1234).count()` zwróci wartość 1234.

Wymuszanie oczekiwania na podstawie okresu (czasu trwania) wymaga stosowania instancji typu `std::chrono::duration<>`. Możemy na przykład spowodować, że czas oczekiwania na gotowość obiektu przyszłości wyniesie 35 milisekund:

```
std::future<int> f=std::async(some_task);
if(f.wait_for(std::chrono::milliseconds(35))==std::future_status::ready)
do_something_with(f.get());
```

Wszystkie funkcje oczekiwania zwracają status określający, czy koniec oczekiwania wynika z wyczerpania limitu czasowego, czy z wystąpienia zdarzenia, na które czekał dany wątek. W tym przypadku wątek czeka na przyszłość, zatem funkcja zwraca wartość `std::future_status::timeout` w przypadku wyczerpania limitu czasowego; wartość `std::future_status::ready`, jeśli przyszłość jest gotowa; lub wartość `std::future_status::deferred`, jeśli zadanie przyszłości zostało odłożone na później. Czas oczekiwania okresowego jest mierzony przy użyciu stabilnego, wewnętrznego zegara biblioteki, zatem 35 milisekund oznacza właśnie 35 milisekund, nawet jeśli w czasie oczekiwania zegar systemowy zostanie przestawiony (w przód lub w tył). Nie można oczywiście zapominąć o kapryśkach systemu szeregowania zadań i o zróżnicowanej precyzji zegarów systemów operacyjnych, które mogą spowodować, że rzeczywisty czas dzielący wywołanie funkcji `wait()` od zwrócenia sterowania będzie dużo dłuższy niż 35 ms.

Skoro potrafimy już stosować okresy, możemy przejść do analizy modelu punktów w czasie.

### 4.3.3. Punkty w czasie

Punkt w czasie jest reprezentowany przez instancję szablonu klasy `std::chrono::time_point<>`. Pierwszy parametr tego szablonu wskazuje zegar, natomiast drugi parametr określa jednostki miary (w tej roli należy użyć specjalizacji szablonu klasy `std::chrono::duration<>`). Wartość punktu w czasie reprezentuje czas (w formie wielokrotności wskazanego okresu) od konkretnego punktu w czasie nazywanego **epoką** zegara. Epoka zegara jest prostą właściwością, która jednak nie jest bezpośrednio dostępna ani wprost definiowana przez standard języka C++. Do najczęściej stosowanych epok należy północ 1 stycznia 1970 roku i moment uruchomienia komputera, na którym działa dana aplikacja. Zegary mogą stosować jedną epokę lub różne, niezależne epoki. Jeśli dwa zegary stosują tę samą epokę, definicja typu `time_point` w klasie jednego zegara może wskazywać drugą klasę jako typ zegara powiązanego z daną definicją `time_point`. Mimo że nie można bezpośrednio uzyskać epoki, **mamy do dyspozycji** funkcję `time_point::since_epoch()`, którą możemy wywołać dla danej instancji typu `time_point`. Funkcja składowa `time_point::since_epoch()` zwraca wartość okresu reprezentującą czas od epoki zegara do określonego punktu w czasie.

Punkt w czasie można zdefiniować na przykład w formie obiektu klasy `std::chrono::time_point<std::chrono::system_clock, std::chrono::minutes>`. Tak skonstruowany obiekt zawiera czas względem zegara systemowego, tyle że mierzony w minutach (nie według standardowej precyzji zegara systemowego, czyli sekund lub części sekundy).

Na obiektach klasy `std::chrono::time_point<>` można wykonywać operacje dodawania i odejmowania okresów, których wynikiem są nowe punkty w czasie. Oznacza to, że na przykład w wyniku dodawania `std::chrono::high_resolution_clock::now() + std::chrono::nanoseconds(500)` otrzymamy punkt w czasie, który nastąpi za 500 nanosekund (licząc od teraz). Wyrażenia tego typu są przydatne podczas obliczania bezwzględnego limitu czasowego w sytuacji, gdy maksymalny czas wykonywania bloku kodu jest znany z wyprzedzeniem. Takie rozwiązanie wymaga jednak wielu wywołań funkcji oczekujących lub funkcji poprzedzających funkcję oczekującą i zaliczanych do bloku, który podlega ograniczeniu czasowemu.

Istnieje też możliwość odjęcia jednego punktu w czasie od innego, pod warunkiem że oba punkty bazują na tym samym zegarze. Wynikiem tej operacji jest okres, który reprezentuje czas dzielący oba punkty. W ten sposób można na przykład sprawdzać czas wykonywania bloków kodu:

```
auto start=std::chrono::high_resolution_clock::now();
do_something();
auto stop=std::chrono::high_resolution_clock::now();
std::cout<<"Wykonanie funkcji do_something() zajęło "  
    <<std::chrono::duration<double,std::chrono::seconds>(stop-start).count()  
    <<" sekund."<<std::endl;
```

Parametr zegara szablonu klasy `std::chrono::time_point<>` decyduje nie tylko o epoce. W przypadku przekazania punktu w czasie na wejściu funkcji oczekującej, która stosuje bezwzględny limit czasowy, wybrany zegar będzie używany do mierzenia czasu. Warto pamiętać o możliwości zmiany wskazań zegara i o tym, że funkcja oczekująca nie zwróci sterowania do momentu, w którym funkcja `now()` tego zegara nie zwróci wartości późniejszej od wskazanego limitu czasowego. Jeśli zegar zostanie przestawiony w przód, łączny czas oczekiwania może być krótszy (w porównaniu z czasem mierzonym przez stabilny zegar); a jeśli zegar zostanie cofnięty, łączny czas oczekiwania zostanie wydłużony.

Jak nietrudno odgadnąć, punkty w czasie są używane w wersjach funkcji oczekujących z przyrostkiem `_until`. Typowym zastosowaniem tego rozwiązania jest wyznaczenie przesunięcia względem godziny zwracanej przez wywołanie `jakiś-zegar::now()` w wybranym punkcie programu. Punkty powiązane z zegarem systemowym można uzyskiwać, konwertując instancje typu `time_t` za pomocą statycznej funkcji składowej `std::chrono::system_clock::to_time_point()`. Jeśli na przykład maksymalny czas oczekiwania na zdarzenie powiązane ze zmienną warunkową wynosi 500 milisekund, można zastosować konstrukcję podobną do tej z listingu 4.11.

#### Listing 4.11. Oczekiwanie na zmienną warunkową z uwzględnieniem limitu czasowego

```
#include <condition_variable>
#include <mutex>
#include <chrono>

std::condition_variable cv;
bool done;
std::mutex m;

bool wait_loop()
{
    auto const timeout= std::chrono::steady_clock::now()+
        std::chrono::milliseconds(500);
    std::unique_lock<std::mutex> lk(m);
    while(!done)
    {
        if(cv.wait_until(lk,timeout)==std::cv_status::timeout)
            break;
    }
    return done;
}
```

Rozwiązanie z listingu 4.11 jest zalecanym sposobem oczekiwania na zmienne warunkowe z uwzględnieniem limitu czasowego w sytuacji, gdy funkcja oczekująca nie otrzymuje na wejściu żadnego predykatu. Maksymalny czas wykonywania pętli jest ograniczony. Jak napisałem w punkcie 4.1.1, jeśli nie stosujemy dodatkowego predykatu, operowanie na zmiennych warunkowych wymaga użycia pętli, która obsługuje nietypowe sposoby budzenia wątku. W przypadku wywołania funkcji `wait_for()` w ciele pętli może się zdarzyć, że warunek wznowienia działania zostanie spełniony bezpośrednio przed upływem limitu czasowego, a w następnej iteracji czas oczekiwania będzie liczony od początku. Taka sytuacja może mieć miejsce wielokrotnie, zatem łączny czas oczekiwania byłby nieograniczony.

Skoro dysponujemy już podstawowymi narzędziami umożliwiającymi stosowanie limitów czasowych, pora omówić funkcje, w których można używać tych limitów.

#### 4.3.4. Funkcje otrzymujące limity czasowe

Najprostszym zastosowaniem limitu czasowego jest dodanie opóźnienia do przetwarzania określonego wątku, tak aby ten wątek nie zajmował czasu procesora (potrzebnego innym wątkom) w czasie, gdy nie wykonuje żadnych wartościowych zadań. Przykład takiego rozwiązania opisałem w podrozdziale 4.1, gdzie kod wielokrotnie sprawdzał stan flagi gotowości w pętli. Do opóźniania działania (usypiania) wątków służą funkcje `std::this_thread::sleep_for()` i `std::this_thread::sleep_until()`. Obie funkcje działają jak proste budziki — wątek jest usypiany albo na określony okres (za pomocą funkcji `sleep_for()`), albo do wskazanego punktu w czasie (za pomocą funkcji `sleep_until()`). W przykładowych rozwiązaniach z podrozdziału 4.1 należałoby użyć funkcji `sleep_for()`, ponieważ w przypadku okresowo wykonywanych operacji wątek powinien być wstrzymywany na pewien czas (nie do określonego punktu w czasie). Funkcja `sleep_until()` umożliwia planowanie budzenia wątku w określonym punkcie w czasie. Funkcja `sleep_until()` może więc służyć na przykład do uruchamiania procedury tworzenia kopii zapasowej o północy, drukowania listy płac o 6 rano lub wstrzymywania wątku do momentu wyświetlenia następnej klatki podczas odtwarzania zapisu wideo.

Usypianie wątku to oczywiście nie jedyne zastosowanie limitów czasowych — jak już wspomniałem, limity tego typu można stosować łącznie ze zmiennymi warunkowymi i przyszłościami. Istnieje nawet możliwość stosowania limitów czasowych podczas prób uzyskania blokady muteksu, jeśli tylko użyty muteks obsługuje takie działanie. Standardowe muteksy typów `std::mutex` i `std::recursive_mutex` nie obsługują limitów czasowych dla operacji blokowania, ale już muteksy typów `std::timed_mutex` i `std::recursive_timed_mutex` dopuszczają takie działanie. Oba te typy udostępniają funkcje składowe `try_lock_for()` i `try_lock_until()`, które próbują uzyskać blokadę muteksu odpowiednio w określonym czasie lub przed osiągnięciem określonego punktu w czasie. W tabeli 4.1 opisano funkcje biblioteki standardowej języka C++, które obsługują limity czasowe, wraz z ich parametrami i typami zwracanych wartości. W miejsce parametru opisanego jako *okres* należy przekazać obiekt klasy `std::duration<>`, natomiast parametr *punkt\_w\_czasie* należy zastąpić obiektem klasy `std::time_point<>`.

Skoro wiemy już, jak działają zmienne warunkowe, obiekty przyszłości i obietnic oraz opakowane zadania, czas przeanalizować te mechanizmy w nieco szerszym kontekście, w szczególności przyjrzeć się technikom upraszczania (za pomocą tych mechanizmów) synchronizacji operacji wykonywanych przez różne wątki.

Tabela 4.1. Funkcje otrzymujące limity czasowe

Klasa lub przestrzeń nazw	Funkcje	Zwracane wartości
Przeźren nazw std::this_thread	sleep_for(okres) sleep_until(punkt_w_czasie)	Nie dotyczy
std::condition_variable lub std::condition_variable_any	wait_for(blokada, okres) wait_until(blokada, punkt_w_czasie) wait_for(blokada, okres, predykat) wait_until(blokada, punkt_w_czasie, predykat)	std::cv_status::timeout lub std::cv_status::no_timeout  bool — wartość zwrócona przez predykat po obudzeniu wątku.
std::timed_mutex lub std::recursive_timed_mutex	try_lock_for(okres) try_lock_until ↳(punkt_w_czasie)	bool — wartość true, jeśli udało się uzyskać blokadę; w przeciwnym razie wartość false
std::unique_lock<Typ ↳ZmożliwościąBlokady ↳Czasowej>	unique_lock ↳(typ_blokowalny, okres) unique_lock(typ_blokowalny, punkt_w_czasie)  try_lock_for(okres) try_lock_until ↳(punkt_w_czasie)	Nie dotyczy — funkcja owns_lock() wywołana dla nowo skonstruowanego obiektu zwraca wartość true, jeśli udało się uzyskać blokadę (w przeciwnym razie zwraca wartość false).  bool — wartość true, jeśli udało się uzyskać blokadę; w przeciwnym razie wartość false
std::future<TypWartości> lub std::shared_future< ↳TypWartości>	wait_for(okres) wait_until(punkt_w_czasie)	Jeśli wyczerpano limit czasu funkcji oczekującej, zwraca wartość std::future_status::timeout; jeśli obiekt przyszłości jest gotowy, zwraca wartość std::future_status::ready; jeśli przyszłość zawiera odroczoną funkcję, która nie została jeszcze wywołana, zwraca wartość std::future_status::deferred.

#### 4.4. Upraszczenie kodu za pomocą technik synchronizowania operacji

Stosowanie mechanizmów synchronizacji, które opisałem w poprzednich podrozdziałach, w roli gotowych elementów składowych umożliwia programiście koncentrowanie się na samych operacjach wymagających synchronizacji, nie na mechanice tej synchronizacji. Mechanizmy synchronizacji pozwalają uprościć kod aplikacji choćby dlatego, że wprowadzają do świata programowania współbieżnego dużo więcej elementów znanych z **programowania funkcyjnego**. Zamiast bezpośrednio współdzielić dane pomiędzy wątkami, każde zadanie może otrzymywać potrzebne dane, a wynik przetwarzania może być przekazywany do wielu innych wątków za pośrednictwem obiektów przyszłości.

#### 4.4.1. Programowanie funkcyjne przy użyciu przyszłości

Termin **programowanie funkcyjne** (ang. *functional programming* — *FP*) odnosi się do stylu programowania, w którym wynik wywołania funkcji zależy wyłącznie od parametrów przekazanych na jej wejściu. Oznacza to, że na wynik funkcji nie ma wpływu zewnętrzny stan. Opisane działanie jest więc zgodne z matematycznym pojęciem funkcji, gdzie każde użycie jednej funkcji z tymi samymi parametrami spowoduje otrzymanie dokładnie takiego samego wyniku. W ten sposób działa wiele matematycznych funkcji biblioteki standardowej języka C++, jak `sin`, `cos` czy `sqrt`, oraz prostych operacji na typach podstawowych, jak `3+3`, `6*9` czy `1.3/4.7`. **Typowa** funkcja nie **modyfikuje** zewnętrznego stanu; skutki wykonywania tej funkcji ograniczają się tylko do zwracanej wartości.

Opisany model programowania ułatwia interpretację kodu, szczególnie jeśli program zawiera elementy przetwarzania współbieżnego, ponieważ wiele problemów związanych z pamięcią współdzieloną (opisanych w rozdziale 3.) w ogóle nie występuje w świecie programowania funkcyjnego. Skoro dane współdzielone nie są modyfikowane, nie mogą wystąpić sytuacje wyścigów, zatem ochrona tych danych za pomocą muteksów jest po prostu niepotrzebna. Właśnie prostota tego modelu powoduje, że takie języki programowania jak Haskell<sup>2</sup>, gdzie wszystkie funkcje **domyślnie** spełniają warunek programowania funkcyjnego, zyskują coraz większą popularność wśród programistów systemów współbieżnych. Ponieważ niemal cały kod jest zgodny z zasadami programowania funkcyjnego, nieliczne funkcje, które **modyfikują** współdzielony stan, na tyle wyróżniają się spośród pozostałych elementów, że można bez trudu ocenić ich udział w całej strukturze aplikacji.

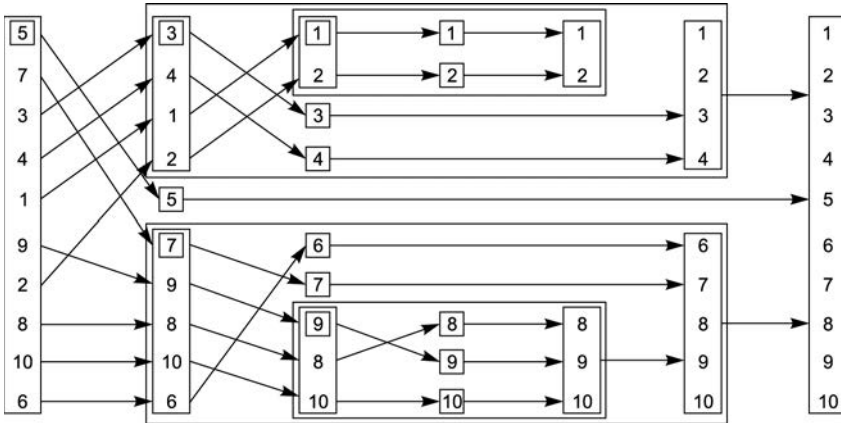
Zalety programowania funkcyjnego nie ograniczają się tylko do języków, w których ten model jest domyślnym paradygmatem. Język C++ łączy w sobie wiele paradygmatów, zatem także pisanie programów według zasad programowania funkcyjnego jest możliwe w tym języku. W wersji C++11 programowanie funkcyjne jest jeszcze prostsze niż w standardzie C++98 dzięki wprowadzeniu funkcji lambda (patrz część A.5 dodatku A), integracji funkcji `std::bind` zaczerpniętej z biblioteki Boost i dokumentu TR1 oraz dodaniu automatycznego wnioskowania typów zmiennych (patrz część A.7 dodatku A). Ostatnim elementem, który ułatwia programowanie funkcyjne w języku C++, są obiekty przyszłości — obiekt przyszłości można przekazywać pomiędzy wątkami, tak aby wynik jednej operacji mógł zależeć od wyniku innej operacji i aby ta zależność **nie wymagała bezpośredniego dostępu do współdzielonych danych**.

#### SZYBKI SORTOWANIE W MODELU PROGRAMOWANIA FUNKCYJNEGO

Aby lepiej zrozumieć możliwe zastosowania przyszłości w modelu programowania funkcyjnego, przeanalizujmy prostą implementację algorytmu sortowania szybkiego (ang. *quicksort*). Podstawowa koncepcja tego algorytmu jest dość prosta — należy z listy wartości wybrać element dzielący, osiowy (ang. *pivot*), po czym podzielić listę na dwa zbiory, z których jeden zawiera elementy mniejsze od elementu dzielącego, a drugi zawiera elementy większe od wybranego elementu. Posortowana kopia listy jest uzyskiwana poprzez sortowanie obu podzbiorów i połączenie odpowiednio posortowanej listy złożonej z wartości mniejszych od elementu dzielącego, samego elementu dzielącego

<sup>2</sup> Patrz <http://www.haskell.org/>.

i posortowanej listy większej od elementu dzielącego. Przykład sortowania listy dziesięciu liczb całkowitych według opisanego schematu pokazano na rysunku 4.2. Na listingu 4.12 pokazano sekwencyjną implementację tego algorytmu opracowaną zgodnie z zasadami programowania funkcyjnego; funkcja `sequential_quick_sort()` otrzymuje listę i zwraca jej posortowaną kopię przez wartość (zamiast sortować listę przekazaną przez referencję, jak w przypadku funkcji `std::sort()`).



Rysunek 4.2. Rekurencyjne sortowanie w modelu programowania funkcyjnego

#### Listing 4.12. Sekwencyjna implementacja algorytmu sortowania szybkiego

```
template<typename T>
std::list<T> sequential_quick_sort(std::list<T> input)
{
    if(input.empty())
    {
        return input;
    }
    std::list<T> result;
    result.splice(result.begin(),input,input.begin()); ← 1
    T const& pivot=*result.begin(); ← 2

    auto divide_point=std::partition(input.begin(),input.end(),
        [&](T const& t){return t<pivot;}); ← 3

    std::list<T> lower_part;
    lower_part.splice(lower_part.end(),input,input.begin(),
        divide_point); ← 4

    auto new_lower(
        sequential_quick_sort(std::move(lower_part))); ← 5
    auto new_higher(
        sequential_quick_sort(std::move(input))); ← 6

    result.splice(result.end(),new_higher); ← 7
    result.splice(result.begin(),new_lower); ← 8
    return result;
}
```

Mimo że zewnętrzny interfejs tej implementacji jest zgodny z regułami programowania funkcyjnego, wewnętrzne mechanizmy zaimplementowano w „tradycyjny” sposób, ponieważ konsekwentne stosowanie modelu funkcyjnego wymagałoby wielu dodatkowych operacji kopiowania. W roli elementu dzielącego jest wybierany pierwszy element, który jest wyodrębniany z listy za pomocą funkcji `splice()` ❶. Chociaż sortowanie na bazie tak wybranego elementu dzielącego może nie być optymalne (liczba operacji porównania i wymiany może być większa, niż to konieczne), pozostałe operacje na strukturze typu `std::list` będą wykonywane szybciej dzięki efektywnemu przeszukiwaniu listy. Wiadomo, że wyodrębniony element dzielący musi się znaleźć na liście wynikowej, zatem jest od razu umieszczany w odpowiedniej strukturze. Ponieważ element dzielący będzie teraz porównywany z pozostałymi elementami, przekazujemy referencję do tego elementu, aby uniknąć wielokrotnego kopiowania ❷. Możemy następnie użyć funkcji `std::partition` do podzielenia sekwencji na wartości **mniejsze** od elementu dzielącego i wartości **nie mniejsze** od tego elementu ❸. Najprostszym sposobem określenia kryterium podziału jest użycie funkcji lambda — aby uniknąć kopiowania wartości elementu dzielącego, zastosowano tutaj technikę przechwytywania referencji (więcej informacji na temat funkcji lambda można znaleźć w części A.5 dodatku A).

Funkcja `std::partition()` przetwarza przekazaną listę i zwraca iterator wskazujący pierwszy element, który **nie** jest mniejszy od wartości elementu dzielącego. Kompletny typ iteratora może być dość długi, zatem w powyższym kodzie użyto mechanizmu automatycznej identyfikacji typów, aby to kompilator automatycznie określił odpowiedni typ (patrz część A.7 dodatku A).

Ponieważ analizowana implementacja udostępnia interfejs zgodny z zasadami programowania funkcyjnego, warunkiem rekurencyjnego posortowania obu „połówek” listy jest utworzenie dwóch odrębnych list. Do tego celu możemy ponownie użyć funkcji `splice()`, aby skopiować wartości z listy wejściowej (do elementu `divide_point`) i umieścić na nowej liście: `lower_part` ❹. Reszta wartości pozostanie na liście wejściowej. Obie listy można następnie posortować za pomocą rekurencyjnych wywołań funkcji `sequential_quick_sort()` ❺ ❻. Użycie funkcji `std::move()` podczas przekazywania list na wejściu rekurencyjnych wywołań pozwala uniknąć kopiowania tych struktur (wyniki obu wywołań są kopiowane automatycznie). I wreszcie możemy ponownie użyć funkcji `splice()` w celu uporządkowania list reprezentujących podzbiory elementów oryginalnej struktury. Wartości z listy `new_higher` trafiają na koniec listy wynikowej ❼ (za element dzielący), natomiast wartości z listy `new_lower` są umieszczane na początku listy ❽ (przed elementem dzielącym).

#### **SZYBKE SORTOWANIE RÓWNOLEGLE W MODELU PROGRAMOWANIA FUNKCYJNEGO**

Ponieważ już w poprzednim przykładzie zastosowano reguły programowania funkcyjnego, konwersja tego algorytmu na wersję równoległą (korzystającą z obiektów przyszłości) nie jest specjalnie trudna (patrz listing 4.13). Zbiór operacji jest taki sam jak w poprzedniej wersji, tyle że teraz część tych operacji jest wykonywana równoległe. W tej wersji użyto implementacji algorytmu sortowania szybkiego łączącej obiekty przyszłości i model programowania funkcyjnego.

Jedną z najważniejszych różnic dzielących obie wersje jest to, że w wersji współbieżnej część listy sprzed elementu dzielącego nie jest sortowana w bieżącym wątku, tylko w dodatkowym wątku — w tym celu zastosowano funkcję `std::async()` ❶. Druga



Listing 4.13. Równoległe sortowanie szybkie z wykorzystaniem przyszłości

```

template<typename T>
std::list<T> parallel_quick_sort(std::list<T> input)
{
    if(input.empty())
    {
        return input;
    }
    std::list<T> result;
    result.splice(result.begin(),input,input.begin());
    T const& pivot=*result.begin();

    auto divide_point=std::partition(input.begin(),input.end(),
        [&](T const& t){return t<pivot;});

    std::list<T> lower_part;
    lower_part.splice(lower_part.end(),input,input.begin(),
        divide_point);

    std::future<std::list<T> > new_lower( ← ❶
        std::async(&parallel_quick_sort<T>,std::move(lower_part)));

    auto new_higher(
        parallel_quick_sort(std::move(input)); ← ❷

    result.splice(result.end(),new_higher); ← ❸
    result.splice(result.begin(),new_lower.get()); ← ❹
    return result;
}

```

część listy jest sortowana tak jak w poprzedniej wersji, a więc przy użyciu bezpośredniego wywołania rekurencyjnego ❷. Rekurencyjne wywołanie funkcji `parallel_quick_sort()` pozwala wykorzystać dostępną współbieżność sprzętową. Jeśli wywołanie funkcji `std::async()` za każdym razem uruchamia nowy wątek, wystarczą trzy poziomy rekurencji, aby program został podzielony na osiem wątków; w przypadku dziesięciu poziomów rekurencji (czyli około tysiąca elementów) program w tej formie uruchomi 1024 wątki (o ile stosowany sprzęt poradzi sobie z taką liczbą). W razie wykrycia zbyt dużej liczby uruchomionych zadań (jeśli na przykład liczba równoległe realizowanych zadań przekroczy dostępną współbieżność sprzętową) biblioteka może przejść w tryb synchronicznego uruchamiania nowych zadań. Zadania będą wykonywane w wątku wywołującym funkcję `get()`, nie w nowym wątku, zatem program uniknie kosztów przekazywania zadania pomiędzy wątkami, jeśli koszty tej operacji nie są rekompensowane przez wzrost wydajności. Jeśli nie przekazano wprost wartości `std::launch::deferred`, uruchamianie nowego wątku dla każdego zadania jest w pełni zgodne z założeniami implementacji `std::async` (nawet jeśli prowadzi do nadsubskrypcji); podobnie jeśli nie przekazano wartości `std::launch::async`, najlepszym rozwiązaniem jest synchroniczne wykonywanie wszystkich zadań. W przypadku stosowania biblioteki oferującej mechanizmy automatycznego skalowania warto sprawdzić w dokumentacji, jak te mechanizmy będą działały w kontekście tego algorytmu.

Zamiast funkcji `std::async()` moglibyśmy użyć własnej funkcji `spawn_task()` w roli prostego opakowania szablonu klasy `std::packaged_task` i klasy `std::thread` (patrz

listing 4.14). W takim przypadku należałoby utworzyć obiekt klasy `std::packaged_task` dla wyniku wywołania funkcji, odczytać obiekt przyszłości z obiektu zadania, uruchomić zadanie w odpowiednim wątku, po czym zwrócić obiekt przyszłości. Takie rozwiązanie samo w sobie nie przyniosłoby co prawda żadnych korzyści (prowadziłoby raczej do dużej nadsubskrypcji), ale może stanowić punkt wyjścia dla bardziej zaawansowanych implementacji, które będą dodawały zadania do kolejki w celu przetworzenia przez wątki robocze dostępne w puli. Zagadnienia związane z pulami wątków zostaną omówione w rozdziale 9. Wybór tego kierunku (zamiast stosowania funkcji `std::async`) jest uzasadniony tylko w przypadku programistów, którzy mają pełną świadomość skutków podejmowanych działań i chcą mieć pełną kontrolę nad sposobem budowy puli wątków i wykonywania zadań.

#### Listing 4.14. Prosta implementacja funkcji `spawn_task`

```
template<typename F,typename A>
std::future<std::result_of<F(A&&)>::type>
  spawn_task(F&& f,A&& a)
{
    typedef std::result_of<F(A&&)>::type result_type;
    std::packaged_task<result_type(A&&)>
        task(std::move(f));
    std::future<result_type> res(task.get_future());
    std::thread t(std::move(task),std::move(a));
    t.detach();
    return res;
}
```

Wróćmy teraz do funkcji `parallel_quick_sort()`. Ponieważ do uzyskania listy `new_higher` użyliśmy bezpośredniego wywołania rekurencyjnego, możemy użyć funkcji `splice()` tak jak w algorytmie jednowątkowym 3. Okazuje się jednak, że zmienna `new_lower` zawiera obiekt klasy `std::future<std::list<T>>`, nie listę, zatem przed wywołaniem funkcji `splice()` musimy wywołać funkcję `get()`, aby uzyskać odpowiednią wartość 4. Wywołanie w tej formie czeka na zakończenie zadania wykonywanego w tle, po czym **przenosi** wynik do wywołania funkcji `splice()`. Funkcja `get()` zwraca referencję do r-wartości wyniku, zatem lista może zostać przeniesiona (więcej informacji na temat referencji do r-wartości i operacji przenoszenia można znaleźć w części A.1.1 dodatku A).

Nawet jeśli przyjąć, że funkcja `std::async()` w optymalny sposób wykorzystuje dostępną współbieżność sprzętową, proponowana implementacja równoległego algorytmu sortowania szybkiego wciąż nie jest optymalna. Funkcja `std::partition` realizuje co prawda znaczną część zadań związanych z działaniem tego algorytmu, ale jej wywołanie ma charakter typowo sekwencyjny. Czytelnicy zainteresowani możliwie najszybszą, równoległą implementacją powinni sięgnąć po odpowiednią literaturę akademicką.

Programowanie funkcyjne nie jest jedynym paradygmatem programowania współbieżnego eliminującym problem współdzielenia zmiennych danych; innym przykładem takiego paradygmatu jest komunikacja procesów sekwencyjnych (ang. *Communicating Sequential Processes* — CSP)<sup>3</sup>, gdzie wątki są w założeniu całkowicie niezależne i nie

<sup>3</sup> C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985. Książka jest dostępna za darmo pod adresem <http://www.usingcsp.com/cspbook.pdf>.

operują na współdzielonych danych — zamiast tego wymieniają komunikaty za pośrednictwem kanałów komunikacyjnych. Paradigmat CSP zastosowano w języku programowania Erlang (<http://www.erlang.org/>) oraz w środowisku MPI (od ang. *Message Passing Interface*) stosowanym w systemach implementowanych w językach C i C++, które muszą gwarantować najwyższą wydajność (<http://www.mpi-forum.org/>). Po tym, co już napisałem, jestem pewien, że wiadomość o możliwości implementacji tego paradymatu także w języku C++ nie będzie dla czytelnika żadnym zaskoczeniem — wystarczy odrobina dyscypliny. Sposób implementacji tego modelu omówię w następnym punkcie.

#### 4.4.2. Synchronizacja operacji za pomocą przesyłania komunikatów

Koncepcja paradymatu CSP jest prosta — nie istnieją żadne współdzielone dane, a każdy wątek można traktować jako zupełnie niezależny byt. Zachowanie wątku zależy wyłącznie od komunikatów, które do niego trafiają. Każdy wątek jest więc swoistą maszyną stanów, która po otrzymaniu komunikatu aktualizuje swój stan i która może (ale nie musi) wysłać co najmniej jeden komunikat do pozostałych wątków. Sposób przetwarzania komunikatu zależy od stanu początkowego „maszyny”. Jednym ze sposobów pisania wątków tego typu jest stworzenie formalnego modelu i implementacja skończonej maszyny stanów, jednak istnieją też lepsze rozwiązania — do wyrażenia maszyny stanów można użyć odpowiedniej struktury aplikacji. To, która metoda sprawdza się lepiej w danym scenariuszu, zależy od szczegółowych wymagań dotyczących zachowań budowanego systemu i od umiejętności zespołu programistów. Jeśli jednak zdecydujemy się na implementację odrębnych wątków, sam podział na niezależne procesy może prowadzić do wyeliminowania wielu komplikacji związanych ze współbieżnym przetwarzaniem danych współdzielonych i tym samym ułatwić programowanie oraz ograniczyć liczbę błędów.

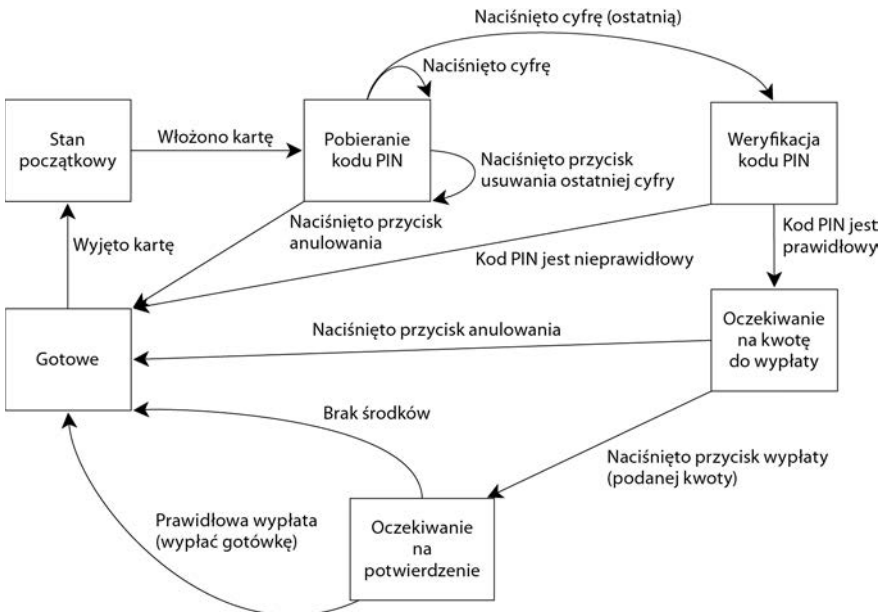
Procesy w pełni zgodne z paradigmatem CSP nie operują na żadnych współdzielonych danych, a cała komunikacja odbywa się za pośrednictwem kolejek komunikatów. Ponieważ jednak wątki języka C++ współdzielą przestrzeń adresową, wymuszenie tego wymagania jest niemożliwe. W tej sytuacji bardzo duże znaczenie ma dyscyplina autorów aplikacji i bibliotek, ponieważ to od nas zależy, czy nasze wątki będą się odwoływać do współdzielonych danych. Same kolejki komunikatów oczywiście muszą być współdzielone (w przeciwnym razie wątki nie mogłyby się komunikować), jednak szczególnej implementacji tych kolejek można opakować w ramach bibliotek.

Wyobraźmy sobie, że implementujemy kod dla bankomatu. Kod tego systemu musi obsługiwać interakcję z użytkownikiem, czyli osobą wypłacającą gotówkę, musi komunikować się z systemem odpowiedniego banku oraz musi sterować fizycznymi urządzeniami odpowiedzialnymi za akceptację karty, wyświetlanie stosownych komunikatów, obsługę klawiatury, wypłatę pieniędzy i zwracanie karty.

Jednym ze sposobów obsługi wszystkich tych zadań jest podzielenie kodu na trzy niezależne wątki: jeden obsługujący fizyczne urządzenia, drugi implementujący logikę samego bankomatu i trzeci odpowiedzialny za komunikację z bankiem. Wątki mogą się komunikować wyłącznie poprzez przekazywanie komunikatów (nie poprzez współdzielenie jakichkolwiek danych). Na przykład wątek obsługujący fizyczne urządzenia mógłby wysłać do wątku logiki bankomatu komunikat o włożeniu karty lub naciśnięciu

przycisku przez użytkownika, natomiast wątek logiki mógłby wysłać do wątku obsługującego fizyczne urządzenie komunikat określający kwotę do wypłacenia.

Jednym ze sposobów modelowania logiki bankomatu jest opracowanie maszyny stanów. W każdym stanie wątek oczekuje na określony komunikat, który jest następnie odpowiednio przetwarzany. W wyniku przetwarzania tego komunikatu wątek może przejść w nowy stan i kontynuować cały cykl. Stany składające się na tę prostą implementację pokazano na rysunku 4.3. W tej uproszczonej implementacji system oczekuje na umieszczenie karty w bankomacie. Po włożeniu karty system czeka, aż użytkownik wpisze kod PIN, naciskając kolejno cyfry tego kodu. Użytkownik może usunąć ostatnią wpisaną cyfrę. Po wpisaniu odpowiedniej liczby cyfr system weryfikuje kod PIN. Jeśli PIN jest nieprawidłowy, cykl działania kończy się — bankomat zwraca kartę i przechodzi w stan oczekiwania na wsunięcie karty przez klienta. Jeśli kod PIN jest prawidłowy, system czeka na anulowanie transakcji albo na wybór kwoty do wypłacenia. W razie anulowania transakcji cykl pracy bankomatu kończy się i urządzenie zwraca kartę. Jeśli klient wybrał kwotę, przed wypłaceniem gotówki system czeka na potwierdzenie ze strony banku, po czym albo wypłaca gotówkę, albo wyświetla komunikat „brak środków” i (niezależnie od wyniku weryfikacji stanu konta) wysuwa kartę. Systemy prawdziwych bankomatów są oczywiście bardziej skomplikowane, jednak opisana powyżej maszyna stanów dobrze ilustruje istotę tego rozwiązania.



**Rysunek 4.3. Prosty model maszyny stanów dla bankomatu**

Po zaprojektowaniu maszyny stanów dla logiki bankomatu możemy przystąpić do implementacji tego rozwiązania w formie klasy, która będzie definiowała po jednej funkcji składowej dla każdego stanu. Każda funkcja składowa może czekać na określony zbiór komunikatów przychodzących i odpowiednio obsługiwać te komunikaty (obsługa może polegać na przechodzeniu do innego stanu). Każdy typ komunikatów jest reprezentowany

przez odrębną strukturę. Na listingu 4.15 pokazano fragment prostej implementacji logiki takiego systemu, w tym główną pętlę oraz implementację pierwszego stanu, w którym system oczekuje na włożenie karty.

**Listing 4.15. Prosta implementacja klasy logiki systemu bankomatu**

```

struct card_inserted
{
    std::string account;
};
class atm
{
    messaging::receiver incoming;
    messaging::sender bank;
    messaging::sender interface hardware;
    void (atm::*state)();

    std::string account;
    std::string pin;

    void waiting_for_card() ← ❶
    {
        interface hardware.send(display_enter_card()); ← ❷
        incoming.wait() ← ❸
        .handle<card_inserted>(
            [&](card_inserted const& msg) ← ❹
            {
                account=msg.account;
                pin="";
                interface hardware.send(display_enter_pin());
                state=&atm::getting_pin;
            }
        );
    }
    void getting_pin():
public:
    void run() ← ❺
    {
        state=&atm::waiting_for_card; ← ❻
        try
        {
            for(;;)
            {
                (this->*state)(); ← ❼
            }
        }
        catch(messaging::close_queue const&)
        {
        }
    }
};

```

Jak widać, wszystkie niezbędne operacje związane z synchronizacją przekazywania komunikatów zostały ukryte w odpowiedniej bibliotece (implementacja tej biblioteki zostanie pokazana w dodatku C wraz z kompletnym kodem tego przykładu).

Jak już wspomniałem, opisana tutaj implementacja jest mocno uproszczona w stosunku do logiki obowiązującej w prawdziwych bankomatach, jednak przykład w tej formie wystarczy do zrozumienia stylu programowania na bazie przekazywania komunikatów. Nie musimy tracić czasu na projektowanie synchronizacji i rozwiązywanie problemów związanych z przetwarzaniem współbieżnym — wystarczy ustalić, jakie komunikaty mogą być odbierane i przetwarzane na poszczególnych etapach oraz które komunikaty należy wysłać. Maszyna stanów logiki bankomatu jest przetwarzana przez jeden wątek; pozostałe elementy systemu, jak interfejs łączący się z bankiem czy interfejs terminala, są obsługiwane przez odrębne wątki. Ten styl projektowania oprogramowania określa się mianem **modelu aktorów** — w systemie istnieje wiele odrębnych aktorów (każdy działa w osobnym wątku), które wymieniają pomiędzy sobą komunikaty niezbędne do realizacji swoich zadań. W modelu aktorów nie istnieje współdzielony stan (wyjątkiem jest mechanizm potrzebny do bezpośredniego przekazywania komunikatów).

Działanie programu rozpoczyna się od funkcji składowej `run()` **5**, która ustawia stan początkowy, czyli `waiting_for_card` **6**, po czym wielokrotnie wywołuje funkcję składową reprezentującą bieżący stan **7** (niezależnie od tego, który to stan). Funkcje stanów mają postać prostych funkcji składowych klasy `atm`. Także funkcja stanu `waiting_for_card` **1** jest dość prosta — jej działanie ogranicza się do wysłania komunikatu do interfejsu w celu wyświetlenia na ekranie tekstu *Czekam na kartę* **2**; zaraz potem funkcja rozpoczyna oczekiwanie na komunikat do obsłużenia **3**. Jedynym rodzajem komunikatów, który może być obsługiwany w tej części kodu, jest komunikat `card_inserted`. Do jego obsługi używamy funkcji lambda **4**. Na wejściu funkcji `handle` można przekazać dowolną funkcję lub dowolny obiekt funkcji, jednak najprostszym rozwiązaniem jest użycie funkcji lambda. Łatwo zauważyć, że wywołanie funkcji `handle()` znalazło się w łańcuchu obejmującym wywołanie funkcji `wait()`, zatem w razie otrzymania komunikatu, który nie pasuje do wskazanego typu, wątek zignoruje ten komunikat i będzie dalej czekał na komunikat właściwego typu.

Sama funkcja lambda zapisuje numer konta w zmiennej składowej, zeruje bieżący kod PIN, wysyła komunikat do sprzętu odpowiedzialnego za obsługę interfejsu bankomatu, aby wyświetlić prośbę o podanie kodu PIN, po czym przechodzi w stan pobierania tego kodu. Po zakończeniu działania przez funkcję obsługującą komunikat funkcja stanu zwraca sterowanie, a główna pętla programu wywołuje funkcję nowego stanu **7**.

Funkcja stanu `getting_pin` jest nieco bardziej skomplikowana, ponieważ może obsługiwać trzy różne typy komunikatów (patrz rysunek 4.3). Kod tej funkcji pokazano na listingu 4.16.

**Listing 4.16.** Funkcja stanu `getting_pin` na potrzeby prostej implementacji systemu bankomatu

```
void atm::getting_pin()
{
    incoming.wait()
        .handle<digit_pressed>( ← 1
            [&](digit_pressed const& msg)
            {
                unsigned const pin_length=4;
                pin+=msg.digit;
                if(pin.length()==pin_length)
```

```

        {
            bank.send(verify_pin(account, pin, incoming));
            state=&atm::verifying_pin;
        }
    }
)
.handle<clear_last_pressed>( ← ❷
    [&](clear_last_pressed const& msg)
    {
        if(!pin.empty())
        {
            pin.resize(pin.length()-1);
        }
    }
)
.handle<cancel_pressed>( ← ❸
    [&](cancel_pressed const& msg)
    {
        state=&atm::done_processing;
    }
);
}

```

Tym razem funkcja musi przetwarzać trzy różne typy komunikatów, zatem łańcuch wywołania funkcji `wait()` obejmuje aż trzy wywołania funkcji `handle()` ❶ ❷ ❸. Każde wywołanie funkcji `handle()` określa typ komunikatu w formie parametru szablonu, po czym przekazuje funkcję lambda, która otrzymuje na wejściu komunikat określonego typu. Ponieważ wszystkie te wywołania umieszczono w jednym łańcuchu wywołań, implementacja funkcji `wait()` „wie”, że czeka na komunikat `digit_pressed`, `clear_last_pressed` lub `cancel_pressed`. Komunikaty wszystkich innych typów będą (tak jak wcześniej) ignorowane.

W tym przypadku otrzymanie komunikatu nie musi prowadzić do zmiany stanu. Jeśli na przykład wątek otrzyma komunikat `digit_pressed` i jeśli wpisana cyfra nie będzie ostatnią cyfrą kodu, wystarczy tę cyfrę dopisać do łańcucha `pin`. Główna pętla na listingu 4.15 ❷ ponownie wywoła funkcję `getting_pin()`, aby czekać na następną cyfrę (bądź wyzerować kod PIN lub anulować całą operację).

Opisana procedura jest zgodna ze schematem pokazanym na rysunku 4.3. Każdy stan przedstawiony na tym rysunku jest implementowany przez inną funkcję składową, która czeka na odpowiednie komunikaty i na tej podstawie aktualizuje stan systemu.

Jak widać, opisany styl programowania może znacznie uprościć zadanie projektowania systemu współbieżnego, ponieważ każdy wątek można traktować całkowicie niezależnie od pozostałych. W opisanym modelu wiele wątków ma za zadanie oddzielanie zagadnień i jako takie wymagają od projektanta jasnych decyzji o podziale zadań pomiędzy wątki.

## 4.5. Podsumowanie

Synchronizacja operacji pomiędzy wątkami jest ważnym aspektem pisania aplikacji stosującej techniki przetwarzania współbieżnego — w razie braku synchronizacji wątki działałyby całkowicie niezależnie, zatem równie dobrze mogłyby mieć postać odrębnych aplikacji uruchamianych w formie pewnej grupy (z racji pokrewnych zadań). W tym

rozdziale omówiłem rozmaite sposoby synchronizacji operacji, w tym podstawowe zmienne warunkowe, przyszłości, obietnice i opakowywane zadania. Opisałem też sposoby rozwiązywania problemów związanych z synchronizacją, w szczególności programowanie funkcyjne, gdzie każde zadanie generuje wynik wyłącznie na podstawie danych wejściowych (nie uwzględnia zewnętrznego środowiska), oraz przekazywanie komunikatów, gdzie komunikacja pomiędzy wątkami odbywa się za pośrednictwem asynchronicznych komunikatów (wysyłanych przy użyciu podsystemu przekazywania komunikatów).

Skoro omówiłem już wiele wysokopoziomowych rozwiązań dostępnych w języku C++, czas przyjrzeć się odpowiednim elementom niskopoziomowym, dzięki którym opisane mechanizmy mogą funkcjonować — w następnym rozdziale skoncentrujemy się na modelu pamięci języka C++ i operacjach atomowych.



# Skorowidz

---

## A

accumulate\_block, 293, 294  
add\_to\_list(), 61  
add\_to\_reclaim\_list(), 239  
aktywne oczekiwanie, 261  
algorytm sortowania szybkiego, 330  
analiza kodu, 357  
    próba szczegółowego wyjaśnienia komuś, 358  
    przeglądanie kodu, 357  
    pytania dotyczące przeglądane go kodu, 358  
asynchroniczne zadanie, 104

## B

bariera, 316  
bariery pamięci, *Patrz* ogrodzenia  
biblioteki języka C++, 29  
    ACE, 29  
    Boost, 29  
    Boost Thread Library, 30  
    C++ Thread Library, 30  
    efektywność klas, 30  
    mechanizm przyszłości, 103  
    okres, 117  
    przyszłości unikatowe, 103  
    przyszłości współdzielone, 103  
    punkt w czasie, 118  
    RAII, 29  
    standardowa, 31  
    typy wywoływalne, 36

    wolne funkcje, 150  
    zegar, 115  
blokady wirujące, 221  
błądzenie, *Patrz* uwięzienie  
Boost Thread Library, 30  
buffer, 44

## C

C++ Thread Library, 30  
clear(), 138, 141  
compare\_exchange\_strong(), 140, 144, 236, 248, 263  
compare\_exchange\_weak(), 140, 144, 225, 247

## D

data.push(), 188  
data\_cond.notify\_one(), 191  
definicja klasy stosu, 68  
delete\_nodes\_with\_no\_hazards(), 238, 240  
detach(), 38, 42  
dispatch(), 405  
do\_something(), 63  
do\_sort(), 275, 333  
done(), 110  
Double-Checked Locking, 85  
drzewo binarne, 209  
dyndający wskaźnik, 226

**E**

empty(), 64, 102, 188  
exchange(), 140, 143

**F**

falszywe współdzielenie, 284, 287  
fetch\_add(), 140, 146, 249  
fetch\_and(), 147  
fetch\_or(), 140, 147  
fetch\_sub(), 146, 176  
fetch\_xor(), 147  
find\_entry\_for(), 212  
find\_first\_if(), 217  
for\_each(), 213, 216  
frameworki aplikacji, 29  
    MFC, 29  
free\_external\_counter(), 260  
front(), 98  
funkcja lambda, 122  
funkcja początkowa, 33  
funkcje  
    accumulate\_block, 293, 294  
    add\_to\_list(), 61  
    add\_to\_reclaim\_list(), 239  
    clear(), 138, 141  
    compare\_exchange\_strong(), 140, 144, 236, 248, 263  
    compare\_exchange\_weak(), 140, 144, 225, 247  
    constexpr, 381  
        wymagania, 385  
    data.push(), 188  
    data\_cond.notify\_one(), 191  
    delete\_nodes\_with\_no\_hazards(), 238, 240  
    detach(), 38, 42  
    dispatch(), 405  
    do\_something(), 63  
    do\_sort(), 275, 333  
    domyślne, 377  
        dokumentacja, 378  
        wymuszanie generowania funkcji, 378  
        wymuszenie deklarowania konstruktora kopiującego, 378  
        wymuszenie generowania destruktora wirtualnego, 378  
        zmiana dostępności funkcji, 378  
    done(), 110  
    empty(), 64, 102, 188  
    exchange(), 140, 143  
    fetch\_add(), 140, 146, 249

fetch\_and(), 147  
fetch\_or(), 140, 147  
fetch\_sub(), 146, 176  
fetch\_xor(), 147  
find\_entry\_for(), 212  
find\_first\_if(), 217  
foo<int&>(), 375  
for\_each(), 213, 216  
free\_external\_counter(), 260  
front(), 98  
get(), 125  
get\_event(), 301  
get\_future(), 106, 114  
get\_hazard\_pointer\_for\_current\_thread(), 234, 236  
get\_id(), 52  
get\_lock(), 80  
get\_tail(), 200  
getting\_pin(), 130  
handle(), 130, 407  
head.get(), 197  
hello(), 33  
interrupt(), 340, 342, 349  
interruptible\_wait(), 343, 348  
interruption\_point(), 341, 344, 351  
is\_lock\_free(), 138  
joinable(), 295  
lambda, 386  
    wyrażenie lambda, 386  
    z konstrukcją rozpoczynającą, 388  
list\_contains(), 61  
load(), 140, 143  
lock(), 60, 79, 80, 142  
main(), 33, 36  
memcmp(), 148  
memcpy(), 148  
my\_thread, 37  
my\_x.do\_lengthy\_work(), 45  
native\_handle(), 32  
nieskładowe, 150  
notify\_one(), 96  
now(), 116  
open\_connection(), 87  
parallel\_accumulate(), 291, 296, 329  
parallel\_quick\_sort(), 126, 275  
początkowa, 33  
pop(), 64, 188, 226, 228, 235, 245, 247, 254, 257, 261  
pop\_head(), 205  
pop\_task\_from\_other\_thread\_queue(), 339  
process(), 82, 301

process\_connections(), 110  
 process\_data(), 81  
 push(), 64, 100, 191, 197, 201, 225, 229, 247,  
     254, 255, 261, 337  
 push\_front(), 216  
 reclaim\_later(), 236, 238, 239  
 remove\_if(), 217  
 run(), 130  
 run\_pending\_task(), 331, 335  
 send\_data(), 87  
 sequential\_quick\_sort(), 124  
 set(), 343  
 set\_condition\_variable(), 344  
 set\_exception(), 111  
 set\_new\_tail(), 264  
 share(), 114  
 size(), 64  
 sleep\_for(), 120  
 sleep\_until(), 120  
 some\_function, 47  
 spawn\_task(), 125  
 splice(), 124  
 square\_root(), 111  
 std::accumulate, 49  
 std::async(), 104, 125, 273, 281, 297  
 std::atomic\_compare\_exchange\_strong, 469  
 std::atomic\_compare\_exchange\_strong\_explicit,  
     469  
 std::atomic\_compare\_exchange\_weak, 470  
 std::atomic\_compare\_exchange\_weak\_explicit,  
     471  
 std::atomic\_exchange, 467  
 std::atomic\_exchange\_explicit, 467  
 std::atomic\_fetch\_add, 476, 486  
 std::atomic\_fetch\_add\_explicit, 476, 486  
 std::atomic\_fetch\_and, 478  
 std::atomic\_fetch\_and\_explicit, 479  
 std::atomic\_fetch\_or, 479  
 std::atomic\_fetch\_or\_explicit, 480  
 std::atomic\_fetch\_sub, 477, 487  
 std::atomic\_fetch\_sub\_explicit, 478, 487  
 std::atomic\_fetch\_xor, 480  
 std::atomic\_fetch\_xor\_explicit, 481  
 std::atomic\_flag\_clear, 459  
 std::atomic\_flag\_clear\_explicit, 460  
 std::atomic\_flag\_test\_and\_set, 459  
 std::atomic\_flag\_test\_and\_set\_explicit, 459  
 std::atomic\_flag::clear, 459  
 std::atomic\_flag::test\_and\_set, 458  
 std::atomic\_init, 463  
 std::atomic\_is\_lock\_free, 464  
 std::atomic\_load, 465  
 std::atomic\_load\_explicit, 465  
 std::atomic\_signal\_fence(), 457  
 std::atomic\_store, 466  
 std::atomic\_store\_explicit, 466  
 std::atomic\_thread\_fence(), 456  
 std::atomic<t\*>::fetch\_add, 486  
 std::atomic<t\*>::fetch\_sub, 487  
 std::atomic<typ-  
     calkowitoliczbowy>::fetch\_add, 476  
 std::atomic<typ-  
     calkowitoliczbowy>::fetch\_and, 478  
 std::atomic<typ-calkowitoliczbowy>::fetch\_or,  
     479  
 std::atomic<typ-  
     calkowitoliczbowy>::fetch\_sub, 477  
 std::atomic<typ-  
     calkowitoliczbowy>::fetch\_xor, 480  
 std::atomic::compare\_exchange\_strong, 468  
 std::atomic::compare\_exchange\_weak, 469  
 std::atomic::exchange, 467  
 std::atomic::is\_lock\_free, 464  
 std::atomic::load, 464  
 std::atomic::store, 466  
 std::bind(), 45, 333  
 std::call\_once, 86  
 std::chrono::duration\_cast, 428  
 std::chrono::duration::count, 423  
 std::chrono::duration::max, 426  
 std::chrono::duration::min, 426  
 std::chrono::duration::zero, 425  
 std::chrono::steady\_clock::now, 434  
 std::chrono::system\_clock::from\_time\_t, 433  
 std::chrono::system\_clock::now, 432  
 std::chrono::system\_clock::to\_time\_t, 433  
 std::chrono::time\_point::max, 431  
 std::chrono::time\_point::min, 431  
 std::chrono::time\_point::time\_since\_epoch, 430  
 std::condition\_variable\_any::notify\_all, 446  
 std::condition\_variable\_any::notify\_one, 446  
 std::condition\_variable\_any::wait, 447  
 std::condition\_variable\_any::wait\_for, 448  
 std::condition\_variable\_any::wait\_until, 450  
 std::condition\_variable::notify\_all, 437  
 std::condition\_variable::notify\_one, 437  
 std::condition\_variable::wait, 344, 438  
 std::condition\_variable::wait\_for, 439  
 std::condition\_variable::wait\_until, 441  
 std::copy\_exception(), 112  
 std::current\_exception(), 112  
 std::find, 306

## funkcje

- std::for\_each, 51, 304
- std::future::get, 494
- std::future::share, 492
- std::future::valid, 492
- std::future::wait, 493
- std::future::wait\_for, 493
- std::future::wait\_until, 494
- std::kill\_dependency(), 174
- std::lock(), 72
- std::move(), 46, 124, 329, 333
- std::mutex::lock, 516
- std::mutex::try\_lock, 516
- std::mutex::unlock, 517
- std::notify\_all\_at\_thread\_exit, 443
- std::packaged\_task::get\_future, 504
- std::packaged\_task::make\_ready\_at\_thread\_exit, 506
- std::packaged\_task::reset, 505
- std::packaged\_task::swap, 504
- std::packaged\_task::valid, 505
- std::partial\_sum, 312
- std::partition(), 124
- std::promise::get\_future, 510
- std::promise::set\_exception, 512
- std::promise::set\_exception\_at\_thread\_exit, 512
- std::promise::set\_value, 510
- std::promise::set\_value\_at\_thread\_exit, 511
- std::promise::swap, 509
- std::recursive\_mutex::lock, 519
- std::recursive\_mutex::try\_lock, 519
- std::recursive\_mutex::unlock, 519
- std::recursive\_timed\_mutex::lock, 526
- std::recursive\_timed\_mutex::try\_lock, 526
- std::recursive\_timed\_mutex::try\_lock\_for, 526
- std::recursive\_timed\_mutex::try\_lock\_until, 527
- std::recursive\_timed\_mutex::unlock, 528
- std::ref, 45
- std::shared\_future::get, 500
- std::shared\_future::valid, 498
- std::shared\_future::wait, 498
- std::shared\_future::wait\_for, 499
- std::shared\_future::wait\_until, 499
- std::terminate(), 37, 349
- std::this\_thread::get\_id, 52, 558
- std::this\_thread::sleep\_for, 559
- std::this\_thread::sleep\_until, 559
- std::this\_thread::yield, 559
- std::thread::detach, 557
- std::thread::get\_id, 558
- std::thread::hardware\_concurrency, 49, 273, 276, 280, 324, 558
- std::thread::join, 557
- std::thread::joinable, 556
- std::thread::native\_handle, 553
- std::thread::swap, 556
- std::timed\_mutex::lock, 521
- std::timed\_mutex::try\_lock, 522
- std::timed\_mutex::try\_lock\_for, 522
- std::timed\_mutex::try\_lock\_until, 523
- std::timed\_mutex::unlock, 524
- std::unique\_lock::lock, 536
- std::unique\_lock::mutex, 539
- std::unique\_lock::operator, 539
- std::unique\_lock::owns\_lock, 539
- std::unique\_lock::release, 539
- std::unique\_lock::swap, 535, 536
- std::unique\_lock::try\_lock, 536
- std::unique\_lock::try\_lock\_for, 537
- std::unique\_lock::try\_lock\_until, 538
- std::unique\_lock::unlock, 537
- store(), 140, 143, 177
- submit(), 326, 329, 335
- swap(), 64
- test\_and\_set(), 138, 141
- thread\_a(), 76
- thread\_b(), 76
- time\_since\_epoch(), 118
- top(), 64
- try\_lock(), 76, 80
- try\_lock\_for(), 120
- try\_lock\_until(), 120
- try\_pop(), 98, 191, 197, 200, 202, 337
- try\_reclaim(), 230
- try\_steal(), 337
- trywialne, 379
- unlock(), 60, 80, 348
- update\_data\_for\_widget, 44
- wait(), 96, 102, 318, 344, 348, 403
- wait\_and\_dispatch(), 405, 407
- wait\_and\_pop(), 98, 191, 202, 204
- wait\_for(), 115, 120, 344
- wait\_for\_data(), 205
- wait\_until(), 115
- worker\_thread(), 326, 335
- zegar::now(), 116

**G**

get(), 125  
 get\_event(), 301  
 get\_future(), 106, 114  
 get\_hazard\_pointer\_for\_current\_thread(), 234, 236  
 get\_id(), 52  
 get\_lock(), 80  
 get\_tail(), 200  
 getting\_pin(), 130

**H**

handle(), 130, 407  
 head.get(), 197  
 hello(), 33

**I**

identyfikowanie wątków, 52  
 identyfikatory wątków, 52  
 interrupt(), 340, 342, 349  
 interruptible\_wait(), 343, 348  
 interruption\_point(), 341, 344, 351  
 is\_lock\_free(), 138  
 iteratory jednokrotnego przebiegu, 52  
 iteratory postępujące, 52

**J**

język C++, 19  
 automatyczne określanie typu zmiennej, 395  
 biblioteka operacji atomowych, 31  
 biblioteka standardowa, 31  
 biblioteka wątków, 30  
 biblioteki, 29  
 efektywność klas, 30  
 frameworki aplikacji, 29  
 mechanizm deklarowania funkcji  
 jako usuniętej, 376  
 mechanizm przyszłości, 103  
 miejsce w pamięci, 134  
 model pamięci, 133  
 muteks, 60  
 obiekty, 134  
 obsługa współbieżności, 28, 30  
 operacje atomowe, 137  
 porządek modyfikacji, 136  
 RAII, 29  
 referencje do r-wartości, 371  
 semantyka przenoszenia danych, 372

szablony zmiennoargumentowe, 391  
 paczka parametrów, 392  
 rozwinięcie paczki, 392  
 typy definiowane przez użytkownika, 382  
 inicjalizacja statyczna, 384  
 warstwy abstrakcji, 30  
 wyrażenia stałe, 381  
 zastosowania, 381  
 wyrażenie lambda, 37  
 wyścig danych, 58  
 zmienne lokalne wątków, 396  
 join(), 39  
 joinable(), 295

**K**

## klasy

dispatcher, 404  
 receiver, 403  
 scoped\_thread, 48  
 sender, 402  
 std::thread, 36  
 std::atomic\_flag, 457  
 std::chrono::high\_resolution\_clock, 116  
 std::chrono::steady\_clock, 116, 433  
 std::chrono::system\_clock, 116, 431  
 std::condition\_variable, 95, 436  
 std::condition\_variable\_any, 95, 444  
 std::future<>, 103  
 std::mutex, 60, 515  
 std::once\_flag, 541  
 std::recursive\_mutex, 90, 517  
 std::recursive\_timed\_mutex, 524  
 std::shared\_future<>, 103  
 std::thread, 549  
 std::thread::id, 550  
 std::timed\_mutex, 520  
 thread\_guard, 48  
 thread\_pool, 331  
 komunikacja procesów sekwencyjnych, 126, 127  
 maszyna stanów, 127, 128  
 model aktorów, 130  
 model maszyny stanów, 128  
 konstruktor domyślny, 52  
 konstruktor przenoszący, 45  
 kontenery  
 std::map<>, 207  
 std::multimap<>, 207  
 std::queue<>, 97  
 std::stack, 64, 66  
 std::unordered\_map<>, 207  
 std::unordered\_multimap<>, 207

**L**

leniwa inicjalizacja, 85  
 linie pamięci podręcznej, 284  
 list\_contains(), 61  
 load(), 140, 143  
 lock(), 60, 79, 80, 142  
 l-wartość, 80

**M**

main(), 33, 36  
 mechanizm przyszłości, 52, 102, 103
 

- asynchroniczne zadanie, 104
- obietnice, 109
- oczekiwanie na wiele wątków, 112
- przyszłości unikatowe, 103
- przyszłości współdzielone, 103
- wiązanie zadania z przeszłością, 106
- wynik obliczeń wykonywanych w tle, 103
- wyścig danych, 112
- zapisywanie wyjątku, 111

 memcmp(), 148  
 memcpy(), 148  
 model aktorów, 130  
 model pamięci języka C++, 133
 

- analiza podstawowych cech strukturalnych, 134
- mechanizmy przetwarzania współbieżnego, 134
- miejsce w pamięci, 134
- modele porządkowania spójnego
  - niesekwencyjnie, 159
- niezdefiniowane zachowanie, 136
- obiekty, 134
- ogrodzenia, 178
- operacje atomowe, 136
- podział struktury, 135
- porządek modyfikacji, 136
- porządkowania pamięci, 155
- porządkowanie poprzez wzajemne
  - wykluczanie, 155, 166
  - operacje uzyskiwania, 166
  - operacje zwalniania, 166
- porządkowanie spójne sekwencyjnie, 155, 156
- porządkowanie złagodzone, 155, 160
  - wyjaśnienie porządkowania, 164
- relacja poprzedzania, 152, 154
- poprzedzanie według zależności, 173
- przechodność, 170
- relacja wprowadzania zależności, 173

relacja synchronizacji, 152
 

- sekwencja zwalniania, 175
- wyścig danych, 136

 muteks, 59, 60, 220
 

- blokowanie, 60
- blokowanie rekurencyjne, 90
- czytelników-pisarzy, 88
- elastyczne blokowanie, 79
- hierarchiczny, 77
- l-wartość, 80
- niezdefiniowane zachowanie, 90
- ochrona listy, 61
- odblokowywanie, 60
- przenoszenie własności, 80
- rekurencyjny, 90
- r-wartość, 80
- stosowanie w języku C++, 60
- szczegółowość blokady, 82
- wirujący, 142
- zakleszczenie, 71

 my\_thread, 37  
 my\_thread.detach(), 39  
 my\_thread.join(), 39  
 my\_x.do\_lengthy\_work(), 45

**N**

nadsubskrypcja, 51, 280, 285  
 native\_handle(), 32  
 niezdefiniowane zachowanie, 58, 86, 90, 136  
 niezmienniki, 56, 355  
 niskie współzawodnictwo, 282  
 notify\_one(), 96  
 now(), 116

**O**

obiekty przyszłości, 122  
 ochrona współdzielonych danych, 56
 

- blokowanie rekurencyjne, 90
- definicja klasy kolejki, 100, 190
- definicja klasy stosu, 68, 187
- Double-Checked Locking, 85
- elastyczne blokowanie muteksu, 79
- hierarchia blokad, 75
- implementacja listy z obsługą iteracji, 214
- implementacja tablicy wyszukiwania, 210
- jednowątkowa implementacja kolejki, 195
- kolejka z mechanizmami blokowania i
  - oczekiwania, 203
- kolejka ze szczegółowymi blokadami, 198

- kolejka ze sztucznym węzłem, 196
  - leniwa inicjalizacja, 85
  - lista jednokierunkowa, 194
  - metody zapewniania bezpieczeństwa, 185
  - muteks, 59, 60
  - muteks czytelników-pisarzy, 88
  - niezdefiniowane zachowanie, 58
  - niezmienniki, 56
  - ochrona listy, 61
  - pamięć transakcyjna, 59
  - programowanie bez blokad, 59
  - przekazywanie referencji, 66
  - przenoszenie własności muteksu, 80
  - stosowanie konstruktora, 67
  - struktury danych bez blokad, 220
  - sytuacja wyścigu, 58
  - szczegółowość blokady, 82
  - szeregowanie, 185
  - tablica wyszukiwania, 207
  - unikanie zakleszczeń, 71
  - wykrywanie sytuacji wyścigu, 63
  - zakleszczenie, 71
  - zwracanie wskaźnika, 67
  - ogrodzenia, 178
  - open\_connection(), 87
  - operacje atomowe, 136, 137
    - funkcje nieskładowe, 150
    - główny szablon, 147
    - modele porządkowania spójnego niesekwencyjnie, 159
  - ogrodzenia, 178
  - operacje dostępne dla typów atomowych, 149
  - operacje ładowania (odczytu), 140, 143
  - operacje odczyt-modyfikacja-zapis, 140, 141, 143, 143
  - operacje porównania-wymiany, 144
  - operacje wymiany i dodania, 146
  - operacje zapisu, 140, 141, 143
  - porównywanie bitowe, 148
  - porządkowanie pamięci, 155
  - porządkowanie poprzez wzajemne wykluczanie, 155, 166
    - operacje uzyskiwania, 166
    - operacje zwalniania, 166
  - porządkowanie spójne sekwencyjnie, 155, 156
  - porządkowanie złagodzone, 155, 160
    - wyjaśnienie porządkowania, 164
  - pozorny błąd, 144
  - relacja poprzedzania, 152, 154
  - poprzedzanie według zależności, 173
  - przechodniość, 170
  - relacja wprowadzania zależności, 173
  - relacja synchronizacji, 152
    - sekwencja zwalniania, 175
  - rozkazy porównywania i wymiany podwójnego słowa, 149
  - sekwencja zwalniania zasobów, 147
  - standardowe definicje typów atomowych, 139
  - standardowe typy atomowe, 138
  - stos bez blokad, 250
  - tworzenie typów niestandardowych, 147
  - wolne funkcje, 150
  - operator przypisania z przenoszeniem, 45
- ## P
- pamięć transakcyjna, 59
  - paradygmat CSP, *Patrz* komunikacja procesów sekwencyjnych
  - parallel\_accumulate(), 291, 296, 329
  - parallel\_quick\_sort(), 126, 275
  - ping-pong buforów, 282
  - podział zagadnień, 25
  - pop(), 64, 188, 226, 228, 235, 245, 247, 254, 257, 261
  - pop\_head(), 205
  - pop\_task\_from\_other\_thread\_queue(), 339
  - porównywanie bitowe, 148
  - potok, 278
  - pozorne budzenie, 97
  - prawo Amdahla, 299
  - problem ABA, 266
  - process(), 82, 301
  - process\_connections(), 110
  - process\_data(), 81
  - procesy demonów, 42
  - programowanie bez blokad, 59
  - programowanie funkcyjne, 122
    - algorytm sortowania szybkiego, 122
    - funkcja lambda, 122
    - obiekty przyszłości, 122
    - rekurencyjne sortowanie, 123
    - równoległe sortowanie szybkie, 125
    - wnioskowanie typów zmiennych, 122
  - projektowanie uniwersalnej kolejki, 97
  - projektowanie współbieżnego kodu, 269
    - bezpieczeństwo wyjątków, 291, 293, 297
    - algorytmy równoległe, 291
  - czynniki wpływające na wydajność kodu, 279
  - falszywe współdzielenie, 284
  - liczba procesorów, 280
  - nadsubskrypcja, 280, 285
  - niskie współzawodnictwo, 282

- projektowanie współbieżnego kodu
  - ping-pong buforów, 282
  - przełączanie zadań, 285
  - współzawodnictwo o dane, 281
  - wybór najlepszego algorytmu, 281
  - wysokie współzawodnictwo, 282
  - zmiana elementu danych, 280
- falszywe współdzielenie, 287
- łatwość testowania, 361
  - eliminacja współbieżności, 362
  - warunki struktury kodu, 361
- mnożenie macierzy, 287
- podział elementów tablicy, 287
- potok, 278
- praktyka, 303
  - bariera, 316
  - implementacja równoległego algorytmu wyszukiwania, 307
  - równoległa implementacja funkcji `partial_sum`, 319
  - równoległa wersja funkcji `std::for_each`, 304
  - równoległe obliczanie sum częściowych, 313
- prawo Amdahla, 299
- pula wątków, 276
- sąsiedztwo danych, 287
- skalowalność, 291, 298
- techniki dzielenia pracy pomiędzy wątki, 270
  - dzielenie przed rozpoczęciem przetwarzania, 271
  - dzielenie sekwencji zadań, 278
  - dzielenie według typu zadania, 276
  - podział sąsiadujących fragmentów danych, 272
  - rekurencyjne dzielenie danych, 272, 273
  - równoległy algorytm sortowania szybkiego, 273
  - sposób izolowania zagadnień, 277
- techniki przetwarzania równoległego, 301
- ukrywanie opóźnień, 300
- współzawodnictwo, 286
- wzorce dostępu do danych, 289
- przekazywanie argumentów do funkcji wątku, 43
  - konstruktor przenoszący, 45
  - kopiowane, 43
  - operator przypisania z przenoszeniem, 45
  - przenoszenie, 45
- przełączanie kontekstu, 21
- przełączanie zadań, 21, 22
- przenoszenie własności wątku, 46, 47
- przetwarzanie współbieżne, 29
  - bezpieczeństwo, 184
  - definicja klasy kolejki, 190
  - definicja klasy stosu, 68
  - implementacja listy z obsługą iteracji, 214
  - implementacja tablicy wyszukiwania, 210
  - kolejka z mechanizmami blokowania i oczekiwania, 203
  - kolejka ze szczegółowymi blokadami, 198
  - mechanizmy przyszłości, 102
  - międzywątkowa relacja poprzedzania, 155
  - niechciane blokowanie, 354
  - oczekiwanie na zewnętrzne dane wejściowe, 355
  - uwięzienie, 354
  - zakleszczenie, 354
- oczekiwanie na zdarzenie, 94
- stos bez blokad, 227, 250
- synchronizacja operacji, 93
- sytuacje wyścigu, 355
  - naruszone niezmienniki, 355
  - problemy związane z czasem życia, 356
  - wyścig danych, 355
- tablica mieszająca, 209
- tablica wyszukiwania, 207
- techniki lokalizacji błędów, 357
  - analiza kodu, 357
  - próba szczegółowego wyjaśnienia komuś, 358
  - pytania dotyczące przeglądanego kodu, 358
  - testowanie współbieżnego kodu, 359
- współdzielenie danych przez wątki, 55
- wywołanie blokujące z limitem czasowym, 115
- zarządzanie wątkami, 36
  - identyfikowanie wątków, 52
  - mechanizm przyszłości, 52
  - oczekiwanie na zakończenie wątku, 39
  - oczekiwanie w razie wystąpienia wyjątku, 39
  - odłączenie wątku, 41
  - przekazywanie argumentów do funkcji wątku, 43
  - przenoszenie własności wątku, 46
  - uruchamianie wątków w tle, 42
  - uruchamianie wątku, 36
  - wątki demonów, 42
  - wybór liczby wątków, 49
  - zmienna warunkowa, 95
- pula wątków, 106, 276, 324
  - wiązanie zadania z przeszłością, 106
- `push()`, 64, 100, 191, 197, 201, 225, 229, 247, 254, 255, 261, 337
- `push_front()`, 216



## R

RAII, 29  
 reclaim\_later(), 236, 238, 239  
 referencje do r-wartości, 371  
   semantyka przenoszenia danych, 372  
   konstruktor kopiujący, 374  
   konstruktor przenoszący, 374  
 relacja poprzedzania, 154  
   poprzedzanie według zależności, 173  
   przechodność, 170  
   relacja wprowadzania zależności, 173  
 relacja synchronizacji, 152  
   sekwencja zwalniania, 175  
 remove\_if(), 217  
 Resource Acquisition Is Initialization, *Patrz* RAI  
 RIAA, 40  
 rozkazy porównywania i wymiany podwójnego  
   słowa, 149  
 równoległość, *Patrz* zrównoległanie zadań  
 równoległość danych, 26  
 run(), 130  
 run\_pending\_task(), 331, 335  
 r-wartość, 80

## S

sąsiedztwo danych, 287  
 scoped\_thread, 48  
 sekwencja zwalniania, 175  
 send\_data(), 87  
 sequential\_quick\_sort(), 124  
 set(), 343  
 set\_condition\_variable(), 344  
 set\_exception(), 111  
 set\_new\_tail(), 264  
 share(), 114  
 size(), 64  
 skalowalność, 298, 369  
 sleep\_for(), 120  
 sleep\_until(), 120  
 some\_function, 47  
 spawn\_task(), 125  
 splice(), 124  
 square\_root(), 111  
 std::accumulate, 49  
 std::async, 104, 125, 273, 281, 297, 513  
 std::atomic, 138, 140, 147, 460  
 std::atomic\_compare\_exchange\_strong, 469  
 std::atomic\_compare\_exchange\_strong\_explicit,  
   469  
 std::atomic\_compare\_exchange\_weak, 470  
 std::atomic\_compare\_exchange\_weak\_explicit, 471  
 std::atomic\_exchange, 467  
 std::atomic\_exchange\_explicit, 467  
 std::atomic\_fetch\_add, 476, 486  
 std::atomic\_fetch\_add\_explicit, 476, 486  
 std::atomic\_fetch\_and, 478  
 std::atomic\_fetch\_and\_explicit, 479  
 std::atomic\_fetch\_or, 479  
 std::atomic\_fetch\_or\_explicit, 480  
 std::atomic\_fetch\_sub, 477, 487  
 std::atomic\_fetch\_sub\_explicit, 478, 487  
 std::atomic\_fetch\_xor, 480  
 std::atomic\_fetch\_xor\_explicit, 481  
 std::atomic\_flag, 138, 141, 457  
 std::atomic\_flag\_clear, 459  
 std::atomic\_flag\_clear\_explicit, 460  
 std::atomic\_flag\_test\_and\_set, 459  
 std::atomic\_flag\_test\_and\_set\_explicit, 459  
 std::atomic\_flag::clear, 459  
 std::atomic\_flag::test\_and\_set, 458  
 std::atomic\_init, 463  
 std::atomic\_is\_lock\_free, 464  
 std::atomic\_load, 465  
 std::atomic\_load\_explicit, 465  
 std::atomic\_signal\_fence(), 457  
 std::atomic\_store, 466  
 std::atomic\_store\_explicit, 466  
 std::atomic\_thread\_fence(), 456  
 std::atomic<bool>, 143  
 std::atomic<int>, 147  
 std::atomic<T\*>, 146  
 std::atomic<t\*>::fetch\_add, 486  
 std::atomic<t\*>::fetch\_sub, 487  
 std::atomic<typ-calkowitoliczbowy>::fetch\_add,  
   476  
 std::atomic<typ-calkowitoliczbowy>::fetch\_and,  
   478  
 std::atomic<typ-calkowitoliczbowy>::fetch\_or,  
   479  
 std::atomic<typ-calkowitoliczbowy>::fetch\_sub,  
   477  
 std::atomic<typ-calkowitoliczbowy>::fetch\_xor,  
   480  
 std::atomic<unsigned long long>, 147  
 std::atomic::compare\_exchange\_strong, 468  
 std::atomic::compare\_exchange\_weak, 469  
 std::atomic::exchange, 467  
 std::atomic::is\_lock\_free, 464  
 std::atomic::load, 464  
 std::atomic::store, 466

std::bind, 45, 333  
 std::call\_once, 86, 542  
 std::chrono::duration, 117, 420  
 std::chrono::duration\_cast, 428  
 std::chrono::duration::count, 423  
 std::chrono::duration::max, 426  
 std::chrono::duration::min, 426  
 std::chrono::duration::zero, 425  
 std::chrono::high\_resolution\_clock, 116  
 std::chrono::steady\_clock, 116, 433  
 std::chrono::steady\_clock::now, 434  
 std::chrono::system\_clock, 116, 431  
 std::chrono::system\_clock::from\_time\_t, 433  
 std::chrono::system\_clock::now, 432  
 std::chrono::system\_clock::to\_time\_t, 433  
 std::chrono::time\_point, 118, 429  
 std::chrono::time\_point::max, 431  
 std::chrono::time\_point::min, 431  
 std::chrono::time\_point::time\_since\_epoch, 430  
 std::condition\_variable, 95, 436  
 std::condition\_variable\_any, 95, 444  
 std::condition\_variable\_any::notify\_all, 446  
 std::condition\_variable\_any::notify\_one, 446  
 std::condition\_variable\_any::wait, 447  
 std::condition\_variable\_any::wait\_for, 448  
 std::condition\_variable\_any::wait\_until, 450  
 std::condition\_variable::notify\_all, 437  
 std::condition\_variable::notify\_one, 437  
 std::condition\_variable::wait, 344, 438  
 std::condition\_variable::wait\_for, 439  
 std::condition\_variable::wait\_until, 441  
 std::copy\_exception(), 112  
 std::current\_exception(), 112  
 std::find, 306  
 std::for\_each, 51, 304  
 std::future, 103, 106, 109, 112, 490  
 std::future::get, 494  
 std::future::share, 492  
 std::future::valid, 492  
 std::future::wait, 493  
 std::future::wait\_for, 493  
 std::future::wait\_until, 494  
 std::kill\_dependency(), 174  
 std::lock, 72, 540  
 std::lock\_guard, 60, 528  
 std::map<>, 207  
 std::move(), 46, 124, 329, 333  
 std::multimap<>, 207  
 std::mutex, 60, 515  
 std::mutex::lock, 516  
 std::mutex::try\_lock, 516  
 std::mutex::unlock, 517  
 std::notify\_all\_at\_thread\_exit, 443  
 std::once\_flag, 86, 541  
 std::packaged\_task, 106, 391, 501  
 std::packaged\_task::get\_future, 504  
 std::packaged\_task::make\_ready\_at\_thread\_exit, 506  
 std::packaged\_task::reset, 505  
 std::packaged\_task::swap, 504  
 std::packaged\_task::valid, 505  
 std::partial\_sum, 312  
 std::partition(), 124  
 std::promise, 109, 507  
 std::promise::get\_future, 510  
 std::promise::set\_exception, 512  
 std::promise::set\_exception\_at\_thread\_exit, 512  
 std::promise::set\_value, 510  
 std::promise::set\_value\_at\_thread\_exit, 511  
 std::promise::swap, 509  
 std::queue<>, 97  
 std::ratio, 421, 544  
 std::ratio\_equal, 546  
 std::ratio\_greater, 548  
 std::ratio\_greater\_equal, 548  
 std::ratio\_less, 547  
 std::ratio\_less\_equal, 548  
 std::ratio\_not\_equal, 547  
 std::recursive\_mutex, 90, 517  
 std::recursive\_mutex::lock, 519  
 std::recursive\_mutex::try\_lock, 519  
 std::recursive\_mutex::unlock, 519  
 std::recursive\_timed\_mutex, 524  
 std::recursive\_timed\_mutex::lock, 526  
 std::recursive\_timed\_mutex::try\_lock, 526  
 std::recursive\_timed\_mutex::try\_lock\_for, 526  
 std::recursive\_timed\_mutex::try\_lock\_until, 527  
 std::recursive\_timed\_mutex::unlock, 528  
 std::ref, 45  
 std::shared\_future, 103, 113, 495  
 std::shared\_future::get, 500  
 std::shared\_future::valid, 498  
 std::shared\_future::wait, 498  
 std::shared\_future::wait\_for, 499  
 std::shared\_future::wait\_until, 499  
 std::stack, 64, 66  
 std::string, 44  
 std::terminate(), 37, 349  
 std::this\_thread::get\_id, 52, 558  
 std::this\_thread::sleep\_for, 120, 559  
 std::this\_thread::sleep\_until, 120, 559  
 std::this\_thread::yield, 559

std::thread, 43, 549  
 std::thread::detach, 557  
 std::thread::get\_id, 558  
 std::thread::hardware\_concurrency, 49, 273, 276, 280, 324, 558  
 std::thread::id, 550  
 std::thread::join, 557  
 std::thread::joinable, 556  
 std::thread::native\_handle, 553  
 std::thread::swap, 556  
 std::timed\_mutex, 520  
 std::timed\_mutex::lock, 521  
 std::timed\_mutex::try\_lock, 522  
 std::timed\_mutex::try\_lock\_for, 522  
 std::timed\_mutex::try\_lock\_until, 523  
 std::timed\_mutex::unlock, 524  
 std::try\_lock, 541  
 std::unique\_lock, 79, 80, 530  
 std::unique\_lock::lock, 536  
 std::unique\_lock::mutex, 539  
 std::unique\_lock::operator, 539  
 std::unique\_lock::owns\_lock, 539  
 std::unique\_lock::release, 539  
 std::unique\_lock::swap, 535, 536  
 std::unique\_lock::try\_lock, 536  
 std::unique\_lock::try\_lock\_for, 537  
 std::unique\_lock::try\_lock\_until, 538  
 std::unique\_lock::unlock, 537  
 std::unordered\_map<>, 207  
 std::unordered\_multimap<>, 207  
 store(), 140, 143, 177  
 submit(), 326, 329, 335  
 swap(), 64  
 synchronizacja współbieżnych operacji, 93  
   definicja klasy kolejki, 100  
   funkcje otrzymujące limity czasowe, 121  
   komunikacja procesów sekwencyjnych, 126  
   mechanizmy przyszłości, 102  
   obietnice, 109  
   oczekiwanie na wiele wątków, 112  
   oczekiwanie na zdarzenie, 94  
   operacje atomowe, 137  
   porządek modyfikacji, 136  
   pozorne budzenie, 97  
   programowanie funkcyjne, 122  
   projektowanie uniwersalnej kolejki, 97  
   prosta kolejka komunikatów, 401  
   przekazywanie zadań pomiędzy wątkami, 107  
   upraszczanie kodu, 121  
   wiązanianie zadania z przyszłością, 106

wynik obliczeń wykonywanych w tle, 103  
 wyścig danych, 112  
 wywołanie blokujące z limitem czasowym, 115  
 zapisywanie wyjątku na potrzeby przyszłości, 111  
   zmienna warunkowa, 95  
 sytuacja wyścigu, 58, 355  
   definicja klasy stosu, 68  
   przekazywanie referencji, 66  
   stosowanie konstruktora, 67  
   zwracanie wskaźnika, 67  
 szablon klasy  
   std::atomic, 138, 140, 147, 460  
   std::chrono::duration, 117, 420  
   std::chrono::time\_point, 118, 429  
   std::future, 103, 109, 112, 490  
   std::lock\_guard, 60, 528  
   std::packaged\_task, 106, 391, 501  
   std::promise, 109, 507  
   std::ratio, 421, 544  
   std::ratio\_equal, 546  
   std::ratio\_greater, 548  
   std::ratio\_greater\_equal, 548  
   std::ratio\_less, 547  
   std::ratio\_less\_equal, 548  
   std::ratio\_not\_equal, 547  
   std::shared\_future, 103, 113, 495  
   std::unique\_lock, 79, 80, 530  
   TemplateDispatcher, 405  
 szczegółowość blokady, 82  
 szeregowanie, 185

## T

tablica mieszająca, 209  
   kubelek, 209  
 tablica posortowana, 209  
 tablica wyszukiwania, 207  
   operacje, 208  
 test\_and\_set(), 138, 141  
 testowanie współbieżnego kodu, 360  
   biblioteka podstawowych mechanizmów, 365  
   czynniki dotyczące środowiska testowego, 361  
   projektowanie struktury wielowątkowego kodu testowego, 366  
   identyfikacja odrębnych fragmentów poszczególnych testów, 366  
   przykład testu dla struktury kolejki, 367  
   scenariusze testowania kolejki, 360  
   testowanie symulowanych kombinacji, 364  
   wady, 365

testowanie współbieżnego kodu  
 testowanie wydajności, 369  
 skalowalność, 369  
 testy siłowe, 363  
 wady, 364  
 thread\_a(), 76  
 thread\_b(), 76  
 thread\_guard, 48  
 thread\_pool, 331  
 time\_since\_epoch(), 118  
 top(), 64  
 try\_lock(), 76, 80  
 try\_lock\_for(), 120  
 try\_lock\_until(), 120  
 try\_pop(), 98, 191, 197, 200, 202, 337  
 try\_reclaim(), 230  
 try\_steal(), 337  
 try-catch, 40  
 typy wywoływalne, 36

## U

unlock(), 60, 80, 348  
 update\_data\_for\_widget, 44  
 uwięzienie, 223, 354

## V

void(), 106

## W

wait(), 96, 102, 318, 344, 348, 403  
 wait\_and\_dispatch(), 405, 407  
 wait\_and\_pop(), 98, 191, 202, 204  
 wait\_for(), 115, 120, 344  
 wait\_for\_data(), 205  
 wait\_until(), 115  
 warstwy abstrakcji, 30  
 wątki demonów, 42  
 wątki sprzętowe, 21  
 wnioskowanie typów zmiennych, 122  
 worker\_thread(), 326, 335  
 wskaźniki ryzyka, 233  
 współbieżne struktury danych, 184  
 aktywne oczekiwanie, 261  
 algorytmy blokujące, 220  
 bezpieczeństwo przetwarzania  
 wielowątkowego, 184  
 blokady wirujące, 221  
 definicja klasy kolejki, 190  
 definicja klasy stosu, 187

drzewo binarne, 209  
 implementacja listy z obsługą iteracji, 214  
 implementacja tablicy wyszukiwania, 210  
 jednowątkowa implementacja kolejki, 195  
 kolejka bez blokad, 252  
 uzyskiwanie nowej referencji, 259  
 zwalnianie licznika zewnętrznego węzła, 260  
 zwalnianie referencji do węzła, 258  
 kolejka nieograniczona, 206  
 kolejka ograniczona, 206  
 kolejka z mechanizmami blokowania  
 i oczekiwania, 203  
 kolejka ze szczegółowymi blokadami, 198  
 kolejka ze sztucznym węzłem, 196  
 kolejka z jednym producentem i jednym  
 konsumentem, 253  
 lista jednokierunkowa, 194  
 problem ABA, 266  
 projektowanie przy użyciu blokad, 186  
 stos bez blokad, 227, 250  
 struktury bez blokad, 220, 221  
 eliminowanie niebezpiecznych wycieków, 228  
 kolejka, 252  
 mechanizm odzyskiwania pamięci, 230  
 problem ABA, 266  
 stos, 227  
 uwięzienie, 223  
 wskazówki dotyczące pisania struktur, 264  
 zalety i wady, 222  
 zarządzanie pamięcią, 228  
 zliczanie referencji, 242  
 zwalnianie węzłów, 229  
 struktury bez oczekiwania, 222  
 szeregowanie, 185  
 tablica mieszająca, 209  
 kubełek, 209  
 tablica posortowana, 209  
 tablica wyszukiwania, 207  
 wskazówki dotyczące projektowania, 185  
 wskaźniki ryzyka, 233  
 implementacja funkcji odzyskujących węzły, 238  
 strategie odzyskiwania węzłów, 240  
 wykrywanie węzłów, 233  
 wywołania blokujące, 220  
 zagłodzenie, 221  
 zliczanie referencji, 242  
 współbieżność, 20  
 bezpieczeństwo przetwarzania, 184  
 mechanizm przyszłości, 102  
 modele, 22  
 nadsubskrypcja, 280  
 oczekiwanie na zdarzenie, 94

operacje atomowe, 136, 137  
 podniesienia wydajności, 26  
 podział zagadnień, 25  
 projektowanie struktury danych, 184  
 przełączanie kontekstu, 21  
 przełączanie zadań, 21, 22  
 przykład programu, 32  
 równoległość danych, 26  
 synchronizacja operacji, 93  
 w języku C++, 28  
 wątki sprzętowe, 21  
 współbieżne struktury danych, 184  
 z wieloma procesami, 23  
 z wieloma wątkami, 24  
 zarządzanie wątkami, 36  
 zmienna warunkowa, 95  
 zrównoleglanie zadań, 26

współdzielenie danych przez wątki, 55  
 blokowanie rekurencyjne, 90  
 definicja klasy stosu, 68  
 Double-Checked Locking, 85  
 elastyczne blokowanie muteksu, 79  
 leniwa inicjalizacja, 85  
 lista dwukierunkowa, 56  
 muteks, 59  
 niezmienniki, 56  
 ochrona danych za pomocą muteksów, 60  
 pamięć transakcyjna, 59  
 problemy, 56  
 programowanie bez blokad, 59  
 przenoszenie własności muteksu, 80  
 sytuacja wyścigu, 58  
 szczegółowość blokady, 82  
 wyścigu danych, 58  
 zakleszczenie, 71

współzawodnictwo, 286  
 wybór liczby wątków, 49  
 nadsubskrypcja, 51  
 wyrażenie lambda, 37  
 wysokie współzawodnictwo, 282  
 zwiększenie prawdopodobieństwa, 283

wyścig danych, 58, 112, 136, 355  
 niezdefiniowane zachowanie, 58

wywołanie blokujące z limitem czasowym, 115  
 funkcje otrzymujące limity czasowe, 121  
 limity bezwzględne, 115  
 maksymalny czas blokowania wątku, 115  
 okres, 117  
 punkt w czasie, 118  
 wymuszanie oczekiwania na podstawie okresu, 117

zastosowanie limitu czasowego, 120  
 zegar, 115

## Z

zaawansowane zarządzanie wątkami, 323  
 przerywanie wykonywania wątków, 340  
 monitorowanie systemu plików w tle, 350  
 obsługa przerwań, 349  
 oczekiwanie na zmienną, 343, 346  
 pozostałe wywołania blokujące, 348  
 przerywanie zadań wykonywanych w tle, 350  
 punkt przerywania, 341  
 wykrywanie przerywania, 342

pula wątków, 324  
 algorytm sortowania szybkiego, 330  
 implementacja algorytmu sortowania  
 szybkiego, 332  
 implementacja prostej puli wątków, 325  
 kolejka umożliwiająca wykradanie zadań, 336  
 lokalne kolejki zadań, 334  
 z mechanizmem oczekiwania na zadanie, 327  
 z mechanizmem wykradania zadań, 337  
 unikanie współzawodnictwa w dostępie do  
 kolejki, 333  
 wątek roboczy, 324  
 wykradanie zadań, 335

zakleszczenie, 71, 354  
 hierarchia blokad, 75  
 unikanie, 71, 73

zasada jednej odpowiedzialności, 277

zegar, 115  
 czasu rzeczywistego, 116  
 epoka zegara, 118  
 najkrótszy możliwy takt, 116  
 okres taktu, 116  
 stabilny, 116

zegar::now(), 116

zliczanie referencji, 242  
 licznik wewnętrzny, 243  
 licznik zewnętrzny, 243  
 stos bez blokad, 250  
 umieszczanie węzła na stosie bez blokad, 243  
 wykrywanie używanych węzłów, 242  
 zdejmowanie węzła ze stosu bez blokad, 245

zmienna warunkowa, 95  
 definicja klasy kolejki, 100  
 oczekiwanie na spełnienie warunku, 95  
 pozorne budzenie, 97  
 projektowanie uniwersalnej kolejki, 97  
 zrównoleglanie zadań, 26



# PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ 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 WYDAWNICZA

 **Helion SA**

# Sprawdź, jak tworzyć niezawodne oprogramowanie wielowątkowe!

Współbieżne przetwarzanie danych to największe wyzwanie dla programisty. Na każdym kroku czyhają na niego najbardziej wymyślne pułapki, a wykrycie pomyłki stanowi nie lada wyzwanie. Każdy programista wzdryga się na samą myśl o implementacji wielowątkowych rozwiązań. Nie musi tak być!

Dzięki tej książce poradzisz sobie z większością zadań i zwinnie ominiesz zastawione pułapki. W trakcie lektury dowiesz się, jak zidentyfikować zadania, w których zastosowanie współbieżności ma sens, oraz jak zarządzać wątkami. Ponadto nauczysz się chronić współdzielone dane oraz synchronizować współbieżne operacje. Duży nacisk został tu położony na zagadnienia związane z projektowaniem współbieżnych struktur danych oraz kodu. Osobny rozdział poświęcono debugowaniu aplikacji wielowątkowych. Książka ta jest długo oczekiwaną pozycją, która ułatwi codzienne życie programistom C++.

Dzięki tej książce:

- 1 zaprojektujesz współbieżny kod oraz struktury
- 2 ochronisz współdzielone dane
- 3 poznasz zaawansowane metody zarządzania wątkami
- 4 bez problemu przeprowadzisz debugowanie Twojej wielowątkowej aplikacji



Nr katalogowy: 12297

Księgarnia internetowa:  
<http://helion.pl>

Zamówienia telefoniczne:  
0 801 339900

0 601 339900

**helion.pl**  
księgarnia  
internetowa

Sprawdź najnowsze promocje:  
• <http://helion.pl/promocje>  
Książki najchętniej czytane:  
• <http://helion.pl/bestsellery>  
Zamów informacje o nowościach:  
• <http://helion.pl/novosci>



**Helion**

Helion SA  
ul. Kościuszki 1c, 44-100 Gliwice  
tel.: 32 230 98 63  
e-mail: [helion@helion.pl](mailto:helion@helion.pl)  
<http://helion.pl>

sięgnij po WIĘCEJ



KOD KORZYŚCI

cena 89,00 zł

ISBN 978-83-246-5086-6



9 788324 650866

Informatyka w najlepszym wydaniu