C++ BIBLIOTEKA STANDARDOWA

WYDANIE II

Lektura obowiązkowa dla każdego programisty C++!

NICOLAI M. JOSUTTIS

Tytuł oryginału: The C++ Standard Library: A Tutorial and Reference (2nd Edition)

Tłumaczenie: Przemysław Szeremiota (wstęp, rozdz. 1 – 6, 9 – 13, 15 – 17, 19, dodatek A), Radosław Meryk (rozdz. 8, 14, 18), Rafał Jońca (rozdz. 7) z wykorzystaniem fragmentów książki „C++. Biblioteka standardowa. Podręcznik programisty" w tłumaczeniu Przemysława Stecia i Rafała Szpotona

ISBN: 978-83-246-5576-2

Authorized translation from the English language edition, entitled: THE C++ STANDARD LIBRARY: A TUTORIAL AND REFERENCE, Second Edition; ISBN 0321623215; by Nicolai M. Josuttis; published by Pearson Education, Inc, publishing as Addison Wesley.

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

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/cpbsp2.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/cpbsp2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

18
Współbieżność

Nowoczesne architektury systemów zazwyczaj umożliwiają uruchamianie wielu zadań i wielu wątków jednocześnie. Wykorzystanie wielu wątków może przyczynić się do znacznej poprawy czasu realizacji programów, zwłaszcza w komputerach, których procesory są wyposażone w wiele rdzeni.

Jednak równoległe uruchamianie programów wprowadza również nowe wyzwania. Zamiast wykonywać instrukcje jedna po drugiej, można uruchomić wiele instrukcji jednocześnie. Może to prowadzić do problemów z jednoczesnym dostępem do tych samych zasobów. W takich sytuacjach operacje tworzenia, czytania, pisania i usuwania mogą odbywać się w nieoczekiwanej kolejności, co może prowadzić do nieoczekiwanych rezultatów. W rzeczywistości współbieżny dostęp do danych z wielu wątków łatwo może stać się koszmarem. Jednym z najprostszych problemów, jakie mogą się pojawić, są zakleszczenia, kiedy to wątki wzajemnie na siebie czekają.

Przed wprowadzeniem standardu C++11 język C++ ani standardowa biblioteka języka nie zawierały obsługi współbieżności, jednak poszczególne implementacje mogły zawierać pewne mechanizmy obsługi współbieżności. Wraz z powstaniem C++11 to się zmieniło. Usprawniono zarówno rdzeń języka, jak i bibliotekę, wprowadzając obsługę programowania współbieżnego (patrz podrozdział 4.5):

- W podstawowym języku zdefiniowano model pamięci, który gwarantuje, że aktualizacje dwóch różnych obiektów wykorzystywanych przez dwa różne wątki są niezależne od siebie. Wprowadzono również nowe słowo kluczowe `thread_local` do definiowania zmiennych z wartościami specyficznymi dla wątków.
- Biblioteka zapewnia obecnie wsparcie dla uruchamiania wielu wątków. Umożliwia również przekazywanie argumentów, zwracanie wartości i zgłaszanie wyjątków poza granicami wątków. Zapewnia także mechanizmy do synchronizacji wielu wątków. Dzięki temu możemy synchronizować zarówno przepływ sterowania, jak i dostęp do danych.

Biblioteka zapewnia wsparcie dla współbieżności na wielu poziomach. Na przykład interfejs wysokiego poziomu umożliwia rozpoczęcie wątku, przekazanie do niego argumentów i obsługę wyników i wyjątków. Interfejs ten bazuje na kilku interfejsach niskiego poziomu dla każdego z tych aspektów. Z drugiej strony, istnieją również własności niskiego poziomu, takie jak muteksy lub atomowe typy danych (ang. *atomics*) obsługujące swobodne kolejkowanie pamięci (ang. *relaxed memory order*).

W tym rozdziale zaprezentowano te funkcje biblioteczne. Należy zauważyć, że temat współbieżności oraz opis bibliotek, które ją obsługują, może wypełnić całe tomy. Z tego powodu w tym rozdziale zaprezentuję ogólne pojęcia i typowe przykłady dla przeciętnego programisty, z głównym naciskiem na interfejsy wysokiego poziomu.

Po szczegółowe informacje, zwłaszcza dotyczące trudnych problemów funkcji i interfejsów niskiego poziomu, odsyłam do wymienionych tutaj konkretnych książek i artykułów. Moją pierwszą i najważniejszą rekomendacją dla całego tematu współbieżności jest książka *C++ Concurrency in Action* autorstwa Anthony'ego Williamsa (patrz [*Williams:C++Conc*]).

Anthony jest jednym z kluczowych światowych ekspertów w tej dziedzinie, powstanie tego rozdziału bez jego wkładu nie byłoby możliwe. Nie tylko przejrzał niniejszą książkę, ale także dostarczył pierwszej implementacji standardowej biblioteki obsługi współbieżności (patrz [*JustThread*]), napisał kilka artykułów oraz przekazał cenne opinie. Wszystko to pomogło mi w przedstawieniu tego tematu — mam nadzieję w przydatny sposób. Dodatkowo jednak chciałbym podziękować kilku innym ekspertom w dziedzinie współbieżności, którzy pomogli mi w napisaniu tego rozdziału: Hansowi Boehmowi, Scottowi Meyersowi, Bartoszowi Milewskiemu, Lawrence'owi Crowlowi i Peterowi Sommerladowi.

Niniejszy rozdział jest zorganizowany w następujący sposób:

- Najpierw zaprezentuję różne sposoby uruchamiania wielu wątków. Po wprowadzeniu w tematykę interfejsów zarówno wysokopoziomowych, jak i niskopoziomowych zaprezentuję szczegółowe informacje związane z uruchamianiem wątków.
- W podrozdziale 18.4 zamieszczono szczegółowe omówienie problemu synchronizowania wątków. Głównym problemem jest jednoczesny dostęp do danych.
- Na koniec omówiono różne funkcje służące do synchronizacji wątków i jednoczesnego dostępu do danych:
  - Muteksy i blokady (patrz podrozdział 18.5), z funkcją `call_once()` włącznie (patrz punkt 18.5.3).
  - Zmienne warunkowe (patrz podrozdział 18.6).
  - Atomowe typy danych (patrz podrozdział 18.7).

18.1. Interfejs wysokiego poziomu: async() i futury

Dla początkujących najlepszym punktem wyjścia do uruchomienia programu z obsługą wielu wątków jest skorzystanie z wysokopoziomowego interfejsu C++ biblioteki standardowej dostarczonego za pośrednictwem wywołania `std::async()` oraz klasy `std::future<>`:

- `async()` zapewnia interfejs umożliwiający uruchomienie fragmentu funkcjonalności — obiektu wywoływalnego (ang. *callable object*) (patrz podrozdział 4.4) — w tle, w osobnym wątku.
- Klasa `future<>` pozwala czekać na zakończenie wątku i zapewnia dostęp do jego wyników: zwróconej wartości lub wyjątku.

W tym rozdziale szczegółowo zaprezentowano ten interfejs wysokiego poziomu. Temat rozszerzono o wprowadzenie do klasy `std::shared_future<>`, która pozwala na oczekiwanie na zakończenie wątku i przetwarzanie jego wyników w wielu miejscach.

18.1.1. Pierwszy przykład użycia funkcji async() i futur

Przypuśćmy, że musimy obliczyć sumę dwóch operandów zwracaną przez dwa wywołania funkcji. Standardowy sposób zaprogramowania tej funkcjonalności jest następujący:

```
func1() + func2()
```

Oznacza to, że przetwarzanie operandów odbywa się sekwencyjnie. Program najpierw wywołuje funkcję `func1()`, a następnie wywołuje funkcję `func2()` lub odwrotnie (zgodnie z regułami języka kolejność jest niezdefiniowana). W obu przypadkach całkowity czas przetwarzania wynosi: czas wykonywania funkcji `func1()` *plus* czas wykonywania funkcji `func2()` plus czas wyliczenia sumy.

Obecnie, kiedy sprzęt wieloprocesorowy jest dostępny niemal wszędzie, możemy wykonywać obliczenia wydajniej. Możemy przynajmniej spróbować uruchomić funkcje `func1()` i `func2()` równolegle. Dzięki temu całkowity czas przetwarzania będzie równy sumie dłuższego z czasów przetwarzania funkcji `func1()` i `func2()` plus czas wyliczenia sumy.

Poniżej zaprezentowano pierwszy program realizujący obliczenia w taki sposób:

```
// concurrency/async1.cpp

#include <future>
#include <thread>
#include <chrono>
#include <random>
#include <iostream>
#include <exception>
using namespace std;
```

```
int doSomething (char c)
{
    // generator liczb losowych (wykorzystuje c jako ziarno do uzyskania różnych sekwencji)
    std::default_random_engine dre(c);
    std::uniform_int_distribution<int> id(10,1000);

    // pętla wyświetlająca znak po upływie losowego czasu
    for (int i=0; i<10; ++i) {
        this_thread::sleep_for(chrono::milliseconds(id(dre)));
        cout.put(c).flush();
    }

    return c;
}

int func1 ()
{
    return doSomething('.');
}

int func2 ()
{
    return doSomething('+');
}

int main()
{
    std::cout << "uruchomienie funkcji func1() w tle,"
              << " a funkcji func2() na pierwszym planie:" << std::endl;
    // uruchomienie funkcji func1() asynchronicznie (teraz, później lub nigdy):
    std::future<int> result1(std::async(func1));

    int result2 = func2(); // wywołanie funkcji func2() synchronicznie (tu i teraz)
    // wyświetlenie wyniku (oczekiwanie na zakończenie funkcji func1() i dodanie jej wyniku do
zmiennej result2
    int result = result1.get() + result2;

    std::cout << "\nwynik sumy func1()+func2(): " << result
              << std::endl;
}
```

W celu wizualizacji tego, co się dzieje, zasymulowaliśmy złożone przetwarzanie wewnątrz funkcji `func1()` i `func2()` poprzez wywołanie funkcji `doSomething()`, która od czasu do czasu wyświetla znak przekazany za pomocą argumentu[1] i na koniec zwraca wartość przekazanego znaku jako wartość `int`. „Od czasu do czasu" zaimplementowano poprzez wykorzystanie generatora liczb losowych, który jest odpowiedzialny za obliczenie interwałów. Interwały te są wykorzystywane w funkcji `std::this_thread::sleep_for()` do wstrzymywania bieżącego wątku (szczegółowe informacje na temat liczb losowych można znaleźć w podrozdziale 17.1, natomiast szczegółowe informacje dotyczące funkcji `sleep_for()`można znaleźć w punkcie 18.3.7). Zwróćmy uwagę, że aby generowane sekwencje liczb losowych

[1] Generowanie wyjścia przez współbieżne wątki jest możliwe, ale może skutkować przeplataniem się znaków z różnych wątków (patrz podrozdział 4.5).

były różne, potrzebujemy unikatowego ziarna (ang. *seed*), które należy przekazać do konstruktora generatora liczb losowych (w tym przykładzie w tej roli użyliśmy znaku c).

Zamiast wywołania:

```
int result = func1() + func2();
```

wywołujemy:

```
std::future<int> result1(std::async(func1));
int result2 = func2();
int result = result1.get() + result2;
```

Zatem najpierw *próbujemy* uruchomić funkcję `func1()` w tle, używając wywołania `std::async()`, a następnie przypisujemy wynik do obiektu klasy `std::future`:

```
std::future<int> result1(std::async(func1));
```

W powyższej instrukcji wywołanie `async()` *próbuje* natychmiast uruchomić przekazaną funkcjonalność asynchronicznie, w osobnym wątku. Tak więc w idealnej sytuacji funkcja `func1()` rozpoczyna działanie w tym miejscu bez blokowania funkcji `main()`. Zwracany obiekt *futury* jest konieczny z dwóch powodów:

1. Umożliwia dostęp do „przyszłego" wyniku funkcji przekazanej do funkcji `async()`. Ten wynik może być zwróconą wartością albo wyjątkiem. Obiekt futury jest wyspecjalizowany według typu zwracanej wartości uruchomionej funkcjonalności. Jeśli uruchomiono tylko zadanie w tle, które niczego nie zwraca, to musi to być obiekt `std::future<void>`.
2. Należy zapewnić, aby przekazana funkcjonalność została prędzej lub później wywołana. Zwróćmy uwagę, że napisałem: funkcja `async()` *próbuje* uruchomić przekazaną funkcjonalność. Jeśli się to nie stanie, musimy wymusić na obiekcie futury uruchomienie w chwili, gdy potrzebujemy wyniku lub gdy chcemy się upewnić, że funkcjonalność została wykonana. Z tego powodu obiekt futury jest potrzebny nawet wtedy, gdy nie jesteśmy zainteresowani wynikiem funkcjonalności uruchomionej w tle.

Do wymiany danych pomiędzy miejscem, z którego uruchomiono funkcjonalność i z którego nią zarządzamy, a zwróconym obiektem futury służy tzw. *stan współdzielony* (patrz podrozdział 18.3).

Oczywiście można również (zazwyczaj tak będziemy robić) użyć słowa kluczowego `auto` do zadeklarowania futury (w tym przykładzie chciałem to jawnie zademonstrować):

```
auto result1(std::async(func1));
```

Po drugie, funkcję `func2()` uruchamiamy na pierwszym planie. Jest to zwyczajne synchroniczne wywołanie funkcji, dlatego program blokuje się w tym miejscu:

```
int result2 = func2();
```

Zatem jeśli funkcja `func1()` została pomyślnie uruchomiona przez funkcję `async()` i jeszcze się nie zakończyła, to funkcje `func1()` i `func2()` są teraz uruchomione jednocześnie.

W trzeciej kolejności przetwarzamy sumę. To jest moment, kiedy potrzebujemy wyniku funkcji `func1()`. Aby go uzyskać, wywołujemy funkcję `get()` w odniesieniu do zwróconego obiektu futury:

```
int result = result1.get() + result2;
```

W momencie wywoływania funkcji `get()` może zachodzić jedna z trzech sytuacji:

1. Jeśli funkcja `func1()` została uruchomiona za pomocą funkcji `async()` w oddzielnym wątku i już się zakończyła, to natychmiast uzyskamy jej wynik.
2. Jeśli funkcja `func1()` została uruchomiona, ale jeszcze się nie zakończyła, to funkcja `get()` blokuje się i czeka na jej zakończenie. Na koniec zwraca wynik.
3. Jeśli funkcja `func1()` jeszcze nie została uruchomiona, będzie zmuszona do natychmiastowego uruchomienia i tak jak dla synchronicznego wywołania funkcji `get()` zablokuje się w oczekiwaniu na wynik.

To zachowanie jest bardzo ważne, ponieważ zapewnia ono działanie programu także w środowisku jednowątkowym oraz w przypadku, kiedy funkcja `async()` z jakiegoś powodu nie mogła uruchomić nowego wątku.

Wywołanie funkcji `async()` nie gwarantuje, że przekazana do niej funkcjonalność zostanie rozpoczęta i się zakończy. Jeśli wątek jest dostępny, to zostanie uruchomiony, ale jeśli nie — ze względu na to, że określone środowisko nie obsługuje wielowątkowości lub nie ma dostępnych więcej wątków — wtedy wywołanie zostanie odroczone do momentu, kiedy jawnie stwierdzimy, że potrzebujemy jego wyniku (poprzez wywołanie `get()`), lub po prostu kiedy będziemy chcieli, aby określona funkcjonalność została wykonana (poprzez wywołanie `wait()` — patrz punkt 18.1.1).

A zatem kombinacja instrukcji:

```
std::future<int> result1(std::async(func1));
```

oraz

```
result1.get()
```

pozwala zoptymalizować program w taki sposób, że jeśli to będzie możliwe, funkcja `func1()` będzie działać współbieżnie w czasie, gdy są przetwarzane następne instrukcje w głównym wątku. Jeśli nie ma możliwości jej współbieżnego uruchomienia, będzie wywołana sekwencyjnie w momencie, gdy zostanie wywołana funkcja `get()`. Oznacza to, że w każdym przypadku uzyskujemy gwarancję, że po wywołaniu `get()` funkcja `func1()` będzie wywołana asynchronicznie lub synchronicznie.

W związku z tym ten program może generować dwa rodzaje wyników. Jeśli funkcja `async()` może pomyślnie uruchomić funkcję `func1()`, wynik może mieć następującą postać:

```
uruchomienie funkcji func1() w tle, a funkcji func2() na pierwszym planie:
++..++++.++.+.+.....
wynik sumy func1()+func2(): 89
```

Jeśli funkcja `async()` nie mogła uruchomić funkcji `func1()`, funkcja ta zostanie uruchomiona po funkcji `func2()`, w momencie wywołania funkcji `get()`. W związku z tym program będzie miał następujący wynik:

```
uruchomienie funkcji func1() w tle, a funkcji func2() na pierwszym planie:
++++++++++.........
wynik sumy func1()+func2(): 89
```

Tak więc na podstawie pierwszego przykładu możemy zdefiniować ogólny sposób na to, by program działał szybciej: możemy zmodyfikować program w taki sposób, aby mógł korzystać ze współbieżności, jeśli jest dostępna na platformie, na której uruchomiono program, ale by mógł działać także w środowiskach jednowątkowych. W tym celu należy wykonać następujące czynności:

- umieścić w programie dyrektywę `#include <future>`;
- przekazać funkcjonalność, które może być uruchomiona samodzielnie jako *wywoływalny obiekt*, do funkcji `std::async()`;
- przypisać wynik do obiektu `future<`*`ZwracanyTyp`*`>`;
- wywołać funkcję `get()` na rzecz obiektu `future<>`, kiedy będzie potrzebny nam wynik lub kiedy będziemy chcieli mieć pewność, że uruchomiona funkcjonalność się zakończyła.

Należy jednak pamiętać, że dotyczy to tylko takiej sytuacji, kiedy nie występuje *wyścig o dane*, co oznacza, że dwa wątki równocześnie korzystają z tych samych danych, co może być przyczyną niezdefiniowanego zachowania (patrz punkt 18.4.1).

Zwróćmy uwagę, że bez wywołania funkcji `get()` nie mamy gwarancji, czy funkcja `func1()` kiedykolwiek będzie wywołana. Zgodnie z tym, co napisano wcześniej, jeśli funkcja `async()` nie może uruchomić przekazanej funkcjonalności natychmiast, *odracza* wywołanie do chwili jawnego zażądania wyniku przekazanej funkcjonalności za pomocą funkcji `get()` (lub `wait()`). Jednak bez takiego żądania zakończenie działania funkcji `main()` spowoduje zakończenie programu nawet bez wywołania wątku działającego w tle.

Zwróćmy także uwagę, że musimy zadbać o to, aby żądanie wyniku funkcjonalności rozpoczętej za pomocą funkcji `async()` nie nastąpiło wcześniej, niż to konieczne. Na przykład poniższa „optymalizacja" prawdopodobnie nie przyniesie spodziewanego efektu:

```
std::future<int> result1(std::async(func1));
int result = func2() + result1.get(); // wywołanie funkcji func2() może nastąpić po
zakończeniu działania funkcji func1()
```

Ponieważ kolejność wykonywania działań po prawej stronie drugiej instrukcji jest niezdefiniowana, wywołanie `result1.get()` może nastąpić przed wywołaniem `func2()`, a zatem będzie to ponownie przetwarzanie sekwencyjne.

Aby uzyskać najlepszy efekt, ogólnie rzecz biorąc, dystans pomiędzy wywołaniami `async()` i `get()` powinien być jak największy. Można również zastosować warunki określone w [*N3194:Futures*]: *Wczesne wywołanie i późne zwrócenie wyniku*.

Jeśli operacja przekazana do funkcji `async()` nie zwraca żadnej wartości, funkcja `async()` zwraca obiekt `future<void>`, który jest częściową specjalizacją obiektu `future<>`. W takim przypadku wywołanie `get()` nie zwraca niczego:

```
std::future<void> f(std::async(func)); // próba asynchronicznego wywołania funkcji
...
f.get(); // oczekiwanie na zakończenie funkcji (zwraca void)
```

Na koniec zwróćmy uwagę, że obiekt przekazany do funkcji `async()` może być *obiektem wywoływalnym* dowolnego typu: funkcją, funkcją składową, obiektem funkcyjnym lub wyrażeniem lambda (patrz podrozdział 4.4). Zatem możemy również przekazać funkcjonalność *inline* jako wyrażenie lambda. Funkcjonalność ta będzie działała we własnym wątku (patrz punkt 3.1.10):

```
std::async([]{ ... }) // próba uruchomienia kodu ... asynchronicznie
```

Strategie uruchamiania

Możemy zapobiec odroczeniu uruchomienia funkcjonalności przekazanej do funkcji `async()` poprzez jawne przekazanie *strategii uruchamiania*[2]. W ten sposób możemy nakazać funkcji `async()`, aby ta uruchomiła przekazaną funkcjonalność dokładnie w momencie wywołania:

```
// wymuszenie asynchronicznego uruchomienia funkcji func1(); w przypadku niepowodzenia zgłaszany
// jest wyjątek std::system_error
std::future<long> result1= std::async(std::launch::async, func1);
```

Jeśli wywołanie asynchroniczne nie jest możliwe w tym miejscu, program zgłosi wyjątek `std::system_error` (patrz punkt 4.3.1) z kodem błędu `resource_unavailable_try_again`. Jest to odpowiednik kodu błędu POSIX `errno EAGAIN` (patrz punkt 4.3.2).

Przy użyciu strategii uruchamiania `async` nie musimy wywoływać funkcji `get()`, ponieważ jeśli czas życia zwróconej futury dobiegnie końca, program zaczeka na zakończenie działania funkcji `func1()`. Jeśli więc nie wywołamy funkcji `get()`, to po opuszczeniu zakresu obiektu futury (tutaj będzie nim koniec funkcji `main()`) program będzie czekał na zakończenie zadania działającego w tle. Niemniej jednak wywołanie funkcji `get()` zanim program zakończy działanie, sprawia, że zachowanie kodu staje się czytelniejsze.

Jeśli nigdzie nie przypiszemy wyniku `std::async(std::launch::async,...)`, obiekt wywołujący zablokuje się do czasu zakończenia przekazanej funkcjonalności. W takim przypadku będzie to równoznaczne z wywołaniem synchronicznym[3].

W podobny sposób możemy wymusić odroczone wywołanie poprzez przekazanie do funkcji `async()` strategii uruchamiania `std::launch:deferred`. Poniższa sekwencja instrukcji spowoduje odroczenie funkcji `func1()` do czasu wywołania funkcji `get()` na rzecz obiektu `f`:

---

[2] Strategia uruchamiania jest *typem wyliczeniowym o określonym zasięgu*, dlatego trzeba kwalifikować wartości (enumeratory) za pomocą prefiksu `std::launch` lub `launch` (patrz punkt 3.1.13).

[3] Należy zauważyć, że w komitecie standaryzacyjnym były kontrowersje dotyczące sposobu interpretacji tych słów w przypadku, gdy wynik funkcji `async()` nie jest używany. Taki był rezultat dyskusji i takie powinno być zachowanie programu we wszystkich implementacjach.

```
std::future<...> f(std::async(std::launch::deferred,
                    func1)); // odroczenie funkcji func1 do czasu wywołania get()
```

W tym przypadku mamy gwarancję, że funkcja `func1()` nie zostanie wywołana bez wywołania `get()` (lub `wait()`).

Ta strategia umożliwia tzw. *leniwe wartościowanie* (ang. *lazy evaluation*). Na przykład[4]:

```
auto f1 = std::async( std::launch::deferred, task1 );
auto f2 = std::async( std::launch::deferred, task2 );
...
auto val = thisOrThatIsTheCase() ? f1.get() : f2.get();
```

Ponadto jawne żądanie strategii uruchamiania `deferred` może pomóc w zasymulowaniu działania funkcji `async()` w środowisku jednowątkowym. Ułatwia również debugowanie (o ile nie zachodzą warunki wyścigu).

Obsługa wyjątków

Dotychczas omawialiśmy tylko taki przypadek, kiedy wątki i zadania wykonywane w tle kończyły się pomyślnie. Jednak co się zdarzy, kiedy wystąpi wyjątek?

Dobra wiadomość jest taka, że nic specjalnego. Wywołanie funkcji `get()` dla futur obsługuje również wyjątki. Jeśli nastąpi wywołanie `get()`, a operacja w tle zostanie przerwana przez wyjątek, który nie został obsłużony wewnątrz wątku, to ten wyjątek będzie propagowany dalej. W efekcie w celu obsługi wyjątków operacji wykonywanych w tle należy postępować z funkcją `get()` w taki sam sposób, w jaki postąpilibyśmy, gdyby operacja była uruchomiona synchronicznie.

Na przykład spróbujmy uruchomić zadanie w tle z nieskończoną pętlą alokującą pamięć w celu wstawienia nowego elementu listy[5]:

```
// concurrency/async2.cpp

#include <future>
#include <list>
#include <iostream>
#include <exception>
using namespace std;

void task1()
{
    // nieskończone wstawianie elementu i alokacja pamięci
    // - prędzej czy później spowoduje zgłoszenie wyjątku
    // - UWAGA: to jest zła praktyka
    list<int> v;
    while (true) {
        for (int i=0; i<1000000; ++i) {
            v.push_back(i);
```

[4] Dziękujemy Lawrence'owi Crowlowi za tę uwagę oraz za dostarczenie przykładu.

[5] Próba zużywania pamięci do momentu, aż wystąpi wyjątek, jest oczywiście złą praktyką. W środowiskach niektórych systemów operacyjnych może to spowodować problemy. W związku z tym uruchamiając ten przykład, należy zachować ostrożność.

```
            }
            cout.put('.').flush();
        }
    }
    int main()
    {
        cout << "uruchomienie 2 zadań" << endl;
        cout << "- task1: przetwarzanie nieskończonej pętli zużywającej pamięć" << endl;
        cout << "- task2: oczekiwanie na zwrócenie sterowania, a następnie na zakończenie
        zadania task1" << endl;

        auto f1 = async(task1); // uruchomienie zadania task1() asynchronicznie (teraz, później lub nigdy)

        cin.get(); // czytanie znaku (podobnie jak getchar())

        cout << "\noczekiwanie na zakończenie zadania task1: " << endl;
        try {
            f1.get(); // oczekiwanie na zakończenie zadania task1() (lub zgłoszenie wyjątku)
        }
        catch (const exception& e) {
            cerr << "WYJĄTEK: " << e.what() << endl;
        }
    }
```

Prędzej czy później nieskończona pętla spowoduje zgłoszenie wyjątku (prawdopodobnie będzie to wyjątek `bad_alloc` — patrz punkt 4.3.1). Ten wyjątek spowoduje zakończenie wątku, ponieważ nie jest nigdzie przechwycony. Ten stan będzie zapisany w obiekcie futury do czasu wywołania funkcji `get()`. Wywołanie funkcji `get()` spowoduje dalszą propagację wyjątku wewnątrz funkcji `main()`.

Możemy teraz podsumować interfejs funkcji `async()` i obiektów futur w następujący sposób: funkcja `async()` daje środowisku programistycznemu szansę na równoległe uruchomienie obliczeń, których wyniki będą wykorzystane później (w momencie wywołania funkcji `get()`). Inaczej mówiąc, jeśli mamy pewną niezależną funkcjonalność `f`, to możemy skorzystać z równoległego przetwarzania, jeśli jest ono możliwe, poprzez przekazanie `f` do funkcji `async()` w chwili, gdy mamy wszystko, co jest potrzebne do uruchomienia tej funkcjonalności. Następnie w miejscu, gdzie jest potrzebny wynik funkcjonalności `f`, należy wstawić wywołanie funkcji `get()` w odniesieniu do futury zwróconej przez funkcję `async()`. W ten sposób uzyskujemy ten sam wynik, ale mamy szansę na lepszą wydajność, ponieważ funkcjonalność `f` może działać współbieżnie, zanim będzie potrzebny jej wynik.

Oczekiwanie i odpytywanie

Funkcję `get()` w odniesieniu do obiektu `future<>` można wywołać tylko raz. Po wywołaniu funkcji `get()` futura jest nieważna. Można to sprawdzić jedynie poprzez wywołanie funkcji `valid()` w odniesieniu do futury. Każde wywołanie inne niż niszczące obiekt spowoduje niezdefiniowane zachowanie (szczegółowe informacje można znaleźć w punkcie 18.3.2).

Futury zapewniają również interfejs pozwalający na oczekiwanie na zakończenie operacji działającej w tle bez przetwarzania jej wyniku. Interfejs ten może być wywoływany więcej niż jeden raz. Aby ograniczyć czas oczekiwania, można do niego przekazać czas trwania lub punkt w czasie.

Samo wywołanie funkcji `wait()` wymusza uruchomienie wątku reprezentowanego przez futurę i oczekiwanie na zakończenie operacji w tle:

```
std::future<...> f(std::async(func)); // próba asynchronicznego wywołania funkcji
...
f.wait();       // oczekiwanie na zakończenie funkcji (może uruchomić zadanie w tle)
```

Istnieją dwie inne odmiany funkcji `wait()`, które można wywoływać w odniesieniu do futur. Funkcje te *nie* wymuszają uruchomienia wątku, jeśli nie został on uruchomiony wcześniej:

1. Funkcja `wait_for()`, do której przekazujemy czas oczekiwania, pozwala czekać na asynchroniczne uruchomienie operacji przez ograniczony czas:

   ```
   std::future<...> f(std::async(func)); // próba asynchronicznego wywołania funkcji
   ...
   f.wait_for(std::chrono::seconds(10)); // oczekiwanie na funkcję func przez co najwyżej 10 sekund
   ```

2. Funkcja `wait_until()` pozwala czekać do określonego punktu w czasie:

   ```
   std::future<...> f(std::async(func)); // próba asynchronicznego wywołania funkcji
   ...
   f.wait_until(std::system_clock::now()+std::chrono::minutes(1));
   ```

Zarówno funkcja `wait_for()`, jak i `wait_until()` zwracają jeden z poniższych wyników:

- `std::future_status::deferred` — jeśli funkcja `async()` odroczyła operację i żadne z wywołań `wait()` lub `get()` jeszcze nie wymusiło jej startu (w tym przypadku obie funkcje natychmiast zwracają sterowanie);
- `std::future_status::timeout` — jeśli operacja została uruchomiona asynchronicznie, ale jeszcze się nie zakończyła (jeśli oczekiwanie zakończyło się ze względu na upływ przekazanego limitu czasu);
- `std::future_status::ready` — jeśli operacja zakończyła się.

Wykorzystanie funkcji `wait_for()` lub `wait_until()` umożliwia tzw. *spekulacyjne uruchomienie* programu. Na przykład rozważmy scenariusz, w którym musimy uzyskać wynik obliczeń w określonym czasie i chcielibyśmy mieć dokładny wynik[6]:

```
int quickComputation();     // przetwarzanie wyniku "szybkie i niedokładne"
int accurateComputation(); // przetwarzanie wyniku "dokładne, ale wolne"

std::future<int> f; //zadeklarowane na zewnątrz ze względu na to, że czas życia funkcji
accurateComputation()
                    // może przekroczyć czas życia funkcji bestResultInTime()
int bestResultInTime()
{
    // zdefiniowanie przedziału czasowego do uzyskania odpowiedzi:
    auto tp = std::chrono::system_clock::now() + std::chrono::minutes(1);
```

[6] Dziękujemy Lawrence'owi Crowlowi za te informacje oraz za dostarczenie przykładu.

```
    // uruchomienie obliczeń zarówno dokładnych, jak i zgrubnych:
    f = std::async (std::launch::async, accurateComputation);
    int guess = quickComputation();

    // przekazanie dokładnym obliczeniom pozostałej części przedziału czasowego:
    std::future_status s = f.wait_until(tp);

    // zwrócenie najlepszego wyniku obliczeń, jaki posiadamy:
    if (s == std::future_status::ready) {
        return f.get();
    }
    else {
        return guess; // funkcja accurateComputation() kontynuuje działanie
    }
}
```

Zwróćmy uwagę, że futura `f` nie może być lokalnym obiektem zadeklarowanym wewnątrz funkcji `bestResultInTime()`, ponieważ jeśli czas na zakończenie funkcji `accurateComputation()` będzie za krótki, to destruktor futury zablokuje się do czasu zakończenia zadania wykonywanego asynchronicznie.

Poprzez przekazanie czasu trwania równego zero lub punktu w czasie, który minął, możemy „odpytać", czy zadanie w tle uruchomiło się oraz czy jeszcze działa:

```
future<...> f(async(task)); // próba wywołania zadania asynchronicznie
...
// wykonanie czegoś w czasie, gdy zadanie jeszcze się nie zakończyło (co może nigdy się nie zdarzyć!)
while (f.wait_for(chrono::seconds(0) != future_status::ready)) {
    ...
}
```

Zwróćmy jednak uwagę, że taka pętla może się nigdy nie zakończyć, ponieważ na przykład w środowiskach jednowątkowych wywołanie zostanie odroczone do czasu wywołania funkcji `get()`. W związku z tym powinniśmy wywołać funkcję `async()`, przekazując strategię uruchamiania `std::launch::async`, lub jawnie sprawdzić, czy funkcja `wait_for()` zwróciła wartość `std::future_status::deferred`:

```
future<...> f(async(task)); // próba wywołania zadania asynchronicznie
...
// sprawdzenie, czy zadanie zostało odroczone:
if (f.wait_for(chrono::seconds(0)) != future_status::deferred) {
    // wykonanie czegoś w czasie, gdy zadanie jeszcze się nie zakończyło
    while (f.wait_for(chrono::seconds(0) != future_status::ready)) {
        ...
    }
}
...
auto r = f.get(); // wymuszenie uruchomienia zadania i oczekiwanie na wynik (lub wyjątek)
```

Innym powodem nieskończonej pętli w tym przypadku może być sytuacja, że wątek, który wykonuje pętlę, zajmuje cały czas procesora, a inne wątki nie otrzymują czasu potrzebnego do tego, by futura była gotowa. Taka sytuacja może doprowadzić do znacznego spadku szybkości działania programu. Najszybszym rozwiązaniem jest wywołanie funkcji `yield()` (patrz punkt 18.3.7) wewnątrz pętli:

```
std::this_thread::yield(); // wskazówka przekazania sterowania do następnego wątku
```

i (lub) uśpienie wątku na krótki czas.

Szczegółowe informacje na temat czasów trwania i punktów czasu, które można przekazać jako argumenty do funkcji `wait_for()` i `wait_until()`, można znaleźć w podrozdziale 5.7. Zwróćmy uwagę, że na działanie funkcji `wait_for()` i `wait_until()` mają wpływ korekty czasu systemowego (szczegółowe informacje można znaleźć w punkcie 5.7.5).

18.1.2. Przykład oczekiwania na dwa zadania

Trzeci program pokazuje kilka możliwości, które wymieniliśmy przed chwilą:

```
// concurrency/async3.cpp

#include <future>
#include <thread>
#include <chrono>
#include <random>
#include <iostream>
#include <exception>
using namespace std;
void doSomething (char c)
{
    // generator liczb losowych (wykorzystuje c jako ziarno do uzyskania różnych sekwencji)
    default_random_engine dre(c);
    uniform_int_distribution<int> id(10,1000);

    // pętla wyświetlająca znak po upływie losowego czasu
    for (int i=0; i<10; ++i) {
        this_thread::sleep_for(chrono::milliseconds(id(dre)));
        cout.put(c).flush();
    }
}

int main()
{
    cout << "uruchomienie 2 zadań asynchronicznie" << endl;

    // uruchomienie dwóch pętli w tle wyświetlających znaki . lub +.
    auto f1 = async([]{ doSomething('.'); });
    auto f2 = async([]{ doSomething('+'); });

    // jeśli działa co najmniej jedno zadanie realizowane w tle
    if (f1.wait_for(chrono::seconds(0)) != future_status::deferred ||
        f2.wait_for(chrono::seconds(0)) != future_status::deferred) {
        // odpytywanie do czasu zakończenia co najmniej jednej pętli
        while (f1.wait_for(chrono::seconds(0)) != future_status::ready &&
               f2.wait_for(chrono::seconds(0)) != future_status::ready) {
            ...;
            this_thread::yield(); // wskazówka przekazania sterowania do następnego wątku
        }
    }
    cout.put('\n').flush();

    // oczekiwanie na zakończenie wszystkich pętli i przetwarzanie wyjątków
    try {
```

```
        f1.get();
        f2.get();
    }
    catch (const exception& e) {
        cout << "\nWYJĄTEK: " << e.what() << endl;
    }
    cout << "\nzrobione" << endl;
}
```

Tak jak w poprzednim przykładzie mamy operację `doSomething()`, która od czasu do czasu wyświetla znak przekazany w roli argumentu (patrz punkt 18.1.1).

Teraz za pomocą funkcji `async()`dwukrotnie uruchamiamy w tle operację `doSomething()`, wyświetlając dwa różne znaki z wykorzystaniem dwóch różnych opóźnień generowanych przez odpowiednie sekwencje liczb losowych:

```
auto f1 = std::async([]{ doSomething('.'); });
auto f2 = std::async([]{ doSomething('+'); });
```

W środowiskach wielowątkowych oznacza to, że w tle będą działały dwie operacje, które „od czasu do czasu" wyświetlą różne znaki.

Następnie „odpytujemy", czy jedna z dwóch operacji zakończyła się[7]:

```
while (f1.wait_for(chrono::seconds(0)) != future_status::ready &&
       f2.wait_for(chrono::seconds(0)) != future_status::ready) {
    ...
    this_thread::yield(); // wskazówka przekazania sterowania do następnego wątku
}
```

Ponieważ jednak ta pętla nigdy by się nie skończyła, gdyby żadne z zadań nie zostało uruchomione w tle, w momencie wywołania funkcji `async()` najpierw musimy sprawdzić, czy co najmniej jedna z operacji nie została odroczona:

```
if (f1.wait_for(chrono::seconds(0)) != future_status::deferred ||
    f2.wait_for(chrono::seconds(0)) != future_status::deferred) {
        ...
}
```

Alternatywnie moglibyśmy wywołać funkcję `async()`, przekazując do niej strategię uruchamiania `std::launch:async`.

Jeśli co najmniej jedna operacja w tle zakończyła się lub żadna z nich się nie rozpoczęła, wyświetlamy znak przejścia do nowego wiersza, a następnie oczekujemy na zakończenie obu pętli:

```
f1.get();
f2.get();
```

W tym przykładzie używamy funkcji `get()` do przetwarzania wyjątków, które mogły wystąpić.

W środowisku wielowątkowym program może mieć na przykład następujący wynik:

[7] Jeśli w pętli nie będziemy wykonywać żadnych użytecznych operacji, będzie to po prostu aktywne oczekiwanie. Oznacza to, że problem lepiej rozwiązać za pomocą zmiennych warunkowych (patrz punkt 18.6.1).

```
uruchomienie 2 zadań asynchronicznie
++.++..+.+..++.+.+
..
zrobione
```

Zwróćmy uwagę, że nie mamy gwarancji co do kolejności trzech znaków `.`, `+` i znaku przejścia do nowego wiersza. Może się zdarzyć, że pierwszym znakiem będzie kropka, ponieważ jest to wynik pierwszej uruchomionej operacji — zatem uruchomiono ją nieco wcześniej — ale jak można zauważyć, znak `+` także może być wyświetlony jako pierwszy. Znaki `.` i `+` mogą wyświetlać się na przemian, ale to także nie jest gwarantowane. Jeśli usuniemy instrukcję `sleep_for()`, która wymusza opóźnienie pomiędzy wyświetlaniem przekazanych znaków, pierwsza pętla zakończy się przed pierwszym przełączeniem kontekstu do innego wątku, zatem wynik może mieć także następującą postać:

```
uruchomienie 2 zadań asynchronicznie
..........
++++++++++
zrobione
```

Taki wynik powstanie również w środowiskach, które nie obsługują wielowątkowości, ponieważ w tym przypadku oba wywołania funkcji `doSomething()` będą wykonane synchronicznie z wywołaniami funkcji `get()`.

Nie wiadomo również dokładnie, kiedy będzie wyświetlony znak przejścia do nowego wiersza. Może się to zdarzyć przed wyświetleniem jakiegokolwiek innego znaku. Stanie się tak w przypadku, gdy odroczono wykonanie obu zadań w tle do czasu wywołania funkcji `get()`. W takim przypadku odroczone zadania będą wywołane jedno po drugim:

```
uruchomienie 2 zadań asynchronicznie
..........++++++++++
zrobione
```

W tym przypadku wiemy jedynie, że znak przejścia do nowego wiersza nie zostanie wyświetlony przed zakończeniem jednej z pętli. Nie ma nawet gwarancji tego, że znak przejścia do nowego wiersza wyświetli się bezpośrednio za ostatnim znakiem w jednej z sekwencji, ponieważ może zająć trochę czasu, zanim zakończenie jednej z pętli zostanie zarejestrowane w odpowiednim obiekcie futury i ten zarejestrowany stan zostanie obliczony (zwróćmy uwagę, że to nie jest przetwarzanie w czasie rzeczywistym). Z tego powodu program może wyświetlić wynik, w którym kilka znaków `+` będzie zapisanych za ostatnią kropką, a przed znakiem przejścia do nowego wiersza:

```
uruchomienie 2 zadań asynchronicznie
.+..+..+..+.+..++
+++
zrobione
```

Przekazywanie argumentów

W poprzednim przykładzie zademonstrowano jeden sposób przekazywania argumentów do zadania uruchomionego w tle: wykorzystujemy w tym celu wyrażenie lambda (patrz punkt 3.1.10), które wywołuje funkcjonalność uruchomioną w tle:

```
auto f1 = std::async([]{ doSomething('.'); });
```

Oczywiście możemy także przekazać argumenty, które istniały przed wywołaniem instrukcji `async()`. Tak jak zwykle argumenty można przekazać przez wartość lub przez referencję:

```
char c = '@';
auto f = std::async([=]{ // znak =: może uzyskać dostęp do obiektów w zakresie przez wartość
                         doSomething(c); // przekazanie kopii znaku c do funkcji doSomething()
                       });
```

Poprzez zdefiniowanie argumentu w postaci wyrażenia `[=]` możemy przekazać *kopię* znaku `c` oraz wszystkich innych widocznych obiektów do funkcji lambda. Dzięki temu wewnątrz wyrażenia lambda możemy przekazać tę kopię znaku `c` do funkcji `doSomething()`.

Istnieją jednak inne sposoby przekazywania argumentów do funkcji `async()`, ponieważ funkcja `async()` dostarcza standardowego interfejsu *obiektów wywoływalnych* (patrz podrozdział 4.4). Na przykład: jeśli przekażemy wskaźnik funkcji jako pierwszy argument do funkcji `async()`, możemy przekazać wiele dodatkowych argumentów. Są one przekazywane jako parametry do wywoływanej funkcji:

```
char c = '@';
auto f = std::async(doSomething,c); // asynchroniczne wywołanie funkcji doSomething(c)
```

Można również przekazywać argumenty przez referencję, ale w takim przypadku ponosimy ryzyko, że przekazane wartości stracą ważność przed rozpoczęciem zadania działającego w tle. Dotyczy to wywoływanych bezpośrednio zarówno wyrażeń lambda, jak i funkcji:

```
char c = '@';
auto f = std::async([&]{ doSomething(c); });  // ryzykowne!

char c = '@';
auto f = std::async(doSomething,std::ref(c)); // ryzykowne!
```

Możemy również zarządzać czasem życia przekazanego argumentu tak, by był on dłuższy od czasu realizacji zadania w tle. Na przykład:

```
void doSomething (const char& c); // przekazanie znaku przez referencję
...
char c = '@';
auto f = std::async([&]{ doSomething(c); }); // przekazanie znaku c przez referencję
...
f.get(); // znak c musi być osiągalny do tego miejsca
```

Należy jednak zachować ostrożność: jeśli przekazujemy argumenty przez referencję, aby móc je modyfikować z osobnego wątku, łatwo może dojść do sytuacji, że program zacznie działać w nieoczekiwany sposób. Rozważmy poniższy przykład,

gdzie po próbie uruchomienia pętli działającej w tle, która wyświetla znak, przełączyliśmy wyświetlany znak:

```
void doSomething (const char& c); // przekazanie znaku przez referencję
...
char c = '@';
auto f = std::async([&]{ doSomething(c); }); // przekazanie znaku c przez referencję
...
c = '_'; // przełączenie wyniku funkcji doSomething() na znak podkreślenia, o ile to zadanie nadal działa
f.get(); // znak c musi być osiągalny do tego miejsca
```

Po pierwsze, kolejność dostępu do znaku `c` w powyższym kodzie i w funkcji `doSomething()` jest niezdefiniowana. Po drugie, przełączenie wyświetlanego znaku może nastąpić przed uruchomieniem pętli wyświetlającej wynik, w jej trakcie lub po wykonaniu tej pętli. Co gorsza, ponieważ modyfikujemy znak `c` w jednym wątku, a inny wątek odczytuje ten znak, mamy tu do czynienia z niezsynchronizowanym jednoczesnym dostępem do tego samego obiektu (tzw. *wyścig o dane* — patrz punkt 18.4.1). Efektem tej sytuacji może być nieoczekiwane działanie, o ile nie zabezpieczymy się przed jednoczesnym dostępem z wykorzystaniem muteksów (patrz podrozdział 18.5) lub zmiennych `atomic` (patrz podrozdział 18.7).

Zatem zapamiętajmy że: **jeśli używamy funkcji `async()`, powinniśmy przekazać wszystkie obiekty potrzebne do uruchomienia przekazanej funkcjonalności przez wartość, tak aby funkcja `async()` używała tylko kopii lokalnych**. Jeśli kopiowanie jest zbyt kosztowne, powinniśmy zapewnić przekazywanie wszystkich obiektów jako stałych referencji, tak aby obiekty mutowalne nie były używane. W każdym innym przypadku powinniśmy być świadomi implikacji naszego podejścia (można o nich przeczytać w podrozdziale 18.4).

Można także przekazać do funkcji `async()` wskaźnik do funkcji składowej. W takim przypadku pierwszym argumentem za funkcją składową powinna być referencja lub wskaźnik do obiektu, na rzecz którego została wywołana funkcja składowa:

```
class X
{
    public:
        void mem (int num);
        ...
};
...

X x;
auto a = std::async(&X::mem, x, 42); // próba asynchronicznego wywołania x.mem(42)
...
```

18.1.3. Współdzielone futury

Jak widzieliśmy, klasa `std::future` zapewnia możliwość przetwarzania wyniku futury dla współbieżnych obliczeń. Jednak ten wynik możemy przetwarzać tylko raz: drugie wywołanie funkcji `get()` spowoduje niezdefiniowane działanie (zgodnie

z wymaganiami standardowej biblioteki C++ zaleca się, aby implementacje zgłaszały wyjątek `std::future_error`, ale nie jest to obowiązkowe).

Czasami jednak sensowne jest przetwarzanie wyniku współbieżnych obliczeń więcej niż raz — zwłaszcza gdy wynik ten jest przetwarzany w wielu innych wątkach. Do tego celu standardowa biblioteka C++ dostarcza klasy `std::shared_future`. W przypadku jej użycia możliwe jest kilka wywołań funkcji `get()`. Za każdym razem zwracają one taki sam wynik bądź zgłaszają taki sam wyjątek.

Przeanalizujmy następujący przykład:

```
// concurrency/sharedfuture1.cpp

#include <future>
#include <thread>
#include <iostream>
#include <exception>
#include <stdexcept>
using namespace std;

int queryNumber ()
{
    // odczytanie liczby
    cout << "wprowadź liczbę: ";
    int num;
   cin >> num;

    // zgłoszenie wyjątku, jeśli nie wprowadzono liczby
    if (!cin) {
        throw runtime_error("nie wprowadzono liczby");
    }

    return num;
}

void doSomething (char c, shared_future<int> f)
{
    try {
        // oczekiwanie na wyświetlenie kilku znaków
        int num = f.get(); // pobranie wyniku funkcji queryNumber()

        for (int i=0; i<num; ++i) {
            this_thread::sleep_for(chrono::milliseconds(100));
            cout.put(c).flush();
        }
    }
    catch (const exception& e) {
        cerr << "WYJĄTEK w wątku " << this_thread::get_id()
                  << ": " << e.what() << endl;
    }
}

int main()
{
    try {
        // uruchomienie wątku w celu zapytania o liczbę
        shared_future<int> f = async(queryNumber);
```

```
        // uruchomienie trzech wątków — każdy przetwarza tę liczbę w pętli
        auto f1 = async(launch::async,doSomething,'.',f);
        auto f2 = async(launch::async,doSomething,'+',f);
        auto f3 = async(launch::async,doSomething,'*',f);

        // oczekiwanie na zakończenie wszystkich pętli
        f1.get();
        f2.get();
        f3.get();
    }
    catch (const exception& e) {
        cout << "\nWYJĄTEK: " << e.what() << endl;
    }
    cout << "\nzrobione" << endl;
}
```

W tym przykładzie jeden wątek wywołuje funkcję `queryNumber()`, która prosi o wprowadzenie liczby całkowitej. Następnie liczba ta jest wykorzystywana przez inne wątki, które zostały uruchomione wcześniej. W celu wykonania tego zadania wynik funkcji `std::async()`, która uruchamia wątek zapytania, jest przypisywany do obiektu `shared_future` wyspecjalizowanego do obsługi zwracanej wartości:

```
shared_future<int> f = async(queryNumber);
```

Zatem współdzielona futura może być zainicjowana przez zwykłą futurę, która przenosi stan od futury do współdzielonej futury. Aby można było użyć słowa kluczowego `auto` dla tej deklaracji, możemy alternatywnie skorzystać ze składowej funkcji `share()`:

```
auto f = async(queryNumber).share();
```

Wewnętrznie wszystkie obiekty współdzielonych futur korzystają ze *współdzielonego stanu*, który funkcja `async()` tworzy w celu przechowywania wyniku przekazanej funkcjonalności (oraz przechowywania samej funkcjonalności, jeśli jest ona odroczona).

Współdzielona funkcjonalność jest następnie przekazywana do innych wątków: funkcja `doSomething()` jest wykonywana ze współdzieloną futurą w roli drugiego argumentu:

```
auto f1 = async(launch::async,doSomething,'.',f);
auto f2 = async(launch::async,doSomething,'+',f);
auto f3 = async(launch::async,doSomething,'*',f);
```

Wewnątrz każdego wywołania funkcji `doSomething()` oczekujemy na przetwarzanie wyniku funkcji `queryNumber()` poprzez wywołanie funkcji `get()` w odniesieniu do przekazanej współdzielonej futury:

```
void doSomething (char c, shared_future<int> f)
{
    try {
        int num = f.get(); // pobranie wyniku funkcji queryNumber()
        ...
    }
    catch (const exception& e) {
```

```
            cerr << "WYJĄTEK w wątku " << this_thread::get_id()
                      << ": " << e.what() << endl;
        }
    }
```

Jeśli funkcja `queryNumber()` zgłosi wyjątek, co może się zdarzyć, jeśli nie można odczytać żadnej liczby całkowitej, wtedy każde wywołanie funkcji `doSomething()` spowoduje uzyskanie tego wyjątku za pośrednictwem wywołania `f.get()`. Dzięki temu zostanie wykonana odpowiednia procedura obsługi wyjątku.

Zatem po odczytaniu liczby 5 w roli danych wejściowych wynik działania programu może być następujący:

```
wprowadź liczbę: 5
*+.*+.*.+*+.*.+
zrobione
```

Jeśli jednak wpiszemy *x* jako dane wejściowe, program może mieć następujący wynik:

```
wprowadź liczbę: x
WYJĄTEK w wątku 3: nie wprowadzono liczby
WYJĄTEK w wątku 4: nie wprowadzono liczby
WYJĄTEK w wątku 2: nie wprowadzono liczby

zrobione
```

Zwróćmy uwagę, że kolejność wyjścia generowanego przez wątki i wartości identyfikatorów wątków są niezdefiniowane (szczegółowe informacje na temat identyfikatorów wątków można znaleźć w punkcie 18.2.1).

Zwróćmy także uwagę na niewielką różnicę w deklaracji funkcji `get()` pomiędzy futurą (`future`) a współdzieloną futurą (`shared_future`):

- Dla klasy `future<>` funkcja `get()` jest dostępna w następujący sposób (`T` jest typem zwracanej wartości):

  ```
  T future<T>::get();                    // ogólna postać funkcji get()
  T& future<T&>::get();                  // specjalna postać dla referencji
  void future<void>::get();              // specjalna postać dla danych void
  ```

  gdzie pierwsza postać zwraca przeniesiony wynik lub kopię wyniku.

- Dla klasy `shared_future<>` funkcja `get()` jest dostępna w następujący sposób:

  ```
  const T& shared_future<T>::get();      // ogólna postać funkcji get()
  T& shared_future<T&>::get();           // specjalna postać dla referencji
  void shared_future<void>::get();       // specjalna postać dla wartości void
  ```

  gdzie pierwsza postać zwraca referencję do wartości wyniku zapisanej w obiekcie *współdzielonego stanu*:

Zgodnie z tym co napisano w [*N3194:Futures*]:

„Funkcja `get()` z wartością do jednorazowego użytku jest zoptymalizowana pod kątem przenoszenia (np. `std::vector<int> v = f.get()`). Funkcja `get()` zwracająca stałą referencję jest zoptymalizowana pod kątem dostępu (np. `int i = f.get()[3]`)".

Taki projekt wprowadza ryzyko problemów związanych z czasem życia lub sytuacjami wyścigu w przypadku, gdy zwrócone wartości zostaną zmodyfikowane (szczegółowe informacje można znaleźć w punkcie 18.3.3).

Można również przekazać współdzieloną futurę przez referencję, tzn. zadeklarować ją jako referencję i skorzystać z wywołania `std::ref()`, aby ją przekazać:

```
void doSomething (char c, const shared_future<int>& f)
auto f1 = async(launch::async,doSomething,'.',std::ref(f));
```

Teraz zamiast wykorzystywać wiele współdzielonych obiektów futur, z których wszystkie współdzielą ten sam *współdzielony stan*, wykorzystamy jeden współdzielony obiekt futury w celu wykonania wielu funkcji `get()` (po jednej w każdym wątku). Jednak takie podejście jest bardziej ryzykowne. Programista musi zadbać o to, aby czas życia futury `f` (tak, futury `f`, a nie *współdzielonego stanu*, do którego się ona odnosi) nie był krótszy niż czas życia uruchomionych wątków. Ponadto zwróćmy uwagę, że funkcje składowe współdzielonych futur nie są ze sobą zsynchronizowane, natomiast *współdzielony stan* jest zsynchronizowany. Jeśli zatem potrzebujemy czegoś więcej niż tylko odczytywania danych, to w celu zapobieżenia sytuacjom *wyścigu o dane* (które spowodowałyby niezdefiniowane zachowanie) mogą nam być potrzebne zewnętrzne techniki synchronizacji (patrz podrozdział 18.4). Warto również zapamiętać zasadę, którą w prywatnej wiadomości sformułował Lawrence Crowl, jeden z autorów biblioteki obsługi współbieżności: „Jeśli kod pozostaje w ścisłej koordynacji ze sobą, przekazywanie obiektów przez referencję sprawdza się. Jeśli kod może propagować do obszarów, gdzie nie są w pełni rozumiane jego cele i ograniczenia, lepsze jest przekazywanie przez wartość. Kopiowanie współdzielonej futury jest kosztowne obliczeniowo, ale nie tak kosztowne, aby usprawiedliwiało ryzyko utajonego błędu w dużym systemie".

Więcej informacji dla temat klasy `shared_future` można znaleźć w punkcie 18.3.3.

18.2. Interfejs niskiego poziomu: wątki i promesy

Oprócz wysokopoziomowego interfejsu, którego bazą jest funkcja `async()` i (współdzielone) futury, standardowa biblioteka C++ dostarcza niskopoziomowego interfejsu do uruchamiania wątków i wykonywania na nich operacji.

18.2.1. Klasa std::thread

Aby uruchomić wątek, należy po prostu zadeklarować obiekt klasy `std::thread` i przekazać do niego w roli pierwszego argumentu zadanie do wykonania. Następnie należy zaczekać na zakończenie wątku albo go *odłączyć*:

```
void doSomething();

std::thread t(doSomething); // uruchomienie funkcji doSomething() w tle
...
t.join(); // oczekiwanie na zakończenie funkcji t (zablokowanie do zakończenia funkcji doSomething())
```

Tak jak w przypadku funkcji `async()` możemy przekazać cokolwiek, co jest *obiektem wywoływalnym* (funkcją, funkcją składową, obiektem funkcyjnym, wyrażeniem lambda — patrz podrozdział 4.4), razem z ewentualnymi dodatkowymi argumentami. Jednak chcielibyśmy podkreślić jeszcze raz, że o ile nie istnieją ku temu ważne powody, to wszystkie obiekty potrzebne do przetwarzania powinny być przekazywane *przez wartość*, tak aby wątek korzystał tylko z *lokalnych kopii* (opis niektórych problemów, które mogą wystąpić, jeśli nie zastosujemy się do tego zalecenia, można znaleźć w podrozdziale 18.4).

Trzeba także pamiętać, że to jest interfejs niskopoziomowy. Interesujące jest zatem to, czego ten interfejs *nie* zapewnia w porównaniu z funkcją `async()` (patrz podrozdział 18.1):

- Klasa `thread` nie obsługuje strategii uruchamiania. Standardowa biblioteka C++ zawsze próbuje uruchomić przekazaną funkcjonalność w nowym wątku. Jeśli to nie jest możliwe, zgłasza wyjątek `std::system_error` (patrz punkt 4.3.1) z kodem błędu `resource_unavailable_try_again` (patrz punkt 4.3.2).
- Nie istnieje interfejs do przetwarzania wyniku działania wątku. Jedyne, co możemy uzyskać, to unikatowy identyfikator wątku (patrz punkt 18.2.1).
- Jeśli wystąpi wyjątek, który nie zostanie przechwycony wewnątrz wątku, program natychmiast zakończy działanie, wywołując funkcję `std::terminate()` (patrz punkt 5.8.2). W celu przekazania wyjątków do kontekstu poza wątkiem trzeba skorzystać z obiektów `exception_ptr` (patrz punkt 4.3.3).
- W wątku wywołującym trzeba zadeklarować, czy chcemy czekać na zakończenie wątku (poprzez wywołanie funkcji `join()`), czy też chcemy *odłączyć* go od uruchomionego wątku, aby pozwolić mu na działanie w tle bez żadnej kontroli (poprzez wywołanie funkcji `detach()`). Jeśli tego nie zrobimy przed zakończeniem czasu życia obiektu wątku lub jeśli nastąpi przypisanie przenoszące, program zakończy działanie poprzez wywołanie `std::terminate()` (patrz punkt 5.8.2).
- Jeśli pozwolimy na działanie wątku w tle, a funkcja `main()` się zakończy, wszystkie wątki zostaną gwałtownie zakończone.

Oto pierwszy kompletny przykład:

```
// concurrency/thread1.cpp

#include <thread>
#include <chrono>
#include <random>
#include <iostream>
#include <exception>
using namespace std;

void doSomething (int num, char c)
{
    try {
        // generator liczb losowych (wykorzystuje c jako ziarno do uzyskania różnych sekwencji)
        default_random_engine dre(42*c);
        uniform_int_distribution<int> id(10,1000);
        for (int i=0; i<num; ++i) {
            this_thread::sleep_for(chrono::milliseconds(id(dre)));
```

```
            cout.put(c).flush();
            ...
        }
    }
    // upewniamy się, czy żaden wyjątek nie opuścił wątku, co spowodowałoby zakończenie
    // działania programu
    catch (const exception& e) {
        cerr << "WYJĄTEK WĄTKU (wątek "
             << this_thread::get_id() << "): " << e.what() << endl;
    }
    catch (...) {
        cerr << "WYJĄTEK WĄTKU (wątek "
             << this_thread::get_id() << ")" << endl;
    }
}

int main()
{
    try {
        thread t1(doSomething,5,'.'); // wyświetlenie pięciu kropek w osobnym wątku
        cout << "- uruchomiono wątek pp" << t1.get_id() << endl;

        // wyświetlenie innych znaków w wątkach działających w tle
        for (int i=0; i<5; ++i) {
            thread t(doSomething,10,'a'+i); // wyświetlenie 10 znaków w osobnym wątku
            cout << "- odłączenie uruchomionego wątku dp" << t.get_id() << endl;
            t.detach(); // odłączenie wątku do drugiego planu
    }

        cin.get(); // oczekiwanie na wejście (zwrócenie sterowania)
        cout << "- podłączenie wątku pp" << t1.get_id() << endl;
        t1.join(); // oczekiwanie na zakończenie wątku t1
    }
    catch (const exception& e) {
       cerr << "WYJĄTEK: " << e.what() << endl;
    }
}
```

W tym przykładzie w funkcji `main()` uruchamiamy kilka wątków, które wykonują instrukcje w funkcji `doSomething()`. Zarówno w funkcji `main()`, jak i w `doSomething()` wstawiliśmy odpowiednie klauzule `try-catch` z następujących powodów:

- W funkcji `main()` utworzenie wątku może spowodować zgłoszenie wyjątku `std::system_error` (patrz punkt 4.3.1) z kodem błędu `resource_unavailable_try_again`.
- W funkcji `doSomething()` uruchomionej jako `std::thread` wszelkie nieprzechwycone wyjątki mogą spowodować zakończenie działania programu.

W przypadku pierwszego wątku uruchomionego w funkcji `main()` później czekamy na jego zakończenie:

```
thread t1(doSomething,5,'.'); // wyświetlenie pięciu kropek w osobnym wątku
...
t1.join();                    // oczekiwanie na zakończenie wątku t1
```

Inne wątki są odłączane po uruchomieniu, dlatego mogą nadal działać w momencie zakończenia funkcji `main()`:

```
for (int i=0; i<5; ++i) {
    thread t(doSomething,10,'a'+i);     // wyświetlenie 10 znaków w osobnym wątku
    t.detach();                         // odłączenie wątku do drugiego planu
}
```

W konsekwencji program natychmiast zakończyłby wszystkie wątki w tle po zakończeniu funkcji `main()`. Mogłoby to nastąpić w konsekwencji próby odczytania danych wejściowych ze względu na instrukcję `cin.get()` lub próby wyświetlenia piątej kropki przez wątek, który wykonuje instrukcję `doSomething(5,'.')`, ze względu na instrukcję `t1.join()`. Ponieważ oczekiwanie na wejście oraz wyświetlanie kropek odbywa się współbieżnie, nie ma znaczenia, co się stanie najpierw.

Gdybym na przykład wcisnął *Return* po wyświetleniu drugiej kropki, program mógłby wyświetlić następujący wynik:

```
uruchomiono wątek pp 1
- odłączenie uruchomionego wątku dp 2
- odłączenie uruchomionego wątku dp 3
- odłączenie uruchomionego wątku dp 4
- odłączenie uruchomionego wątku dp 5
- odłączenie uruchomionego wątku dp 6
ecad.dbcebabd.a
- podłączenie wątku pp 1
b.ceade.bbcadbe.
```

Uwaga na odłączone wątki

Odłączone wątki mogą łatwo stać się problemem, jeśli wykorzystują zasoby nielokalne. Problem polega na tym, że tracimy kontrolę nad odłączonym wątkiem i nie ma łatwego sposobu, aby dowiedzieć się, czy i jak długo to trwa. Dlatego powinniśmy zadbać, aby odłączony wątek nie korzystał z żadnych obiektów po zakończeniu ich życia. Z tego powodu przekazywanie do wątku zmiennych i obiektów przez referencję zawsze niesie ryzyko. Zalecane jest przekazywanie argumentów przez wartość.

Należy jednak pamiętać, że problem czasu życia dotyczy także obiektów globalnych i statycznych, ponieważ po zakończeniu działania programu odłączony wątek może nadal działać, co oznacza, że może on uzyskać dostęp do globalnych lub statycznych obiektów, które już zostały zniszczone lub są w trakcie niszczenia. Niestety, takie działanie spowodowałoby niezdefiniowane zachowanie[8].

Z tego względu ogólną zasadą postępowania z odłączonymi wątkami powinno być uwzględnienie następujących reguł:

- Odłączone wątki powinny preferować dostęp wyłącznie do kopii lokalnych.
- Jeśli odłączony wątek korzysta z obiektu globalnego albo statycznego, powinniśmy wykonać jedną z następujących czynności:
  - Zadbać o to, aby te globalne (statyczne) obiekty nie zostały zniszczone, zanim wszystkie odłączone wątki, które z nich korzystają, nie zakończą działania (lub zakończą korzystanie z tych globalnych bądź statycznych

[8] Dziękujemy Hansowi Boehmowi i Anthony'emu Williamsowi za zwrócenie uwagi na ten problem.

obiektów). Jednym ze sposobów rozwiązania tego problemu jest wykorzystanie zmiennych warunkowych (patrz podrozdział 18.6), które są wykorzystywane przez odłączone wątki do sygnalizowania o swoim zakończeniu. Przed opuszczeniem funkcji `main()` lub wywołaniem instrukcji `exit()` należy ustawić te zmienne warunkowe, a następnie zasygnalizować, że jest możliwe zniszczenie[9].

- Zakończyć program poprzez wywołanie `quick_exit()`, wprowadzonej dokładnie w tym celu, aby zakończyć program bez wywoływania destruktorów globalnych i statycznych obiektów (patrz punkt 5.8.2).

Ponieważ obiekty `std::cin`, `std::cout` oraz `std::cerr`, a także inne obiekty strumieni globalnych (patrz punkt 15.2.2) zgodnie ze standardem „nie są niszczone podczas wykonywania programu", dostęp do tych obiektów w odłączonych wątkach nie powinien powodować niezdefiniowanego działania. Mogą jednak pojawić się inne problemy, na przykład przeplatanie znaków.

Niemniej jednak należy zapamiętać regułę, że jedynym bezpiecznym sposobem zakończenia odłączonego wątku jest skorzystanie z jednej z funkcji „`...at_thread_exit()`", które zmuszają główny wątek do oczekiwania na faktyczne zakończenie odłączonych wątków. Możemy również zignorować korzystanie z odłączonych wątków, zgodnie z opinią recenzenta, który napisał: „Temat odłączonych wątków to jedno z zagadnień, które należałoby przenieść do rozdziału poświęconego niebezpiecznym własnościom, które prawie nikomu nie są potrzebne".

Identyfikatory wątków

Jak można zauważyć, program wyświetla identyfikatory wątków dostarczone albo przez obiekt wątku, albo za pośrednictwem przestrzeni nazw `this_thread` (dostępnej również przez plik nagłówkowy `<thread>`):

```
void doSomething (int num, char c)
{
    ...
    cerr << "WYJĄTEK WĄTKU (wątek "
         << this_thread::get_id() << ")" << endl;
    ...
}

thread t(doSomething,5,'.'); // wyświetlenie pięciu kropek w osobnym wątku
cout << "- uruchomiono wątek pp" << t1.get_id() << endl;
```

Ten identyfikator jest specjalnym typem obiektu `std::thread::id`, który daje gwarancję niepowtarzalności dla każdego wątku. Dodatkowo klasa `id` ma domyślny konstruktor, który zwraca unikatowy ID reprezentujący „brak wątku":

```
std::cout << "ID wątku \"brak wątku\": " << std::thread::id()
          << std::endl;
```

[9] W idealnej sytuacji należy skorzystać z instrukcji `notify_all_at_thread_exit()` (patrz punkt 18.6.4) w celu ustawienia zmiennej warunkowej, aby zapewnić zniszczenie wszystkich zmiennych lokalnych wątku.

Jedynymi operacjami, jakie są dozwolone w odniesieniu do identyfikatorów wątków, są porównania oraz wywołanie operatora wyjścia dla strumienia. Nie należy robić żadnych dodatkowych założeń w rodzaju: wątek „brak wątku" ma ID 0 lub główny wątek ma ID 1. W rzeczywistości w konkretnych implementacjach identyfikatory te mogą być generowane „w locie", na żądanie, a nie przy uruchomieniu wątku, dlatego numer głównego wątku zależy od liczby żądań o identyfikatory wątku, które nastąpiły wcześniej. Zatem poniższy kod:

```
std::thread t1(doSomething,5,'.');
std::thread t2(doSomething,5,'+');
std::thread t3(doSomething,5,'*');
std::cout << "ID t3:       " << t3.get_id() << std::endl;
std::cout << "ID funkcji main:     " << std::this_thread::get_id() << std::endl;
std::cout << "ID wątku 'brak wątku': " << std::thread::id() << std::endl;
```

może wygenerować następujący wynik:

```
ID t3:       1
ID funkcji main:     4
ID wątku 'brak wątku': 0
```

lub:

```
ID t3:       3
ID funkcji main:     4
ID wątku 'brak wątku': 0
```

lub:

```
ID t3:       1
ID funkcji main:     2
ID wątku 'brak wątku': 3
```

Identyfikatorami wątków mogą być też znaki.

Z tego powodu jedynym sposobem identyfikacji wątku — na przykład wątku głównego — jest porównanie go z zapisanym identyfikatorem w momencie uruchomienia wątku:

```
std::thread::id masterThreadID;
void doSomething()
{
    if (std::this_thread::get_id() == masterThreadID) {
        ...
    }
    ...
}

std::thread master(doSomething);
masterThreadID = master.get_id();
...
std::thread slave(doSomething);
...
```

Zwróćmy uwagę, że identyfikatory zakończonych wątków mogą być wykorzystane ponownie.

Więcej informacji dla temat klasy `thread` można znaleźć w punkcie 18.3.6.

18.2.2. Promesy

Możemy teraz zadać sobie pytanie o to, w jaki sposób można przekazywać parametry i obsługiwać wyjątki pomiędzy wątkami (wyjaśnia to również, w jaki sposób są zaimplementowane interfejsy wysokopoziomowe, takie jak `async()`). Oczywiście aby przekazać dane do wątku, możemy po prostu przekazać je jako argumenty. A jeśli są nam potrzebne wyniki, możemy przekazać *argumenty wyjściowe* przez referencję, tak jak opisano dla funkcji `async()` (patrz punkt 18.1.2).

Istnieje jednak inny mechanizm ogólnego przeznaczenia, który pozwala przekazywać wartości wyników i wyjątki jako wynik działania wątków: klasa `std::promise`. Obiekt *promesy* jest odpowiednikiem obiektu *futury*. Oba pozwalają na tymczasowe przechowywanie *współdzielonego stanu*, który reprezentuje wartość (wynik) lub wyjątek. O ile obiekt futury pozwala na odczytywanie danych (za pomocą funkcji `get()`), obiekt promesy umożliwia dostarczanie danych (za pomocą jednej z jego składowych `set_...()`). Pokazano to w poniższym przykładzie:

```
// concurrency/promise1.cpp

#include <thread>
#include <future>
#include <iostream>
#include <string>
#include <exception>
#include <stdexcept>
#include <functional>
#include <utility>

void doSomething (std::promise<std::string>& p)
{
    try {
        // odczytanie znaku i zgłoszenie wyjątku, jeśli jest to znak 'x'
        std::cout << "wprowadź znak ('x', aby zgłosić wyjątek): ";
        char c = std::cin.get();
        if (c == 'x') {
            throw std::runtime_error(std::string("wczytano znak ")+c);
      }
    ...
    std::string s = std::string("przetworzono znak ") + c;
    p.set_value(std::move(s)); // zapamiętanie wyniku
    }
    catch (...) {
        p.set_exception(std::current_exception()); // zapamiętanie wyjątku
    }
}

int main()
{
    try {
        // uruchomienie wątku z wykorzystaniem promesy do przechowania wyniku
        std::promise<std::string> p;
        std::thread t(doSomething,std::ref(p));
        t.detach();
    ...
```

```
    // utworzenie obiektu futury w celu przetwarzania wyniku
    std::future<std::string> f(p.get_future());

    // przetwarzanie wyniku
    std::cout << "wynik: " << f.get() << std::endl;
}
catch (const std::exception& e) {
    std::cerr << "WYJĄTEK: " << e.what() << std::endl;
    }
    catch (...) {
       std::cerr << "WYJĄTEK " << std::endl;
    }
}
```

Po włączeniu pliku nagłówkowego `<future>`, gdzie znajdują się także deklaracje dotyczące promes, możemy zadeklarować obiekt promesy wyspecjalizowany do wartości, która ma być w nim przechowywana lub zwrócona (albo do typu `void` w przypadku braku takiej wartości):

```
std::promise<std::string> p; // przechowuje wynik w postaci łańcucha znaków lub wyjątku
```

Promesa wewnętrznie tworzy *współdzielony stan* (patrz podrozdział 18.3), który można tu wykorzystać do przechowania wartości odpowiedniego typu lub wyjątku. Można go również wykorzystać w obiekcie futury do odczytania danych jako wyniku działania wątku.

Promesa ta jest następnie przekazywana do zadania działającego jako osobny wątek:

```
std::thread t(doSomething,std::ref(p));
```

Poprzez użycie wywołania `std::ref()` (patrz punkt 5.4.3) zapewniamy przekazywanie promesy przez referencję. Dzięki temu możemy manipulować jej stanem (kopiowanie dla promes nie jest możliwe).

Następnie wewnątrz wątku możemy zapisać wartość lub wyjątek poprzez wywołanie odpowiednio funkcji `set_value()` lub `set_exception()`:

```
void doSomething (std::promise<std::string>& p)
{
    try {
        ...
        p.set_value(std::move(s)); // zapamiętanie wyniku
    }
    catch (...) {
        p.set_exception(std::current_exception()); // zapamiętanie wyjątku
    }
}
```

Do zapamiętania wyniku służy funkcja pomocnicza `std::current_exception()`, zdefiniowana w module `<exception>` (patrz punkt 4.3.3). Funkcja zwraca obsługiwany wyjątek jako typ `std::exception_ptr` lub `nullptr`, jeśli w danym momencie nie obsługujemy wyjątku. Obiekt promesy przechowuje ten wyjątek wewnątrz siebie.

Wartość lub wyjątek zapisane w obiekcie *współdzielonego stanu* stają się *gotowe* natychmiast po zapisaniu. Z tego powodu możemy odczytać ich wartość w innym miejscu. Jednak by móc ją odczytać, potrzebujemy obiektu futury, który dzieli ten

sam *współdzielony stan*. W tym celu wewnątrz funkcji `main()`wywołujemy metodę `get_future()` obiektu promesy i w ten sposób tworzymy obiekt futury. Obiekt ten charakteryzuje się standardową semantyką zaprezentowaną w podrozdziale 18.1. Mogliśmy również stworzyć obiekt futury przed uruchomieniem wątku:

```
std::future<std::string> f(p.get_future());
```

Następnie za pomocą funkcji `get()` odczytujemy zapisany wynik albo zapisany wyjątek jest ponownie zgłaszany (poprzez wywołanie funkcji `std::rethrow_exception()` w odniesieniu do zapisanego obiektu `exception_ptr`):

```
f.get(); // przetwarzanie wyniku wątku
```

Zwróćmy uwagę, że funkcja `get()` blokuje się do czasu, aż *współdzielony stan* będzie *gotowy*. Oznacza to dokładnie ten moment, kiedy w odniesieniu do obiektu promesy zostanie wykonana funkcja `set_value()` lub `set_exception()`. *Nie* oznacza to, że wątek, który ustawił promesę, się zakończył. Wątek może w dalszym ciągu przetwarzać inne instrukcje — na przykład zapisywać inne wyniki do innych promes.

Jeśli chcemy, aby *współdzielony stan* stał się *gotowy* w chwili, kiedy wątek naprawdę się zakończy — w celu wykonania instrukcji „sprzątania" lokalnych obiektów wątku i innych zadań przed przetwarzaniem wyników — powinniśmy wywołać funkcję `set_value_at_thread_exit()` lub `set_exception_at_thread_exit()`:

```
void doSomething (std::promise<std::string>& p)
{
    try {
        ...
        p.set_value_at_thread_exit(std::move(s));
    }
    catch (...) {
        p.set_exception_at_thread_exit(std::current_exception());
    }
}
```

Zwróćmy uwagę, że korzystanie z promes i futur nie ogranicza się do problemów wielowątkowych. Nawet w aplikacjach jednowątkowych możemy korzystać z promes do przechowywania wyników (wartości) lub wyjątków, które chcemy przetwarzać później, korzystając z obiektu futury.

Zwróćmy również uwagę na to, że nie możemy zapamiętać w promesie zarówno wartości, jak i wyjątku. Każda próba zrobienia czegoś podobnego zakończy się zgłoszeniem wyjątku `std::future_error` z kodem błędu `std::future_errc::promise_already_satisfied` (patrz punkt 4.3.1).

Więcej informacji na temat klasy `promise` można znaleźć w punkcie 18.3.4.

18.2.3. Klasa packaged_task<>

Funkcja `async()` daje uchwyt pozwalający na przetwarzanie wyniku zadania uruchomionego natychmiast w tle. Czasami jednak chcemy przetwarzać wynik zadania w tle, które niekoniecznie zostało uruchomione natychmiast. Na przykład

tym, czy i w jaki sposób wiele zadań równocześnie działa w tle, może zarządzać inny obiekt — taki jak pula wątków. W tym przypadku zamiast kodu:

```
double compute (int x, int y);

std::future<double> f = std::async(compute,7,5); // próba uruchomienia zadania w tle
...
double res = f.get(); // oczekiwanie na jego zakończenie i przetwarzanie wyniku (lub wyjątku)
```

możemy skorzystać z kodu w następującej postaci:

```
double compute (int x, int y);
std::packaged_task<double(int,int)> task(compute); // utworzenie zadania
std::future<double> f = task.get_future(); // pobranie jego futury
...
task(7,5); // uruchomienie zadania (zazwyczaj w osobnym wątku)
...
double res = f.get(); // oczekiwanie na jego zakończenie i przetwarzanie wyniku (lub wyjątku)
```

gdzie samo zadanie jest zazwyczaj (choć niekoniecznie) uruchamiane w osobnym wątku.

Tak więc klasa `std::packaged_task<>`, która również jest zdefiniowana w pliku nagłówkowym `<future>`, przechowuje zarówno funkcjonalność, która ma być uruchomiona, jak i jej możliwy wynik (tzw. *współdzielony stan* funkcjonalności — patrz podrozdział 18.3).

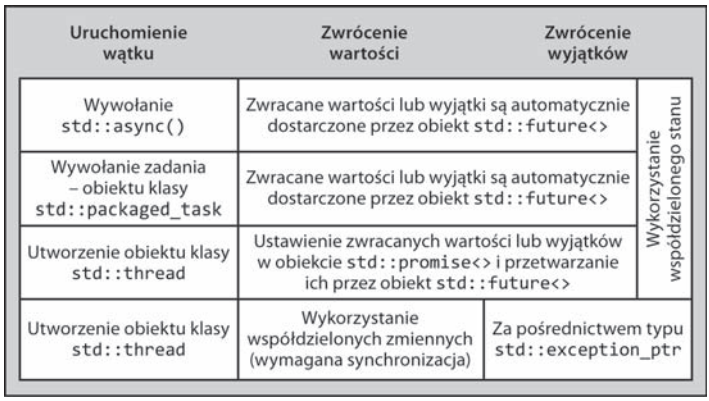
Więcej informacji dla temat klasy `packaged_task` można znaleźć w punkcie 18.3.5.

18.3. Uruchamianie wątku w szczegółach

Po wprowadzeniu w tematykę interfejsów wysokiego i niskiego poziomu pozwalających na (ewentualne) uruchamianie wątków i obsługę zwracanych wartości lub wyjątków spróbujmy podsumować koncepcje i podać kilka szczegółów, których w tym miejscu jeszcze nie wymieniono.

Mamy następujące warstwy umożliwiające uruchomienie wątków i obsługę wartości bądź wyjątków, które są przez nie zwracane (patrz rysunek 18.1):

- Za pomocą interfejsu niskopoziomowego klasy `thread` możemy uruchomić wątek. Aby zwrócić dane, potrzebujemy współdzielonych zmiennych (globalnych, statycznych lub przekazanych jako argumenty). Do zwrócenia wyjątków możemy skorzystać z typu `std::exception_ptr`, który jest zwracany przez funkcję `std::current_exception()` i może być przetwarzany przez funkcję `std::rethrow_exception()` (patrz punkt 4.3.3).
- Pojęcie *współdzielonego stanu* umożliwia obsługę zwracanych wartości lub wyjątków w wygodniejszy sposób. Taki *współdzielony stan*, który następnie można przetwarzać za pomocą obiektu klasy `future`, możemy stworzyć za pośrednictwem niskopoziomowego interfejsu klasy `promise`.
- Na wyższym poziomie za pośrednictwem klasy `packaged_task` lub metody `async()` *współdzielony stan* jest tworzony automatycznie. Następnie jest on ustawiany za pomocą instrukcji `return` lub poprzez nieobsłużony wyjątek.



RYSUNEK 18.1. Warstwy interfejsów wątków

- Za pomocą klasy `packaged_task` możemy stworzyć obiekt ze *współdzielonym stanem*, w którym musimy jawnie zaprogramować czas uruchomienia wątku.
- W przypadku użycia obiektu `std::async()` nie musimy przejmować się tym, kiedy wątek zostanie uruchomiony. Musimy jedynie wiedzieć, że kiedy jest potrzebny wynik, trzeba wywołać funkcję `get()`.

Współdzielone stany

Jak można zauważyć, centralnym pojęciem używanym prawie we wszystkich własnościach jest *współdzielony stan*. Pozwala on obiektom uruchamiającym i zarządzającym funkcjonalnościami w tle (promesie, pakietowi zadania lub funkcji `async()`) na komunikowanie się z obiektami przetwarzającymi wynik (futurą lub współdzieloną futurą). Zatem współdzielony stan pozwala na przechowywanie funkcjonalności, która ma być uruchomiona, pewnych informacji o jej stanie oraz wyniku (zwróconej wartości lub wyjątku).

Współdzielony stan jest *gotowy*, kiedy przechowuje wynik swojej funkcjonalności (kiedy wartość bądź wyjątek są gotowe do pobrania). Współdzielony stan zazwyczaj jest zaimplementowany jako obiekt z licznikiem referencji, który jest usuwany, kiedy zwolni go ostatni obiekt, który się do niego odwołuje.

18.3.1. Funkcja async() w szczegółach

Ogólnie rzecz biorąc, jak wspomniano w podrozdziale 18.1, `std::async()` jest funkcją pomocniczą, która — *jeśli to możliwe* — uruchamia pewną funkcjonalność w osobnym wątku. W rezultacie możemy wykonywać współbieżnie funkcjonalność, jeżeli pozwala na to platforma, ale nie tracimy żadnych własności, jeśli tak nie jest.

Jednakże dokładne zachowanie funkcji `async()` jest skomplikowane i w dużym stopniu zależy od strategii uruchamiania, którą można przekazać w roli pierwszego,

opcjonalnego argumentu. Z punktu widzenia programisty aplikacji istnieją trzy standardowe formy wywołania funkcji `async()`:

future **async** (`std::launch::async`, *F func*, *args*...)

- Próbuje uruchomić funkcjonalność *func* z argumentami *args* jako zadanie asynchroniczne (wątek równoległy).
- Jeśli to nie jest możliwe, zgłasza wyjątek `std::system_error` z kodem błędu `resource_unavailable_try_again` (patrz punkt 4.3.1).
- Jeśli program gwałtownie nie zakończy działania, uruchomiony wątek ma gwarancję możliwości zakończenia przed zakończeniem działania programu.
- Wątek zakończy działanie:
  - jeśli w odniesieniu do zwróconej futury zostanie wywołana funkcja `get()` lub `wait()`;
  - jeśli zostanie zniszczony ostatni obiekt odwołujący się do współdzielonego stanu reprezentowanego przez zwróconą futurę.
- Wynika stąd, że wywołanie funkcji `async()` zablokuje się do czasu zakończenia funkcjonalności *func*, o ile nie zostanie wykorzystana zwrócona wartość funkcji `async()`.

future **async** (`std::launch::deferred`, *F func*, *args*...)

- Przekazuje funkcjonalność *func* z argumentami *args* jako zadanie „odroczone", które zostanie wywołane synchronicznie, kiedy będzie wywołana funkcja `wait()` lub `get()` w odniesieniu do zwróconej futury.
- Jeśli żadna z funkcji: `wait()` lub `get()` nie zostanie wywołana, zadanie nigdy się nie rozpocznie.

future **async** (*F func*, *args*...)

- Jest to kombinacja wywołania funkcji `async()` ze strategiami uruchamiania `std::launch:async` i `std::launch::deferred`. Zgodnie z bieżącą sytuacją będzie wybrana jedna z dwóch form. Zatem funkcja `async()` spowoduje *odroczenie* wywołania funkcjonalności *func*, o ile bezpośrednie wywołanie w asynchronicznej strategii uruchamiania nie będzie możliwe.
- Zatem funkcja `async()` może rozpocząć nowy wątek dla funkcjonalności *func*, o ile uda się ją uruchomić. W przeciwnym razie wywołanie funkcjonalności *func* zostanie odroczone do czasu wywołania funkcji `get()` lub `wait()` w odniesieniu do zwróconej futury.
- Jedyną gwarancją, jaką daje to wywołanie, jest to, że po wywołaniu funkcji `get()` lub `wait()` dla zwróconej futury funkcjonalność *func* zostanie wywołana i zakończona.
- Bez wywołania funkcji `get()` lub `wait()` dla zwróconej futury funkcjonalność *func* może nigdy nie zostać wywołana.
- Zwróćmy uwagę, że ta forma funkcji `async()` nie zgłasza wyjątku `std::system_error`, o ile nie może wywołać funkcji `func` asynchronicznie (chociaż może wywołać błąd systemowy z innego powodu).

W odniesieniu do wszystkich tych form funkcji `async()` funkcjonalność *func* może być *wywoływalnym obiektem* (funkcją, funkcją składową, obiektem funkcyjnym, wyrażeniem lambda; patrz podrozdział 4.4). Kilka przykładów można znaleźć w punkcie 18.1.2.

Przekazanie strategii uruchamiania `std::launch::async|std::launch::deferred` do funkcji `async()` może wywołać takie samo zachowanie jak nieprzekazanie żadnej strategii uruchamiania. Przekazanie `0` w roli strategii uruchamiania skutkuje niezdefiniowanym działaniem (tym przypadkiem nie zajmuje się standardowa biblioteka C++, a różne implementacje zachowują się różnie).

18.3.2. Futury w szczegółach

Klasa `future<>`[10], wprowadzona w podrozdziale 18.1, reprezentuje *wynik* operacji. Może to być zwrócona wartość albo wyjątek, ale nie jedno i drugie. Wynik jest zarządzany za pomocą *współdzielonego stanu*, który ogólnie rzecz biorąc, może być stworzony przez obiekty `std::async()`, `std::packaged_task` lub obiekt promesy. Wynik może jeszcze nie istnieć, zatem futura może również zawierać wszystko to, co jest konieczne do wygenerowania wyniku.

Jeśli futura została zwrócona przez funkcję `async()` (patrz punkt 18.3.1), a powiązane z nią zadanie było *odroczone*, funkcje `get()` lub `wait()` uruchomią je synchronicznie. Zwróćmy uwagę, że funkcje `wait_for()` oraz `wait_until()` *nie* rozpoczynają odroczonego zadania.

Wynik może być pobrany tylko raz. Z tego powodu futura może być ważna bądź nieważna: jeśli jest *ważna*, to istnieje powiązana z nią operacja, dla której jeszcze nie pobrano wyniku lub wyjątku.

Operacje dostępne dla klasy `future<>` zestawiono w tabeli 18.1.

TABELA 18.1. Operacje klasy future<>

Operacja	Efekt
future *f*	Domyślny konstruktor. Tworzy futurę, która jest nieważna.
future *f*(*rv*)	Konstruktor przenoszący. Tworzy nową futurę, która uzyskuje stan *rv* i unieważnia stan *rv*.
f.~***future***()	Niszczy stan oraz niszczy obiekt `*this`.
f = rv	Przypisanie z przeniesieniem. Niszczy poprzedni stan futury *f*, uzyskuje stan *rv* i unieważnia stan *rv*.
f.valid()	Zwraca `true`, jeśli *f* jest ważna. W takim przypadku można wywołać funkcje składowe wymienione poniżej.
f.get()	Blokuje się do czasu wykonania operacji w tle (wymuszając synchroniczne uruchomienie *odroczonej* funkcjonalności powiązanej z futurą). Zwraca wynik (o ile jest dostępny) lub zgłasza wyjątek, jeśli jakiś wystąpił, oraz unieważnia stan futury.

[10] Pierwotnie w procesie standaryzacji klasa miała nazwę `unique_future`.

TABELA 18.1. Operacje klasy future<> (ciąg dalszy)

Operacja	Efekt
`f.wait()`	Blokuje się do czasu wykonania operacji w tle (wymuszając synchroniczne uruchomienie *odroczonej* funkcjonalności).
`f.wait_for(dur)`	Blokuje się na czas *dur* lub do czasu wykonania operacji w tle (nie następuje wymuszenie uruchomienia *odroczonej* funkcjonalności).
`f.wait_until(tp)`	Blokuje się, aż upłynie czas *tp* lub do czasu wykonania operacji w tle (nie następuje wymuszenie uruchomienia *odroczonego* wątku).
`f.share()`	Zwraca obiekt `shared_future` z bieżącym stanem i unieważnia futurę *f*.

Zwróćmy uwagę, że wartość zwrócona przez funkcję `get()` zależy od typu, dla którego obiekt `future<>` został wyspecjalizowany:

- Jeśli to jest typ `void`, funkcja `get()` także ma typ `void` i nie zwraca niczego.
- Jeśli futura jest sparametryzowana z typem referencyjnym, to funkcja `get()` zwraca referencję do zwróconej wartości.
- W przeciwnym razie funkcja `get()` zwraca kopię zwracanej wartości lub realizuje przypisanie z przeniesieniem (ang. *move assignment*) w zależności od tego, czy typ zwracanej wartości pozwala na semantykę przypisania z przeniesieniem.

Zwróćmy uwagę, że funkcję `get()` można wywołać tylko raz, ponieważ wywołanie `get()` powoduje unieważnienie stanu futury.

Dla futury, która jest nieważna, wywołanie czegokolwiek innego niż destruktora, operatora przypisania z przeniesieniem bądź funkcji `valid()` skutkuje niezdefiniowanym zachowaniem. Z tego względu standard zaleca zgłoszenie wyjątku typu `future_error` (patrz punkt 4.3.1) z kodem `std::future_errc::no_state`, ale nie jest to konieczne.

Zwróćmy uwagę, że ani konstruktor kopiujący, ani operator przypisania z kopiowaniem nie jest dostępny. Dzięki temu mamy pewność, że żadne dwa obiekty nie będą współdzielić stanu operacji w tle. Stan można przenieść do innego obiektu futury tylko poprzez wywołanie konstruktora przenoszącego lub operatora przypisania z przeniesieniem. Stan zadań w tle może być jednak współdzielony w wielu obiektach dzięki wykorzystaniu obiektu klasy `shared_future` zwracanego przez funkcję `share()`.

Jeśli dla futury, która jest ostatnim właścicielem współdzielonego stanu, zostanie wywołany destruktor, a powiązane z futurą zadanie zostało uruchomione, ale jeszcze się nie zakończyło, to destruktor zablokuje się do czasu zakończenia zadania.

18.3.3. Futury współdzielone w szczegółach

Klasa `shared_future<>` (wprowadzona w punkcie 18.1.3) udostępnia taką samą semantykę i interfejs co klasa `future` (patrz punkt 18.3.2) z następującymi różnicami:

- Dozwolonych jest wiele wywołań funkcji `get()`. Z tego powodu wywołanie funkcji `get()` nie unieważnia stanu futury.
- Dostępne są mechanizmy semantyki kopiowania (konstruktor kopiujący, operator kopiowania z przypisaniem).
- Funkcja `get()` jest stałą funkcją składową, która zwraca stałą referencję do wartości zapisanej we *współdzielonym stanie* (co oznacza, że musimy zadbać o to, aby czas życia zwróconej referencji był krótszy od czasu życia *współdzielonego stanu*). Dla klasy `std::future` funkcja `get()` jest *niestałą* funkcją składową zwracającą kopię po przeniesieniu z przypisaniem (lub kopię, jeśli jest dozwolona), pod warunkiem że klasa jest wyspecjalizowana dla typu referencyjnego.
- Funkcja składowa `share()` nie jest dostępna.

Fakt, że wartość zwracana przez funkcję `get()` nie jest kopiowana, stwarza pewne ryzyko. Oprócz problemów związanych z czasem życia możliwe są sytuacje wyścigu. Sytuacje wyścigu występują ze względu na nieczytelną kolejność kolidujących ze sobą operacji na tych samych danych — na przykład niezsynchronizowany odczyt i zapis z wielu wątków. Powodują one niezdefiniowane zachowanie programu (patrz punkt 18.4.1).

Ten sam problem dotyczy wyjątków. Podczas procesu standaryzacyjnego omawiano przypadek, kiedy wyjątek został przechwycony przez referencję, a następnie zmodyfikowany:

```
try {
    shared_future<void> sp = async(f);
    sp.get();
}
catch (E& e) {
    e.modify(); // ryzyko niezdefiniowanego zachowania ze względu na wyścig o dane
}
```

Powyższy kod wprowadza problem wyścigu o dane, o ile inny wątek przetwarza wyjątek. W celu rozwiązania tego problemy zaproponowano, aby funkcje `current_exception()` i `rethrow_exception()`, które są wewnętrznie używane do przekazywania wyjątków pomiędzy wątkami, tworzyły kopie wyjątków. Uznano jednak, że koszty takiej zmiany będą zbyt wysokie. W związku z tym programiści muszą wiedzieć, co robią, jeśli posługują się niestałymi referencjami wykorzystywanymi w różnych wątkach.

18.3.4. Klasa std::promise w szczegółach

Obiekt klasy `std::promise`, który zaprezentowano po raz pierwszy w punkcie 18.2.2, służy do tymczasowego przechowywania wartości (wyniku) bądź wyjątku. Ogólnie rzecz biorąc, promesa może zawierać *współdzielony stan* (patrz punkt 18.3). O *współdzielonym stanie* mówimy, że jest *gotowy*, jeśli zawiera wartość bądź wyjątek. Operacje dostępne dla klasy `promise` zestawiono w tabeli 18.2.

TABELA 18.2. Operacje klasy promise<>

Operacja	Efekt
`promise p`	Domyślny konstruktor. Tworzy promesę razem ze współdzielonym stanem.
`promise p(allocator_arg,alloc)`	Tworzy promesę ze współdzielonym stanem, który używa *alloc* w roli alokatora.
`promise p(rv)`	Konstruktor przenoszący. Tworzy nowy obiekt promesy, która uzyskuje stan *rv* i przenosi współdzielony stan z *rv*.
`p.~promise()`	Zwalnia współdzielony stan, a jeśli nie jest on gotowy (nie zawiera wartości lub wyjątku), zapisuje wyjątek `std::future_error` z warunkiem `broken_promise`.
`p = rv`	Przypisanie z przeniesieniem. Przenosi i przypisuje stan *rv* do *p*, a jeśli promesa *p* nie była gotowa, zapisuje w niej wyjątek `std::future_error` z kodem błędu `broken_promise`.
`swap(p1,p2)`	Zamienia miejscami stany promes *p1* z *p2*.
`p1.swap(p2)`	Zamienia miejscami stany promes *p1* z *p2*.
`p.get_future()`	Zwraca obiekt futury do odczytania *współdzielonego stanu* (wynik wątku).
`p.set_value(val)`	Ustawia *val* jako wartość (wynik) i przełącza stan na *gotowy* (lub zgłasza wyjątek `std::future_error`).
`p.set_value_at_thread_exit(val)`	Ustawia *val* jako wartość (wynik) i przełącza stan na *gotowy* na końcu bieżącego wątku (lub zgłasza wyjątek `std::future_error`).
`p.set_exception(e)`	Ustawia *e* jako wyjątek i przełącza stan na *gotowy* (lub zgłasza wyjątek `std::future_error`).
`p.set_exception_at_thread_exit(e)`	Ustawia *e* jako wyjątek i przełącza stan na *gotowy* na końcu bieżącego wątku (lub zgłasza wyjątek `std::future_error`).

Zwróćmy uwagę, że metodę `get_future` można wywołać tylko raz. Drugie wywołanie powoduje zgłoszenie wyjątku `std::future_error` z kodem błędu `std::future_errc::future_already_retrieved`. Ogólnie rzecz biorąc, jeśli z promesą nie jest powiązany *współdzielony stan*, wywołanie `get_future()` może spowodować zgłoszenie wyjątku `std::future_error` z kodem błędu `std::future_errc::no_state`.

Wszystkie funkcje składowe, które ustawiają wartość lub wyjątek, zapewniają bezpieczeństwo wątków. Oznacza to, że zachowują się one w taki sposób, jakby muteks zapewniał, że tylko jedna z nich może w danym momencie aktualizować *współdzielony stan*.

18.3.5. Klasa std::packaged_task w szczegółach

Klasa `std::packaged_task<>` służy do przechowywania zarówno funkcjonalności, która ma być uruchomiona, jak i jej wyniku (tzw. *współdzielonego stanu* funkcjonalności — patrz podrozdział 18.3), który może być wartością zwracaną przez funkcjonalność albo zgłoszonym wyjątkiem. Pakiet zadania można zainicjować za pomocą

powiązanej funkcjonalności. Następnie można wywołać tę funkcjonalność za pomocą operatora () w odniesieniu do pakietu zadania. Na koniec możemy przetworzyć wynik poprzez pobranie futury dla pakietu zadania. Operacje dostępne dla klasy `packaged_task` zestawiono w tabeli 18.3.

TABELA 18.3. Operacje klasy packaged_task<>

Operacja	Efekt
`packaged_task pt`	Domyślny konstruktor. Tworzy pakiet zadania bez współdzielonego stanu i bez zapisanego w nim zadania.
`packaged_task pt(f)`	Tworzy obiekt dla zadania *f*.
`packaged_task pt(alloc,f)`	Tworzy obiekt dla zadania *f* z wykorzystaniem alokatora *alloc*.
`packaged_task pt(rv)`	Konstruktor przenoszący. Przenosi pakiet zadania *rv* (zadanie i stan) do pakietu zadania *pt* (po tej operacji *rv* nie ma współdzielonego stanu).
`pt.~packaged_task()`	Niszczy obiekt `*this` (może przełączać współdzielony stan do gotowości).
`pt = rv`	Przypisanie z przeniesieniem. Przenosi i przypisuje pakiet zadania *rv* (zadanie i stan) do pakietu zadania *pt* (po tej operacji *rv* nie ma współdzielonego stanu).
`swap(pt1,pt2)`	Zamienia miejscami pakiety zadań.
`pt1.swap(pt2)`	Zamienia miejscami pakiety zadań.
`pt.valid()`	Zwraca `true`, jeśli pakiet zadania *pt* ma współdzielony stan.
`pt.get_future()`	Zwraca obiekt futury do odczytania *współdzielonego stanu* (wynik zadania).
`pt(args)`	Wywołuje zadanie (z opcjonalnymi argumentami) i przełącza współdzielony stan do gotowości.
`pt.make_ready_at_thread_exit(args)`	Wywołuje zadanie (z opcjonalnymi argumentami) i na końcu wątku przełącza współdzielony stan do stanu *gotowy*.
`pt.reset()`	Tworzy nowy współdzielony stan dla pakietu zadania *pt* (może przełączać stary współdzielony stan do gotowości).

Wyjątki spowodowane przez konstruktor pobierający zadanie — na przykład jeśli brakuje pamięci — także są zapisywane we *współdzielonym stanie*.

Próba wywołania zadania lub funkcji `get_future()` w sytuacji, gdy nie jest dostępny stan, zwraca błąd `std::future_error` (patrz punkt 4.3.1) z kodem błędu `std::future_errc::no_state`. Wywołanie funkcji `get_future()` po raz drugi powoduje zgłoszenie wyjątku typu `std::future_error` z kodem błędu `std::future_errc::future_already_retrieved`. Wywołanie zadania po raz drugi powoduje zgłoszenie wyjątku `std::future_error` z kodem błędu `std::future_errc::promise_already_satisfied`.

Wywołanie destruktora oraz funkcji `reset()` powoduje *porzucenie* współdzielonego stanu. Oznacza to, że pakiet zadania zwalnia współdzielony stan, a jeśli nie był jeszcze gotowy, przełącza go do stanu *gotowy*, zapisując w nim wyjątek `std::future_error` z kodem błędu `std::future_errc::broken_promise`.

Dostępna jest również funkcja `make_ready_at_thread_exit()`, która zapewnia sprzątanie lokalnych obiektów i innych konstrukcji wątku z zakończeniem zadania, zanim wyniki zostaną przetworzone.

18.3.6. Klasa std::thread w szczegółach

Obiekt klasy `std::thread`, który zaprezentowano po raz pierwszy w punkcie 18.2.1, służy do uruchamiania i reprezentowania wątku. Obiekty te są przeznaczone do odwzorowania jeden-do-jednego wątków dostarczanych przez system operacyjny. Operacje dostępne dla klasy `thread` zestawiono w tabeli 18.4.

TABELA 18.4. Operacje obiektów klasy thread

Operacja	Efekt
thread *t*	Domyślny konstruktor. Tworzy niełączny (ang. *nonjoinable*) obiekt `thread`.
thread *t*(*f*,...)	Tworzy obiekt `thread` reprezentujący funkcjonalność *f*, która jest uruchamiana w osobnym wątku (z dodatkowymi argumentami) bądź zgłasza wyjątek `std::system_error`.
thread *t*(*rv*)	Konstruktor przenoszący. Tworzy nowy obiekt wątku. Wątek uzyskuje stan *rv* i staje się niełączny.
t.~***thread***()	Niszczy obiekt `*this`; wywołuje funkcję `std::terminate()`, jeśli obiekt jest *łączny*.
t = *rv*	Przypisanie z przeniesieniem. Przenosi i przypisuje stan *rv* do *t* lub wywołuje funkcję `std::terminate()`, jeśli obiekt *t* jest *łączny*.
t.joinable()	Zwraca `true`, jeśli z obiektem *t* jest powiązany wątek (wątek jest łączny).
t.join()	Oczekuje na zakończenie powiązanego wątku (zwraca wyjątek `std::system_error`, jeśli wątek nie jest *łączny*), i powoduje, że obiekt staje się *niełączny*.
t.detach()	Zwalnia powiązanie obiektu *t* z jego wątkiem. Wątek kontynuuje działanie (zgłasza wyjątek `std::system_error`, jeśli wątek nie jest *łączny*), i powoduje, że obiekt staje się *niełączny*.
t.get_id()	Zwraca unikatową wartość `std::thread::id`, jeśli obiekt *t* jest *łączny*, lub `std::thread::id()`, jeśli tak nie jest.
t.native_handle()	Zwraca specyficzny dla platformy typ `native_handle_type` dla nieprzenośnych rozszerzeń.

Powiązanie pomiędzy obiektem `thread` a wątkiem rozpoczyna się od zainicjowania (albo przeniesienia, skopiowania lub przypisania) *wywoływalnego obiektu* (patrz podrozdział 4.4) wraz z opcjonalnymi dodatkowymi argumentami. Powiązanie kończy się wywołaniem funkcji `join()` (oczekiwaniem na wynik wątku) albo wywołaniem funkcji `detach()` (jawną utratą powiązania z wątkiem). Jedna bądź druga funkcja musi być wywołana, zanim upłynie czas życia obiektu wątku, albo zostanie przypisany nowy wątek. W przeciwnym razie program kończy działanie poprzez wywołanie funkcji `std::terminate()` (patrz punkt 5.8.2).

O obiekcie `thread`, z którym jest powiązany wątek, mówimy, że jest *łączny* (ang. *joinable*). W takim przypadku funkcja `joinable()` zwraca `true`, natomiast funkcja `get_id()` zwraca identyfikator wątku, który jest różny od identyfikatora zwracanego przez funkcję `std::thread::id()`.

Identyfikatory wątków mają własny typ `std::thread::id`. Domyślny konstruktor tego typu zwraca unikatowy identyfikator reprezentujący „brak wątku". Funkcja `thread::get_id()` zwraca tę wartość, jeśli żaden wątek nie jest powiązany z obiektem `thread`, albo inny unikatowy identyfikator, jeśli obiekt `thread` jest powiązany z wątkiem (jest *łączny*). Jedyną operacją dostępną dla identyfikatorów wątków jest porównywanie ich bądź zapisywanie do strumienia wyjściowego. Oprócz tego dostępna jest funkcja haszująca, która pozwala zarządzać identyfikatorami wątków w nieuporządkowanych kontenerach (patrz podrozdział 7.9). Identyfikator wątku, który został zakończony, może być wykorzystany ponownie. Poza tym nie należy robić żadnych założeń dotyczących identyfikatorów wątków — zwłaszcza odnośnie do ich wartości. Szczegółowe informacje na ten temat można znaleźć w punkcie 18.2.1.

Zwróćmy uwagę, że odłączone wątki nie powinny próbować dostępu do obiektów, których czas życia się zakończył. To implikuje problem polegający na tym, że w przypadku zakończenia programu trzeba zadbać o to, by odłączone wątki nie korzystały z obiektów globalnych (statycznych) (patrz punkt 18.2.1).

Dodatkowo klasa `std::thread` zawiera statyczną funkcję składową do odpytywania sugestii możliwej liczby równoległych wątków:

`unsigned int` **`std::thread::hardware_concurrency`** `()`

- Zwraca liczbę możliwych wątków.
- Wartość ta jest jedynie sugestią. Nie ma gwarancji, że będzie to wartość dokładna.
- Zwraca `0`, jeśli liczby nie da się obliczyć lub jeśli nie jest ona dobrze zdefiniowana.

18.3.7. Przestrzeń nazw this_thread

Dla wszystkich wątków, z wątkiem głównym włącznie, w pliku nagłówkowym `<thread>` zadeklarowano przestrzeń nazw `std::this_thread`, dostarczającą specyficznych dla wątków funkcji globalnych wyszczególnionych w tabeli 18.5.

TABELA 18.5. Operacje specyficzne dla wątków z przestrzeni nazw std::this_thread

Operacja	Efekt
`this_thread::get_id()`	Zwraca identyfikator ID bieżącego wątku.
`this_thread::sleep_for(dur)`	Blokuje wątek na czas *dur*.
`this_thread::sleep_until(tp)`	Blokuje wątek do chwili *tp*.
`this_thread::yield()`	Sugestia przekazania sterowania do następnego wątku.

Zwróćmy uwagę, że funkcje `sleep_for()` i `sleep_until()` zazwyczaj działają inaczej w przypadku korekt czasu systemowego (szczegółowe informacje można znaleźć w punkcie 5.7.5).

Operacja `this_thread::yield()` umożliwia przekazanie do systemu sugestii, że warto zrezygnować z pozostałej części slotu czasowego bieżącego wątku. Dzięki temu środowisko wykonawcze może zmodyfikować harmonogram działania wątków, by umożliwić działanie innym wątkom. Typowym przykładem zastosowania tego mechanizmu jest zrezygnowanie ze sterowania w czasie oczekiwania lub „odpytywania" innego wątku (patrz punkt 18.1.1) lub ustawienie atomowej flagi przez inny wątek (patrz punkt 18.4.3)[11]:

```
while (!readyFlag) { // pętla do chwili, kiedy dane będą gotowe
    std::this_thread::yield();
}
```

Innym przykładem zastosowania tego mechanizmu może być sytuacja, gdy nie uda nam się uzyskać blokady bądź muteksu w przypadku korzystania z wielu blokad bądź muteksów naraz. Wtedy możemy przyspieszyć działanie aplikacji poprzez wykorzystanie funkcji `yield()`, zanim spróbujemy wykorzystać blokady (muteksy) w innej kolejności[12].

18.4. Synchronizacja wątków, czyli największy problem współbieżności

Korzystanie z wielu wątków niemal zawsze wiąże się z równoległym dostępem do danych. Rzadko się zdarza, aby wiele wątków działało całkowicie niezależnie od siebie. Wątki mogą dostarczać danych, które są przetwarzane przez inne wątki, lub przygotowywać warunki wstępne niezbędne do uruchomienia innych procesów.

W tym aspekcie wielowątkowość staje się trudna. Jest wiele rzeczy, które mogą się nie udać. Albo mówiąc inaczej, wiele rzeczy może zachowywać się inaczej w porównaniu z tym, czego może oczekiwać naiwny (a nawet doświadczony) programista.

[11] Dziękujmy Bartoszowi Milewskiemu za ten przykład.
[12] Dziękujmy Howardowi Hinnantowi za ten przykład.

Zatem zanim przejdziemy do omówienia różnych sposobów synchronizacji wątków i współbieżnego dostępu do danych, powinniśmy zrozumieć ten problem. Następnie będziemy mogli omówić następujące techniki synchronizacji wątków:

- Muteksy i blokady (patrz podrozdział 18.5), włącznie z funkcją `call_once()` (patrz punkt 18.5.3).
- Zmienne warunkowe (patrz podrozdział 18.6).
- Zmienne `atomic` (patrz podrozdział 18.7).

18.4.1. Uwaga na współbieżność!

Zanim przejdziemy do szczegółów problemów współbieżności, sformułuję pierwszą zasadę, tak na wszelki wypadek, gdyby któryś z czytelników zechciał zacząć programowanie bez wchodzenia w szczegóły tego punktu. Gdybyśmy zdecydowali się, że nauczymy się tylko jednej zasady postępowania z wieloma wątkami, to powinna to być następująca zasada:

> **Jedynym bezpiecznym sposobem współbieżnego dostępu do tych samych danych przez wiele wątków bez stosowania synchronizacji jest sytuacja, kiedy *WSZYSTKIE* wątki jedynie *CZYTAJĄ* dane.**

Pisząc „te same dane", mam na myśli dane korzystające z tej samej lokalizacji w pamięci. Jeśli różne wątki współbieżnie korzystają z *różnych* zmiennych bądź obiektów albo różnych ich składowych, wtedy nie ma problemu, ponieważ począwszy od C++11, wszystkie zmienne oprócz pól bitowych mają gwarantowane własne lokalizacje w pamięci[13]. Jedynym wyjątkiem są pola bitowe, ponieważ różne pola bitowe mogą współdzielić lokalizację w pamięci, dlatego dostęp do różnych pól bitowych oznacza współdzielony dostęp do tych samych danych.

Jednak jeśli dwa bądź więcej wątków współbieżnie korzysta z tej *samej* zmiennej, obiektu albo jego składowej i co najmniej jeden z wątków wprowadza modyfikacje, możemy łatwo wpaść w kłopoty, jeśli nie zadbamy o synchronizację tego dostępu. Tę sytuację zgodnie z terminologią języka C++ nazywamy *wyścigiem o dane*. Zgodnie ze standardem C++11 *wyścig o dane* jest zdefiniowany jako „dwie kolidujące ze sobą akcje w różnych wątkach, z których co najmniej jedna nie jest atomowa i żadna nie jest wykonywana wcześniej niż druga". Wyścig o dane zawsze kończy się niezdefiniowanym zachowaniem.

Tak jak zwykle w sytuacjach wyścigu problem polega na tym, że kod *często* realizuje to, co zamierzamy, ale nie zawsze *działa*. Jest to jeden z najbardziej poważnych problemów, na jakie napotykamy w programowaniu. Takie sytuacje jak użycie innych danych, przejście do trybu produkcyjnego lub zmiana platformy mogą

[13] Gwarancji osobnych lokalizacji pamięci dla różnych obiektów nie było w standardach przed C++11. Standardy C++98/C++03 dotyczyły wyłącznie aplikacji jednowątkowych. Zatem ściśle rzecz biorąc, przed C++11 współbieżny dostęp do różnych obiektów skutkował niezdefiniowanym zachowaniem, choć w praktyce zwykle nie powodowało to żadnych problemów.

spowodować nagłe problemy z działaniem kodu. Z tego względu w przypadku, gdy korzystamy z wielu wątków, powinniśmy zadbać o współbieżny dostęp do danych.

18.4.2. Przyczyna problemu jednoczesnego dostępu do danych

Aby zrozumieć problemy jednoczesnego dostępu do danych, musimy uświadomić sobie, co gwarantuje język C++ w kontekście współbieżności. Należy pamiętać, że język programowania, jakim jest C++, zawsze jest abstrakcją do obsługi różnych platform i sprzętu, które zapewniają różne możliwości i interfejsy w zależności od ich struktury i przeznaczenia. Zatem standard, jakim jest C + +, określa *efekt* instrukcji i operacji, a nie odpowiadający im wygenerowany kod asemblera. Standard daje odpowiedź na pytanie *co*, a nie *jak*.

Na ogół zachowanie nie jest zdefiniowane tak precyzyjnie, aby istniał tylko jeden sposób jego implementacji. W rzeczywistości zachowanie może nawet nie być jawnie zdefiniowane. Na przykład kolejność wartościowania argumentów w wywołaniu funkcji jest nieokreślona. Program oczekujący konkretnej kolejności wartościowania będzie działał w niezdefiniowany sposób.

W związku z tym powstaje zasadnicze pytanie: Jakie gwarancje daje język? Programiści nie powinni spodziewać się więcej, pomimo że dodatkowe gwarancje wydają się „oczywiste". W rzeczywistości zgodnie z tzw. regułą *jak gdyby* (ang. *as-if rule*) każdy kompilator może optymalizować kod, o ile na zewnątrz program zachowuje się tak samo. Zatem wygenerowany kod jest *czarną skrzynką* i może być różny, o ile *obserwowane zachowanie* pozostaje niezmienne. Cytując standard C++:

> Implementacja może swobodnie pomijać wszelkie wymogi niniejszego międzynarodowego standardu, o ile wynik określony na podstawie obserwowanego zachowania programu wygląda tak, *jakby* wymaganie było przestrzegane. Na przykład faktyczna implementacja nie musi wartościować części wyrażenia, jeżeli może wywnioskować, że jego wartość nie będzie używana i że z tego powodu nie będzie żadnych skutków ubocznych wpływających na obserwowane zachowanie programu.

Wszelkie niezdefiniowane zachowania pozostawiają dostawcom zarówno kompilatorów, jak i sprzętu swobodę generowania możliwie najlepszego kodu, niezależnie od ich kryteriów oceny tego, co uważają za „najlepsze". Dotyczy to zarówno dostawców kompilatorów, jak i sprzętu: kompilatory mogą rozwijać pętle, zmieniać kolejność instrukcji, eliminować martwy kod, wcześniej pobierać dane, a w nowoczesnych architekturach bufor sprzętowy może zmieniać kolejność ładowania lub zapisywania.

Zmiany kolejności instrukcji mogą być przydatne dla poprawy szybkości działania programu, ale mogą zmienić zachowanie programu. Korzystanie z szybkości jest przydatne, ale bezpieczeństwo nie jest domyślne. W związku z tym zwłaszcza

w aspekcie jednoczesnego dostępu do danych musimy zrozumieć, jakie gwarancje daje nam język.

18.4.3. Zakres problemu, czyli co może pójść źle?

Aby dać kompilatorom i sprzętowi wystarczającą swobodę optymalizowania kodu, C++, ogólnie rzecz biorąc, *nie daje* pewnych gwarancji, których można by się było spodziewać. Powodem jest to, że stosowanie tych gwarancji w każdym przypadku, a nie tylko tam, gdzie to jest przydatne, wiązałoby się ze zbyt dużym kosztem wydajności. W rzeczywistości w języku C++ mogłyby wystąpić następujące problemy:

- **Niezsynchronizowany dostęp do danych**. Gdy dwa wątki działające równolegle czytają i zapisują te same dane, pozostaje otwartą kwestią to, która instrukcja będzie wykonana w pierwszej kolejności.
- **Dane w trakcie zapisywania**. Gdy jeden wątek czyta dane, które inny wątek modyfikuje, wątek czytający może odczytywać dane nawet w trakcie zapisywania przez drugi wątek. Zatem odczytywana wartość nie jest ani nowa, ani stara.
- **Zmieniona kolejność instrukcji**. Można zmienić kolejność instrukcji i operacji w taki sposób, że zachowanie każdego pojedynczego wątku będzie poprawne, ale po połączeniu *wszystkich* wątków zachowanie będzie różne od oczekiwanego.

Niezsynchronizowany dostęp do danych

Poniższy kod zapewnia, że funkcja `f()` jest wywoływana dla bezwzględnej wartości `val`. Argument `val` jest negowany, jeżeli ma ujemną wartość:

```
if (val >= 0) {
    f(val);      // przekazanie dodatniej wartości val
}
else {
    f(-val);     // przekazanie zanegowanej wartości val
}
```

W środowisku jednowątkowym ten kod działa poprawnie. Jednak w kontekście wielowątkowym nie musi działać. Jeśli wiele wątków ma dostęp do zmiennej `val`, to wartość `val` może się zmienić między klauzulą `if` a wywołaniem `f()`, zatem do funkcji `f()` zostanie przekazana ujemna wartość.

Z tego samego powodu prosty kod, na przykład:

```
std::vector<int> v;
...
if (!v.empty()) {
    std::cout << v.front() << std::endl;
}
```

może być problemem, jeśli zmienna `v` jest współdzielona między wieloma wątkami. Pomiędzy wywołaniem `empty()`a wywołaniem `front()` zmienna `v` może stać się pusta, co może spowodować zachowanie niezdefiniowane (patrz punkt 7.3.2).

Należy zauważyć, że problem ten dotyczy również kodu implementującego funkcje dostarczane przez bibliotekę standardową C++. Na przykład gwarancja, że instrukcja:

```
v.at(5) // zwraca wartość elementu o indeksie 5
```

zgłosi wyjątek, jeśli `v` nie ma wystarczająco dużo elementów, nie ma zastosowania, jeśli jakiś inny wątek może modyfikować `v` w czasie, gdy jest wywoływana funkcja `at()`. Zatem warto pamiętać o następującej regule:

> **O ile nie zaznaczono inaczej, standardowe funkcje biblioteczne C++ zazwyczaj nie wspierają współbieżnego zapisywania lub odczytywania w przypadku realizowania zapisu do tej samej struktury danych[14].**

Zatem o ile nie podano inaczej, wiele wywołań tego samego obiektu z wielu wątków może spowodować niezdefiniowane zachowanie.

Jednak standardowa biblioteka C++ daje pewne gwarancje dotyczące bezpieczeństwa wątków (patrz podrozdział 4.5). Na przykład:

- Współbieżny dostęp do *różnych elementów* tego samego kontenera jest możliwy (z wyjątkiem klasy `vector<bool>`). Tak więc różne wątki mogą jednocześnie czytać i (lub) zapisywać różne elementy tego samego kontenera. Na przykład każdy wątek może coś przetwarzać i zapisywać wynik w „swoim" elemencie współdzielonego wektora.
- Jednoczesny dostęp do strumienia łańcuchów znaków, strumienia pliku lub bufora strumienia skutkuje niezdefiniowanym zachowaniem. Jednak zgodnie z tym, co widzieliśmy wcześniej w tym rozdziale, sformatowane wejście i wyjście do standardowego strumienia, który jest synchronizowany za pomocą mechanizmów wejścia-wyjścia języka C (patrz punkt 15.14.1), jest możliwe, chociaż może powodować przeplatanie znaków.

Dane w trakcie zapisywania

Załóżmy, że mamy następującą zmienną[15]:

```
long long x = 0;
```

oraz jeden wątek zapisujący dane:

```
x = -1;
```

[14] Jak wskazuje Hans Boehm, podejście polegające na wspieraniu jednoczesnego dostępu do obiektów bibliotecznych w ogóle nie byłoby przydatne, bo jeśli potrzebujemy synchronizacji wokół dostępu do struktury danych, to zazwyczaj nie chodzi o ochronę indywidualnego dostępu, ale o ochronę większych fragmentów kodu. Oznacza to, że programiści i tak powinni stosować swoje własne blokady, a blokady dostarczone za pośrednictwem biblioteki będą w najlepszym razie nadmiarowe.

[15] Ten przykład, za zgodą autora, przytoczono z pozycji [N2480:MemMod].

i drugi, który te dane odczytuje:

```
std::cout << x;
```

Jaki jest wynik działania programu? Tzn. jaką wartość odczyta drugi wątek przed wyświetleniem wartości x? Możliwe są następujące odpowiedzi:

- 0 (stara wartość x), jeśli pierwszy wątek jeszcze nie przypisał wartości –1;
- -1 (nowa wartość x), jeśli pierwszy wątek już przypisał wartość –1;
- *dowolna inna wartość*, jeśli drugi wątek czyta zmienną x w czasie przypisywania wartości -1 przez pierwszy wątek.

Ostatni przypadek — *dowolna inna wartość* — może łatwo się zdarzyć, jeśli na przykład na 32-bitowej maszynie przypisanie skutkuje dwoma zapisami w pamięci, a odczyt przez drugi wątek zachodzi w momencie, kiedy pierwszy zapis został wykonany, a drugi jeszcze nie.

Należy pamiętać, że nie dotyczy to tylko zmiennych typu `long long`. Nawet w przypadku podstawowych typów danych, takich jak `int` lub `bool`, standard *nie* gwarantuje, że odczyt bądź zapis będzie *atomowy*, tzn. że odczyt lub zapis wiąże się z wyłącznym i niczym niezakłóconym dostępem do danych. Wyścig o dane może być mniej prawdopodobny, ale wyeliminowanie możliwości takiej sytuacji wymaga podjęcia odpowiednich kroków.

To samo dotyczy bardziej złożonych struktur danych, nawet jeśli są one dostarczone za pośrednictwem standardowej biblioteki C++. Na przykład w przypadku obiektu `std::list<>` (patrz podrozdział 7.5) do programisty należy zadbanie o to, aby lista nie została zmodyfikowana przez inny wątek w czasie, gdy wątek bieżący wstawia lub usuwa element. W przeciwnym razie inny wątek może wykorzystywać niespójny stan listy, gdzie na przykład wskaźnik w przód został już zmodyfikowany, ale wskaźnik wstecz jeszcze nie.

Zmieniona kolejność instrukcji

Przeanalizujemy inny prosty przykład[16]. Przypuśćmy, że mamy dwa współdzielone obiekty, zmienną `int` do przekazywania danych z jednego wątku do innego oraz zmienną `readyFlag` typu `Boolean`, która sygnalizuje, kiedy pierwszy wątek dostarczył danych:

```
long data;
bool readyFlag = false;
```

Naiwne podejście polega na synchronizacji ustawienia zmiennej `data` w jednym wątku i „skonsumowaniu" zmiennej `data` w innym wątku. Zatem w wątku dostarczającym dane znajdą się wywołania:

```
data = 42;
readyFlag = true;
```

[16] Ten przykład pochodzi z wielu artykułów ze strony internetowej Bartosza Milewskiego „Programming Cafe" (szczegółowe informacje można znaleźć w [Milewski:Multicore] oraz [Milewski:Atomics]).

natomiast w wątku konsumującym dane wywołania:

```
while (!readyFlag) { // pętla do chwili, kiedy dane będą gotowe
    ;
}
foo(data);
```

Bez znajomości żadnych innych szczegółów każdy programista na pierwszy rzut oka przypuszczałby, że drugi wątek wywołuje funkcję `foo()` w momencie, gdy zmienna `data` ma wartość `42`. Zakładałby przy tym, że wywołanie `foo()` jest osiągalne tylko wtedy, gdy zmienna `readyFlag` ma wartość `true`, a to może się stać tylko wtedy, gdy pierwszy wątek przypisał wartość `42` do zmiennej `data`, ponieważ to się dzieje, zanim zmienna `readyFlag` przyjmie wartość `true`.

Jednak w rzeczywistości może być inaczej. W drugim wątku zmienna `data` może mieć wartość *sprzed* momentu, gdy pierwszy wątek przypisał do niej `42` (lub nawet dowolną inną wartość, ponieważ w tym momencie przypisywanie wartości `42` mogło się jeszcze nie zakończyć).

Oznacza to, że kompilator i (lub) sprzęt mógł zmienić kolejność instrukcji, tak że w rezultacie została wykonana następująca sekwencja instrukcji:

```
readyFlag = true;
data = 42;
```

Ogólnie rzecz biorąc, taka zmiana kolejności jest dozwolona ze względu na reguły języka C++, które wymagają jedynie, aby *obserwowane zachowanie wewnątrz wątku* wygenerowanego kodu było poprawne. Dla zachowania pierwszego wątku nie ma znaczenia, czy pierwszy wątek zmodyfikuje zmienną `readyFlag`, czy `data`. Z punktu widzenia tego wątku zmienne te są od siebie niezależne. Zatem zmiana kolejności instrukcji jest dozwolona, pod warunkiem że obserwowany efekt na zewnątrz pojedynczego wątku będzie taki sam.

Z tego samego powodu nawet drugi wątek może zmienić kolejność instrukcji, pod warunkiem że nie wpłynie to na zachowanie tego wątku:

```
foo(data);
while (!readyFlag) { // pętla do chwili, kiedy dane będą gotowe
    ;
}
```

Zwróćmy uwagę, że taka zmiana kolejności instrukcji może wpłynąć na obserwowane zachowanie wątku, jeśli funkcja `foo()` zgłosi wyjątek. Tak więc o tym, czy zmiana kolejności instrukcji jest dozwolona, decydują szczegóły, ale ogólnie rzecz biorąc, problem istnieje.

Powodem zezwolenia na takie modyfikacje jest to, że domyślnie kompilatory C++ powinny generować kod, który jest w wysokim stopniu zoptymalizowany, a niektóre optymalizacje mogą powodować zmianę kolejności instrukcji. Domyślnie te optymalizacje nie muszą dbać o ewentualne inne wątki, co sprawia, że ich realizacja staje się łatwiejsza, ponieważ wystarczają lokalne analizy.

18.4.4. Mechanizmy pozwalające na rozwiązanie problemów

Aby rozwiązać trzy główne problemy współbieżnego dostępu do danych, potrzebujemy następujących mechanizmów:

- **Niepodzielność**. Oznacza ona, że odczyt lub zapis do zmiennej lub wykonanie sekwencji instrukcji dzieje się w sposób wyłączny i bez żadnych przerw, dzięki czemu jeden wątek nie może czytać stanów pośrednich spowodowanych przez inny wątek.
- **Kolejność**. Potrzebujemy mechanizmów, które gwarantują kolejność wykonania określonych instrukcji bądź grup określonych instrukcji.

Biblioteka standardowa C++ dostarcza różnych sposobów obsługi tych mechanizmów, dzięki czemu programy mogą korzystać z dodatkowych gwarancji dotyczących współbieżnego dostępu do danych:

- Można skorzystać z *futur* (patrz podrozdział 18.1) i *promes* (patrz punkt 18.2.2), które gwarantują zarówno niepodzielność, jak i kolejność: ustawienie *wyniku* (zwracanej wartości lub wyjątku) *współdzielonego stanu* jest gwarantowane przed przetworzeniem tego wyniku, co implikuje, że dostęp do odczytu i zapisu nie zachodzi równocześnie.
- Można skorzystać z *muteksów* i *blokad* (patrz podrozdział 18.5) w celu obsługi *sekcji krytycznych* lub *stref chronionych*, w których możemy zagwarantować wyłączny dostęp. Dzięki temu nic nie może się zdarzyć pomiędzy sprawdzeniem warunku a operacją bazującą na tym warunku. Blokady zapewniają niepodzielność poprzez zablokowanie dostępu przy użyciu drugiej blokady ustanowionej do chwili, kiedy pierwsza blokada do tego samego zasobu zostanie zwolniona. Mówiąc dokładniej: C++ gwarantuje, że zwolnienie obiektu blokady uzyskanej przez jeden z wątków odbędzie się, zanim innemu wątkowi uda się uzyskać ten sam obiekt blokady. Jednakże jeśli dwa wątki używają dostępu do danych chronionego przez blokadę, to kolejność, w jakiej korzystają z danych, może się różnić pomiędzy kolejnymi uruchomieniami.
- Możemy skorzystać ze *zmiennych warunkowych* (patrz podrozdział 18.6), aby umożliwić jednemu wątkowi oczekiwanie na to, aż pewien predykat kontrolowany przez inny wątek stanie się prawdziwy. Pomaga to zarządzać kolejnością działania wielu wątków. Można bowiem zezwolić jednemu lub większej liczbie wątków na przetwarzanie danych lub statusu dostarczanych przez jeden bądź więcej innych wątków[17].
- Można skorzystać z *atomowych typów danych* (patrz podrozdział 18.7). W ten sposób można zapewnić niepodzielność każdego dostępu do zmiennej bądź obiektu. Jednocześnie kolejność operacji na atomowych typach danych pozostaje stabilna.

[17] Eksperci w dziedzinie współbieżności nie uznają zmiennych warunkowych za narzędzie rozwiązywania problemów z równoległym dostępem do danych, ponieważ zmienne warunkowe w większym stopniu są narzędziem poprawy wydajności niż zapewnienia poprawności działania.

- Można skorzystać z *niskopoziomowego interfejsu atomowych typów danych* (patrz punkt 18.7.4). Za jego pomocą eksperci w dziedzinie współbieżności mogą złagodzić reguły kolejności atomowych instrukcji lub zastosować manualne bariery dostępu do pamięci (tzw. *płoty*).

Ogólnie rzecz biorąc, powyższa lista została posortowana od mechanizmów najwyższego poziomu do własności niskiego poziomu. Mechanizmy wysokopoziomowe takie jak futury i promesy albo muteksy i blokady są łatwe do użycia i wiążą się z niewielkim ryzykiem. Mechanizmy niskopoziomowe, takie jak atomowe typy danych, a zwłaszcza ich niskopoziomowy interfejs, mogą zapewnić lepszą wydajność, ponieważ ich użycie wiąże się z mniejszymi opóźnieniami, a tym samym są bardziej skalowalne, ale ryzyko ich nieprawidłowego użycia znacznie wzrasta. Niemniej jednak własności niskiego poziomu czasami dostarczają prostych rozwiązań dla specyficznych problemów wysokiego poziomu.

Dzięki zastosowaniu atomowych typów danych zmierzamy w kierunku *programowania bez blokad* (ang. *lock-free programming*), z którym miewają problemy nawet eksperci w dziedzinie współbieżności. Cytując Herba Suttera z książki [*Sutter:LockFree*]: „[kod bez blokad jest] trudny nawet dla ekspertów. Z łatwością można napisać kod bez blokad, który sprawia wrażenie działającego, ale bardzo trudno jest napisać kod bez blokad, który jest prawidłowy i działa wydajnie. Nawet w dobrych czasopismach i uznanych tytułach opublikowano sporo kodu bez blokad, który w istocie zawierał subtelne błędy i wymagał korekt".

Słowo kluczowe volatile a współbieżność

Zwróćmy uwagę, że nie wspomniałem o słowie kluczowym `volatile` jako o mechanizmie umożliwiającym współbieżny dostęp do danych. Można jednak było tego oczekiwać z następujących powodów:

- `volatile` jest znanym słowem kluczowym języka C++ służącym do wyeliminowania nadmiernej optymalizacji;
- w Javie słowo kluczowe `volatile` daje pewne gwarancje dotyczące niepodzielności i kolejności wykonywanych operacji.

W języku C++ słowo kluczowe `volatile` określa „tylko" to, że dostęp do zewnętrznych zasobów, na przykład do współdzielonej pamięci, nie powinien być optymalizowany. Na przykład bez słowa kluczowego `volatile` kompilator mógłby wyeliminować redundantne operacje ładowania tego samego segmentu współdzielonej pamięci ze względu na brak zaobserwowanych modyfikacji tego segmentu w całym programie. Ale w C++ słowo kluczowe `volatile` nie zapewnia ani niepodzielności, ani konkretnej kolejności[18]. Zatem semantyka słowa kluczowego `volatile` jest inna w języku C++, a inna w języku Java.

Warto zajrzeć również do punktu 18.5.1, gdzie można znaleźć dyskusję o tym, dlaczego słowo kluczowe `volatile` zazwyczaj nie jest wymagane, kiedy do czytania danych w pętli są wykorzystywane muteksy.

[18] Dziękuję Scottowi Meyersowi za zwrócenie mi na to uwagi.

18.5. Muteksy i blokady

Muteks (od ang. *mutual exclusion* — wzajemne wykluczenie) jest obiektem, który pomaga kontrolować współbieżny dostęp do zasobu poprzez zapewnienie wyłącznego dostępu do niego. Zasób może być obiektem albo kombinacją wielu obiektów. Aby uzyskać wyłączny dostęp do zasobu, odpowiedni wątek blokuje muteks. Zapobiega to blokowaniu tego muteksu przez inne wątki do czasu, aż pierwszy wątek odblokuje muteks.

18.5.1. Wykorzystywanie muteksów i blokad

Załóżmy, że chcemy zabezpieczyć się przed współbieżnym dostępem do obiektu `val`, który jest używany w różnych miejscach:

```
int val;
```

Naiwne podejście do synchronizacji tego współbieżnego dostępu to zdefiniowanie muteksu wykorzystywanego do umożliwienia wyłącznego dostępu i zarządzania nim:

```
int val;
std::mutex valMutex; // zarządzanie wyłącznym dostępem do val
```

Następnie w każdej operacji dostępu trzeba zablokować ten muteks w celu uzyskania wyłącznego dostępu. Na przykład jeden z wątków może być zaprogramowany w następujący sposób (zwróćmy uwagę, że jest to słabe rozwiązanie, które poprawimy):

```
valMutex.lock(); // żądanie wyłącznego dostępu do val
if (val >= 0) {
    f(val); // val ma dodatnią wartość
}
else {
f(-val);     // przekazanie zanegowanej wartości val
}
valMutex.unlock(); // zwolnienie wyłącznego dostępu do obiektu val
```

Inny wątek może próbować uzyskać dostęp do tego samego zasobu w następujący sposób:

```
valMutex.lock(); // żądanie wyłącznego dostępu do val
++val;
valMutex.unlock(); // zwolnienie wyłącznego dostępu do obiektu val
```

Istotne znaczenie ma to, aby we wszystkich miejscach, w których możliwy jest współbieżny dostęp, korzystać z tego samego muteksu. Dotyczy to zarówno dostępu do odczytu, jak i do zapisu.

To proste podejście może się jednak nieco skomplikować. Na przykład powinniśmy zapewnić, aby każdy wyjątek, który kończy wyłączny dostęp, powodował także odblokowanie odpowiedniego muteksu. W przeciwnym przypadku

zasób mógłby pozostać zablokowany na zawsze. Możliwe są również scenariusze zakleszczeń, kiedy dwa wątki oczekują na zablokowanie innego wątku, zanim zwolnią własną blokadę.

Standardowa biblioteka C++ próbuje obsłużyć te problemy, ale nie może ich wszystkich rozwiązać. Na przykład: aby poradzić sobie z wyjątkami, nie można blokować i odblokowywać muteksów samodzielnie. Należy zastosować zasadę RAII (*Resource Acquisition Is Initialization*), według której dostęp do zasobu uzyskujemy w konstruktorze. Dzięki temu destruktor, który jest wywoływany zawsze (nawet wtedy, gdy wyjątek spowoduje zakończenie życia obiektu), automatycznie zwalnia zasób. Z tego powodu standardowa biblioteka C++ dostarcza klasy `std::lock_guard`:

```
int val;
std::mutex valMutex; // zarządzanie wyłącznym dostępem do val
...
std::lock_guard<std::mutex> lg(valMutex); // założenie blokady i automatyczne odblokowanie
    if (val >= 0) {
f(val); // val ma dodatnią wartość
}
else {
    f(-val);     // przekazanie zanegowanej wartości val
}
```

Należy jednak pamiętać, że blokady powinny ograniczać się do możliwie jak najkrótszego czasu, ponieważ uniemożliwiają one innemu kodowi równoległe działanie. Ponieważ destruktor zwalnia blokadę, możemy jawnie ująć kod w nawiasy klamrowe, aby blokada została zwolniona, zanim będą uruchomione dalsze instrukcje:

```
int val;
std::mutex valMutex; // zarządzanie wyłącznym dostępem do val
...
{
    std::lock_guard<std::mutex> lg(valMutex); // założenie blokady i automatyczne odblokowanie
    if (val >= 0) {
        f(val); // val ma dodatnią wartość
    }
    else {
        f(-val);     // przekazanie zanegowanej wartości val
    }
} // zapewnienie zdjęcia blokady w tym miejscu
...
```

lub po prostu:

```
...
{
    std::lock_guard<std::mutex> lg(valMutex); // założenie blokady i automatyczne odblokowanie
    ++val;
} // zapewnienie zdjęcia blokady w tym miejscu
...
```

To jest tylko pierwszy, prosty przykład. Można jednak zauważyć, że zagadnienie to łatwo może stać się dość złożone. Tak jak zwykle programiści muszą wiedzieć, co robi ich program — zwłaszcza działający współbieżnie. Poza tym dostępne są różne muteksy i blokady. Zostały one omówione w kolejnych podpunktach.

Pierwszy kompletny przykład użycia muteksu i blokady

Przyjrzyjmy się pierwszemu kompletnemu przykładowi:

```
// concurrency/mutex1.cpp

#include <future>
#include <mutex>
#include <iostream>
#include <string>

std::mutex printMutex; // zarządzanie zsynchronizowanym wyjściem z wykorzystaniem funkcji
print()

void print (const std::string& s)
{
    std::lock_guard<std::mutex> l(printMutex);
    for (char c : s) {
        std::cout.put(c);
    }
    std::cout << std::endl;
}

int main()
{
    auto f1 = std::async (std::launch::async,
                          print, "Pozdrowienia z pierwszego wątku");
    auto f2 = std::async (std::launch::async,
                          print, "Pozdrowienia z drugiego wątku");
    print ("Pozdrowienia z głównego wątku");
}
```

W tym przykładzie funkcja `print()` zapisuje wszystkie znaki przekazanego łańcucha do standardowego wyjścia. Zatem bez blokady wynik może mieć następującą postać[19]:

```
PPozdPozdrowienia z drugiego wątku
ozdrowienia z pierwszego wątku
rowienia z głównego wątku
```

albo:

```
PozdrowieniaPozdrowienia zPozdrowienia z pierwdrugiego drszego zugie głównegowątkuego
wąt
ku
```

[19] Przeplatane znaki w przypadku wykonywanych współbieżnie operacji zapisu wynikają stąd, że każdy znak jest wyświetlany osobno za pomocą oddzielnego wywołania funkcji `put()`. W przypadku zapisywania całego łańcucha jako całości implementacje często nie przeplatają znaków, ale nawet takie zachowanie nie jest gwarantowane.

Aby zsynchronizować wyjście w taki sposób, żeby każde wywołanie funkcji `print()`wyświetlało wyłącznie własne znaki, wprowadzimy muteks dla operacji wyświetlania oraz blokadę-strażnika, która blokuje odpowiednią chronioną sekcję:

```
std::mutex printMutex; // zarządzanie zsynchronizowanym wyjściem z wykorzystaniem funkcji
print()
...
void print (const std::string& s)
{
    std::lock_guard<std::mutex> l(printMutex);
    ...
}
```

Teraz wynik działania programu po prostu jest następujący:

```
Pozdrowienia z pierwszego wątku
Pozdrowienia z głównego wątku
Pozdrowienia z drugiego wątku
```

Taki wynik jest również możliwy (ale nie jest gwarantowany) w przypadku, gdy nie są używane blokady.

W tym przykładzie wywołanie `lock()` dla muteksu zrealizowane przez konstruktor blokady-strażnika blokuje się, jeśli zasób został już zablokowany. Blokada trwa do momentu, aż sekcja chroniona na nowo stanie się dostępna. Jednak kolejność blokad w dalszym ciągu jest niezdefiniowana. W związku z tym komunikaty z trzech wątków w dalszym ciągu mogą pojawić się w dowolnej kolejności.

Blokady rekurencyjne

Czasami występuje potrzeba zastosowania blokad rekurencyjnych. Typowymi przykładami takich sytuacji są obiekty aktywne lub monitory, które zawierają muteks i ustanawiają blokadę w każdej metodzie publicznej w celu zabezpieczenia przed sytuacjami wyścigu, które niszczą wewnętrzny stan obiektu. Na przykład interfejs dostępu do bazy danych może mieć następującą postać:

```
class DatabaseAccess
{
    private:
        std::mutex dbMutex;
        ... // stan dostępu do bazy danych
    public:
        void createTable (...)
        {
            std::lock_guard<std::mutex> lg(dbMutex);
            ...
    }
    void insertData (...)
{
        std::lock_guard<std::mutex> lg(dbMutex);
        ...
    }
    ...
};
```

Gdy wprowadzimy składową funkcję publiczną, która może wywoływać inne składowe funkcje publiczne, może to stać się skomplikowane:

```
void createTableAndInsertData (...)
{
    std::lock_guard<std::mutex> lg(dbMutex);
    ...
    createTable(...); // BŁĄD: zakleszczenie, ponieważ dbMutex jest ponownie zablokowany
}
```

Wywołanie funkcji `createTableAndInsertData()` spowoduje zakleszczenie, ponieważ po zablokowaniu muteksu `dbMutex` wywołanie funkcji `createTable()` spróbuje zablokować muteks `dbMutex` ponownie i funkcja zablokuje się do czasu, gdy blokada muteksu `dbMutex` zostanie zwolniona. To jednak nigdy nie nastąpi, ponieważ funkcja `createTableAndInsertData()` jest zablokowana do czasu zakończenia działania funkcji `createTable()`.

Biblioteka standardowa C++ dopuszcza możliwość zgłoszenia wyjątku `std::system_error` po raz drugi (patrz punkt 4.3.1) z kodem błędu `resource_deadlock_would_occur` (patrz punkt 4.3.2), o ile platforma zdoła wykryć takie zakleszczenie. Takie zachowanie nie jest jednak wymagane i często nie następuje.

Użycie muteksu rekurencyjnego (`recursive_mutex`) pozwala poradzić sobie z tym problemem. Taki muteks umożliwia zastosowanie wielu blokad przez ten sam wątek i zwolnienie blokady w momencie, kiedy zostanie wywołana ostatnia funkcja `unlock()`:

```
class DatabaseAccess
{
    private:
        std::recursive_mutex dbMutex;
        ... // stan dostępu do bazy danych
   public:
    void insertData (...)
    {
        std::lock_guard<std::recursive_mutex> lg(dbMutex);
        ...
    }
    void insertData (...)
    {
        std::lock_guard<std::recursive_mutex> lg(dbMutex);
        ...
    }
    void createTableAndinsertData (...)
    {
        std::lock_guard<std::recursive_mutex> lg(dbMutex);
    ...
        createTable(...); // OK: nie ma zakleszczenia
    }
    ...
};
```

Próby uzyskania blokady i blokady czasowe

Czasami program chce zdobyć blokadę, ale nie chce blokować się na zawsze, kiedy zdobycie tej blokady nie jest możliwe. Do obsłużenia tej sytuacji muteksy oferują składową funkcję `try_lock()`, która *próbuje* uzyskać blokadę. Jeśli próba się powiedzie, funkcja zwraca `true`, a jeśli nie — zwraca `false`.

Aby w dalszym ciągu móc użyć blokady `lock_guard` tak, by wyjście z bieżącego zakresu nie spowodowało odblokowania muteksu, możemy przekazać dodatkowy argument `adopt_lock` do konstruktora muteksu:

```
std::mutex m;
// próba uzyskania blokady i wykonania innych działań w czasie, gdy jest to niemożliwe
while (m.try_lock() == false) {
    doSomeOtherStuff();
}
std::lock_guard<std::mutex> lg(m,std::adopt_lock);
...
```

Zwróćmy uwagę, że funkcja `try_lock()` może się nie powieść (zwrócić `false`), nawet gdy blokada nie jest zajęta[20].

Aby oczekiwać na blokadę tylko przez określony czas, możemy skorzystać z muteksu czasowego. Specjalne klasy muteksów `std::timed_mutex` i `std::recursive_timed_mutex` dodatkowo pozwalają na wywoływanie funkcji `try_lock_for()` lub `try_lock_until()`, które czekają przez co najwyżej określony przedział czasu lub do określonego punktu w czasie. To może pomóc na przykład wtedy, gdy mamy wymagania dotyczące pracy w czasie rzeczywistym lub chcemy uniknąć możliwych sytuacji zakleszczeń. Na przykład:

```
std::timed_mutex m;
// próba uzyskania blokady przez jedną sekundę
if (m.try_lock_for(std::chrono::seconds(1))) {
    std::lock_guard<std::timed_mutex> lg(m,std::adopt_lock);
    ...
}
else {
    couldNotGetTheLock();
}
```

Zwróćmy uwagę, że funkcje `try_lock_for()` i `try_lock_until()` zazwyczaj działają inaczej w przypadku korekt czasu systemowego (szczegółowe informacje można znaleźć w punkcie 5.7.5).

Korzystanie z wielu blokad

Zazwyczaj wątek powinien blokować za każdym razem tylko jeden muteks. Czasami jednak konieczne jest zablokowanie więcej niż jednego muteksu (na przykład w celu przesłania danych od jednego chronionego zasobu do innego). W takim

[20] Takie zachowanie wynika z powodów szeregowania pamięci, ale nie jest ono powszechnie znane. Dziękuję Hansowi Boehmowi i Bartoszowi Milewskiemu za zwrócenie uwagi na ten problem.

przypadku posługiwanie się mechanizmami blokad zaprezentowanymi do tej pory może okazać się skomplikowane i ryzykowne: możemy uzyskać pierwszą blokadę, ale nie uzyskać drugiej. Może też dojść do zakleszczenia, jeśli założymy blokady w niewłaściwej kolejności.

Z tego względu standardowa biblioteka C++ dostarcza funkcji pomocniczych pozwalających na próby blokowania wielu muteksów. Na przykład:

```
std::mutex m1;
std::mutex m2;
...
{
    std::lock (m1, m2); // zablokowanie obu muteksów (albo żadnego, jeśli to nie jest możliwe)
    std::lock_guard<std::mutex> lockM1(m1,std::adopt_lock);
    std::lock_guard<std::mutex> lockM2(m2,std::adopt_lock);
    ...
} // automatyczne odblokowanie wszystkich muteksów
```

Globalna funkcja `std::lock()` blokuje wszystkie muteksy przekazane w roli argumentów. Funkcja blokuje się do czasu, aż wszystkie muteksy będą zablokowane, albo dopóki nie zostanie zgłoszony wyjątek. W tym drugim przypadku muteksy, które zostały pomyślnie zablokowane, są odblokowywane. Tak jak zwykle po pomyślnym ustanowieniu blokady można i należy użyć strażnika blokady zainicjowanego za pomocą obiektu `adopt_lock` przekazanego w roli drugiego argumentu, aby zapewnić odblokowanie muteksów po opuszczeniu zakresu. Zwróćmy uwagę, że ta metoda `lock()` dostarcza mechanizmu unikania zakleszczeń, co jednak oznacza, że kolejność blokowania dla wielokrotnej blokady jest niezdefiniowana.

W taki sam sposób możemy *próbować* uzyskać wiele blokad bez wstrzymania działania, jeśli nie wszystkie blokady są dostępne. Globalna funkcja `std::try_lock()` zwraca –1, jeśli były możliwe wszystkie blokady. Jeśli nie, to zwracana wartość jest indeksem (począwszy od zera) oznaczającym pierwszą blokadę, która okazała się nieskuteczna. W takim przypadku wszystkie blokady, które udało się ustanowić, zostaną zwolnione. Na przykład:

```
std::mutex m1;
std::mutex m2;

int idx = std::try_lock (m1, m2); // próba zablokowania obu muteksów
if (idx < 0) { // ustanowienie obydwu blokad zakończyło się sukcesem
        std::lock_guard<std::mutex> lockM1(m1,std::adopt_lock);
        std::lock_guard<std::mutex> lockM2(m2,std::adopt_lock);
        ...
} // automatyczne odblokowanie wszystkich muteksów
else {
    // idx jest indeksem (począwszy od zera) oznaczającym pierwszą nieskuteczną blokadę
    std::cerr << "nie można zablokować muteksu m" << idx+1 << std::endl;
}
```

Zwróćmy uwagę, że ta funkcja `try_lock()` nie zapewnia mechanizmu zapobiegania zakleszczeniom. Zamiast tego gwarantuje, że próby stosowania blokad będą wykonywane w kolejności przekazywanych argumentów.

Zwróćmy również uwagę na to, że wywołanie `lock()` lub `try_lock()` bez zastosowania strażnika `adopt_lock` zazwyczaj nie jest pożądanym działaniem. Chociaż kod sprawia wrażenie, jakby automatycznie zwalniał blokadę po wyjściu z zakresu, to w rzeczywistości tak nie jest. Muteksy pozostaną zablokowane:

```
std::mutex m1;
std::mutex m2;
...
{
    std::lock (m1, m2); // zablokowanie obu muteksów (albo żadnego, jeśli to nie jest możliwe)
    // blokady nie zostały zaadaptowane
    ...
}
... // OOPS: Muteksy są nadal zablokowane.
```

Klasa unique_lock

Oprócz klasy `lock_guard<>` standardowa biblioteka C++ oferuje klasę `unique_lock<>`, która jest o wiele bardziej elastyczna do obsługi blokad dla muteksów. Klasa `unique_lock<>` zapewnia taki sam interfejs co klasa `lock_guard<>` oraz dodatkowo możliwość jawnego programowania, kiedy i w jaki sposób należy zablokować bądź odblokować związany z nią muteks. Zatem ten obiekt blokady może, ale nie musi *posiadać* muteksu. Pod tym względem obiekt ten różni się od obiektu `lock_guard<>`, który przez cały czas swojego życia posiada zablokowany obiekt[21]. Dodatkowo w przypadku unikatowych blokad możemy odpytać, czy muteks jest w danym momencie zablokowany, poprzez wywołanie funkcji `owns_lock()` lub `operator bool()`.

Główną zaletą tej klasy nadal jest to, że kiedy muteks jest zablokowany w momencie niszczenia obiektu, to destruktor automatycznie wywołuje dla niego funkcję `unlock()`. Jeśli żaden muteks nie jest zablokowany, destruktor nie robi niczego.

W porównaniu z klasą `lock_guard` klasa `unique_lock` dostarcza następujących, uzupełniających konstruktorów:

- konstruktor pozwalający na przekazanie obiektu `try_to_lock` w celu wykonania nieblokującej próby ustanowienia blokady muteksu:

```
std::unique_lock<std::mutex> lock(mutex,std::try_to_lock);
...
if (lock) { // jeśli próba ustanowienia blokady była udana
    ...
}
```

- konstruktor umożliwiający przekazanie przedziału czasu lub punktu w czasie w celu wykonywania prób ustanowienia blokady przez wskazany czas:

```
std::unique_lock<std::timed_mutex> lock(mutex,
                                        std::chrono::seconds(1));
...
```

[21] Nazwa *unique lock* (unikatowa blokada) wyjaśnia, skąd się bierze takie zachowanie. Tak jak w przypadku unikatowych wskaźników (patrz punkt 5.2.5) można przemieszczać blokady pomiędzy zakresami, ale mamy gwarancję, że tylko jedna blokada w określonym czasie posiada muteks.

- konstruktor pozwalający na przekazanie obiektu `defer_lock` w celu zainicjowania blokady bez blokowania muteksu (jeszcze):

```
std::unique_lock<std::mutex> lock(mutex,std::defer_lock);
...
lock.lock(); // lub (czasowa blokada) try_lock()
...
```

Flagę `defer_lock` można wykorzystać na przykład do stworzenia jednej lub większej liczby blokad i zablokowania ich później:

```
std::mutex m1;
std::mutex m2;

std::unique_lock<std::mutex> lockM1(m1,std::defer_lock);
std::unique_lock<std::mutex> lockM2(m2,std::defer_lock);
...
std::lock (m1, m2); // zablokowanie obu muteksów (albo żadnego, jeśli to nie jest możliwe)
```

Dodatkowo klasa `unique_lock` umożliwia zwolnienie posiadanego przez nią muteksu (`release()`) lub przekazanie własności muteksu do innej blokady. Szczegółowe informacje na ten temat można znaleźć w punkcie 18.5.2.

Korzystając zarówno z obiektu `lock_guard`, jak i `unique_lock`, możemy zaimplementować naiwny przykład, w którym jeden wątek czeka na inny poprzez odpytywanie flagi gotowości:

```
#include <mutex>
...
bool readyFlag;
std::mutex readyFlagMutex;

void thread1()
{
    // wykonanie operacji, których wątek thread2 potrzebuje do przygotowania
    ...
    std::lock_guard<std::mutex> lg(readyFlagMutex);
    readyFlag = true;
}

void thread2()
{
    // oczekiwanie, aż flaga gotowości będzie miała wartość true (wątek thread1 jest zakończony)
    {
        std::unique_lock<std::mutex> ul(readyFlagMutex);
        while (!readyFlag) {
            ul.unlock();
            std::this_thread::yield(); // wskazówka przekazania sterowania do następnego wątku
            std::this_thread::sleep_for(std::chrono::milliseconds(100));
            ul.lock();
        }
    } // zwolnienie blokady

    // wykonanie operacji, które powinny być wykonane po przygotowaniu działań przez wątek thread1
    ...
}
```

Oto dwie uwagi dotyczące typowych pytań, jakie mogą powstać po analizie powyższego kodu:

- Jeżeli zastanawiasz się, dlaczego użyliśmy muteksu do zarządzania dostępem do odczytu i zapisu flagi `readyFlag`, przypomnij sobie regułę wprowadzoną na początku tego rozdziału: każdy współbieżny dostęp, który obejmuje co najmniej jedną operację zapisu, powinien być synchronizowany. Szczegółowe informacje na ten temat można znaleźć w podrozdziale 18.4 oraz w podrozdziale 18.7.
- Jeśli zastanawiasz się, dlaczego nie jest potrzebne użycie słowa kluczowego `volatile` do zadeklarowania flagi `readyFlag` w celu uniknięcia optymalizacji wielokrotnych prób odczytu tej flagi w wątku `thread2()`, zwróć uwagę na następujący fakt: próby odczytu flagi `readyFlag` występują w *sekcji krytycznej*, zdefiniowanej pomiędzy ustanowieniem a zwolnieniem blokady. Taki kod nie jest optymalizowany w taki sposób, aby operacje odczytu (bądź zapisu) były przemieszczane poza sekcję krytyczną. Zatem operacje odczytu flagi `readyFlag` muszą odbywać się w następujących miejscach:
  - na początku pętli, pomiędzy deklaracją blokady `ul` a pierwszym wywołaniem funkcji `unlock()`;
  - wewnątrz pętli, pomiędzy każdym wywołaniem funkcji `lock()` i `unlock()`;
  - na końcu pętli, pomiędzy ostatnim wywołaniem funkcji `lock()` a zniszczeniem obiektu `ul`, które powoduje odblokowanie muteksu, jeśli jest on zablokowany.

Niemniej jednak takie *odpytywanie* o spełnienie warunku zazwyczaj nie jest dobrym rozwiązaniem. Lepszym rozwiązaniem jest skorzystanie ze *zmiennych warunkowych*. Szczegółowe informacje na ten temat można znaleźć w punkcie 18.6.1.

18.5.2. Muteksy i blokady w szczegółach

Muteksy w szczegółach

Biblioteka standardowa C++ dostarcza następujących klas muteksów (patrz tabela 18.6):

- Klasa `std::mutex` dostarcza prostego muteksu, który można zablokować tylko raz i tylko przez jeden wątek. Jeśli muteks jest zablokowany, każde kolejne wywołanie funkcji `lock()` zablokuje się do czasu, aż muteks stanie się ponownie dostępny. Z kolei wywołanie `try_lock()` zwróci `false`.
- Klasa `std::recursive_mutex` definiuje muteks, który pozwala na ustanawianie wielu blokad jednocześnie przez ten sam wątek. Typowym zastosowaniem takiego muteksu jest sytuacja, gdy funkcja uzyskuje blokadę i wewnętrznie wywołuje inną funkcję, która uzyskuje tę samą blokadę jeszcze raz.

TABELA 18.6. Przegląd muteksów i ich możliwości

Operacja	mutex	recursive_mutex	timed_mutex	recursive_timed_mutex
`lock()`	Zdobywa muteks (blokuje się, jeśli muteks nie jest dostępny).			
`try_lock()`	Zdobywa muteks (zwraca `false`, jeśli muteks nie jest dostępny).			
`unlock()`	Odblokowuje zablokowany muteks.			
`try_lock_for()`	–	–	Próbuje uzyskać muteks na określony czas.	
`try_lock_until()`	–	–	Próbuje uzyskać muteks do określonego punktu w czasie.	
wielokrotne blokady	Nie	Tak (ten sam wątek)	Nie	Tak (ten sam wątek)

- Klasa `std::timed_mutex` definiuje prosty muteks, który dodatkowo umożliwia przekazanie przedziału czasu lub punktu w czasie określającego, jak długo muteks próbuje uzyskać blokadę. Do tego celu służą metody `try_lock_for()` i `try_lock_until()`.
- Klasa `std::recursive_timed_mutex` definiuje muteks, który pozwala na ustanawianie wielu blokad jednocześnie przez ten sam wątek oraz opcjonalnie pozwala na przekazywanie limitu czasu.

Operacje dostępne dla muteksów zestawiono w tabeli 18.7.

TABELA 18.7. Operacje klas muteksów

Operacja	Efekt
mutex *m*	Domyślny konstruktor. Tworzy niezablokowany muteks.
m.~***mutex***()	Niszczy muteks (nie może być zablokowany).
m.lock()	Blokuje muteks (funkcja blokuje interfejs na czas ustanawiania blokady, zwraca błąd, jeśli muteks jest zablokowany i nie jest rekurencyjny).
m.try_lock()	Próbuje zablokować muteks (zwraca `true`, jeśli próba zakończyła się sukcesem).
m.try_lock_for(*dur*)	Próba uzyskania blokady przez czas *dur* (zwraca `true`, jeśli próba zakończyła się sukcesem).
m.try_lock_until(*tp*)	Próbuje zablokować muteks do punktu w czasie *tp* (zwraca `true`, jeśli próba zakończyła się sukcesem).
m.unlock()	Odblokowuje muteks (jeśli muteks nie jest zablokowany, działanie funkcji jest niezdefiniowane).
m.native_handle()	Zwraca specyficzny dla platformy typ `native_handle_type` w przypadku nieprzenośnych rozszerzeń.

Funkcja `lock()` może zgłosić wyjątek `std::system_error` (patrz punkt 4.3.1) z następującymi kodami błędów (patrz punkt 4.3.2):

- `operation_not_permitted`, jeśli wątek nie ma uprawnień do wykonania operacji;
- `resource_deadlock_would_occur`, jeśli platforma wykryje możliwość wystąpienia zakleszczenia;
- `device_or_resource_busy`, jeśli muteks jest już zablokowany i ustanowienie blokady nie jest możliwe.

Zachowanie programu będzie niezdefiniowane, jeśli nastąpi zwolnienie blokady obiektu muteksu, który do programu nie należy, zniszczenie obiektu muteksu należącego do dowolnego wątku, albo jeśli nastąpi zakończenie działania wątku, jeśli posiada on obiekt muteksu.

Zwróćmy uwagę, że funkcje `try_lock_for()` i `try_lock_until()` zazwyczaj działają inaczej w przypadku korekt czasu systemowego (szczegółowe informacje można znaleźć w punkcie 5.7.5).

Klasa lock_guard w szczegółach

Klasa `std::lock_guard`, którą po raz pierwszy zaprezentowano w punkcie 18.5.1, dostarcza niewielkiego interfejsu, którego zadaniem jest zwolnienie muteksu zawsze wtedy, gdy opuści zakres (patrz tabela 18.8). W czasie swojego życia obiekt jest zawsze powiązany z blokadą, która może być zażądana jawnie lub zaadaptowana w czasie konstruowania obiektu.

TABELA 18.8. Operacje dostępne dla klasy lock_guard

Operacja	Efekt
`lock_guard` *lg*(*m*)	Tworzy strażnika blokady dla muteksu *m* i blokuje muteks.
`lock_guard` *lg*(*m*,`adopt_lock`)	Tworzy strażnika blokady dla zablokowanego wcześniej muteksu *m*.
lg.~`lock_guard`()	Odblokowuje muteks i niszczy strażnika blokady.

Klasa unique_lock w szczegółach

Klasa `std::unique_lock`, którą po raz pierwszy zaprezentowano w punkcie 18.5.1, dostarcza strażnika blokady dla muteksu, który niekoniecznie musi być zablokowany (posiadany przez wątek). Klasa dostarcza interfejsu zaprezentowanego w tabeli 18.9. Jeśli obiekt tej klasy zablokuje muteks, to w czasie działania destruktora go odblokuje. Możemy jednak jawnie zarządzać tym, czy obiekt klasy `unique_lock` ma powiązany ze sobą muteks i czy ten muteks jest zablokowany. Można również próbować zablokować muteks, przekazując limit czasu bądź nie.

Funkcja `lock()` może zgłosić wyjątek `std::system_error` (patrz punkt 4.3.1) z kodami błędów identycznymi jak w przypadku funkcji `lock()` muteksów. Jeśli unikatowa blokada nie jest ustanowiona, wywołanie funkcji `unlock()` może zgłosić wyjątek `std::system_error` z kodem błędu `operation_not_permitted`.

TABELA 18.9. Operacje klasy unique_lock

Operacja	Efekt
`unique_lock l`	Domyślny konstruktor. Tworzy blokadę, która nie jest powiązana z muteksem.
`unique_lock l(m)`	Tworzy strażnika blokady dla muteksu *m* i blokuje muteks.
`unique_lock l ↪(m,adopt_lock)`	Tworzy strażnika blokady dla zablokowanego wcześniej muteksu *m*.
`unique_lock l ↪(m,defer_lock)`	Tworzy strażnika blokady dla muteksu *m* bez jego blokowania.
`unique_lock l ↪(m,try_lock)`	Tworzy strażnika blokady dla muteksu *m* i próbuje go zablokować.
`unique_lock l(m,dur)`	Tworzy strażnika blokady dla muteksu *m* i próbuje go zablokować przez czas *dur*.
`unique_lock l(m,tp)`	Tworzy strażnika blokady dla muteksu *m* i próbuje go zablokować do punktu w czasie *tp*.
`unique_lock l(rv)`	Konstruktor przenoszący. Przenosi stan blokady z *rv* do *l* (po tej operacji z *rv* nie jest powiązany muteks).
`l.~unique_lock()`	Odblokowuje muteks, jeśli był zablokowany, i niszczy strażnika blokady.
`unique_lock l = rv`	Przeniesienie z przypisaniem. Przenosi stan blokady z *rv* do *l* (po tej operacji z *rv* nie jest powiązany muteks).
`swap(l1,l2)`	Zastępuje blokady miejscami.
`l1.swap(l2)`	Zastępuje blokady miejscami.
`l.release()`	Zwraca wskaźnik do muteksu powiązanego z blokadą i zwalnia muteks.
`l.owns_lock()`	Zwraca `true`, jeśli muteks powiązany z blokadą jest zablokowany.
`if (l)`	Sprawdza, czy muteks powiązany z blokadą jest zablokowany.
`l.mutex()`	Zwraca wskaźnik do muteksu powiązanego z blokadą.
`l.lock()`	Blokuje muteks powiązany z blokadą.
`l.try_lock()`	Próbuje zablokować muteks powiązany z blokadą (zwraca `true`, jeśli próba zakończyła się sukcesem).
`l.try_lock_for(dur)`	Próbuje zablokować muteks powiązany z blokadą przez czas *dur* (zwraca `true`, jeśli próba zakończyła się sukcesem).
`l.try_lock_until(tp)`	Próbuje zablokować muteks powiązany z blokadą do punktu w czasie *tp* (zwraca `true`, jeśli próba zakończyła się sukcesem).
`l.unlock()`	Odblokowuje muteks powiązany z blokadą.

18.5.3. Wywoływanie funkcjonalności raz dla wielu wątków

Czasami wiele wątków nie potrzebuje pewnej funkcjonalności, która powinna zostać przetworzona zawsze wtedy, gdy potrzebuje jej pewien wątek. Typowym przykładem jest leniwa inicjalizacja. Za pierwszym razem, kiedy jeden z wątków potrzebuje pewnej funkcjonalności, przetwarzamy ją (ale nie wcześniej, ponieważ chcemy zaoszczędzić czas przetwarzania w przypadku, gdyby ta funkcjonalność nie była potrzebna).

Zwykłe podejście dla środowisk jednowątkowych jest proste: flaga typu `Boolean` sygnalizuje, czy funkcjonalność już była wywoływana:

```
bool initialized = false; // flaga globalna
...
if (!initialized) { // zainicjowanie funkcjonalności, jeśli jeszcze nie była zainicjowana
    initialize();
    initialized = true;
}
```

albo:

```
static std::vector<std::string> staticData;

void foo()
{
    if (staticData.empty()) {
        staticData = initializeStaticData();
    }
    ...
}
```

Taki kod nie działa w kontekście wielowątkowym, ponieważ mogą wystąpić sytuacje wyścigu o dane w przypadku, gdy dwa lub więcej wątków sprawdza, czy jeszcze nie nastąpiła inicjalizacja, a jeśli nie, to zaczyna tę inicjalizację. W związku z tym trzeba ochronić obszar sprawdzania oraz inicjalizację na wypadek współbieżnego dostępu.

Tak jak zwykle można do tego celu użyć muteksów, ale standardowa biblioteka C++ dostarcza w tym celu specjalnego rozwiązania. Można po prostu użyć obiektu `std::once_flag` i wywołać funkcję `std::call_once` (także udostępnianą w pliku nagłówkowym `<mutex>`):

```
std::once_flag oc;              // flaga globalna
...
std::call_once(oc,initialize); // inicjalizacja, jeśli dotąd nie była przeprowadzona
```

albo:

```
static std::vector<std::string> staticData;

void foo()
{
    static std::once_flag oc;
    std::call_once(oc,[]{
```

```
                staticData = initializeStaticData();
            });
    ...
}
```

Jak można zauważyć, pierwszym argumentem przekazanym do funkcji `call_once()` musi być właściwy obiekt `once_flag`. Kolejne argumenty to typowe argumenty dla *obiektów wywoływalnych*: funkcji, funkcji składowych, obiektów funkcyjnych lub wyrażeń lambda plus opcjonalne argumenty dla wywoływanych funkcji (patrz podrozdział 4.4).

Zatem leniwa inicjalizacja obiektu w środowisku wielowątkowym może mieć następującą postać:

```
class X {
    private:
        mutable std::once_flag initDataFlag;
        void initData() const;
    public:
        data getData () const {
            std::call_once(initDataFlag,&X::initData,this);
            ...
    }
};
```

Ogólnie rzecz biorąc, można wywoływać różne funkcje dla tej samej flagi `once_flag`. Flaga jednorazowa przekazana do funkcji `call_once()` w roli pierwszego argumentu gwarantuje, że przekazana funkcjonalność będzie uruchomiona tylko raz. Jeśli więc pierwsze wywołanie zakończyło się powodzeniem, kolejne wywołania z tą samą flagą `once_flag` nie spowodują wywołania przekazanej funkcjonalności nawet wtedy, gdy ta funkcjonalność będzie inna.

Wszelkie wyjątki zgłaszane przez wywoływaną funkcjonalność są również zgłaszane przez funkcję `call_once()`. W przypadku wystąpienia wyjątku „pierwsze" wywołanie nie jest uznawane za udane, zatem kolejne wywołanie funkcji `call_once()` może spowodować uruchomienie przekazanej funkcjonalności[22].

18.6. Zmienne warunkowe

Czasami zadania realizowane przez różne wątki muszą wzajemnie na siebie czekać. Zatem czasami zachodzi potrzeba synchronizacji współbieżnych operacji z innych powodów niż dostęp do tych samych danych.

Niektórzy czytelnicy pewnie uważają, że już zaprezentowaliśmy taki mechanizm: futury (patrz podrozdział 18.1) pozwalają na zablokowanie wątku do czasu, aż inny wątek dostarczy danych albo inny wątek zakończy działanie. Jednak

[22] Standard określa również, że funkcja `call_once()` może zgłosić wyjątek `std::system_error`, jeśli argument `once_flag` nie jest już „ważny" (tzn. został zniszczony). Jednak to sformułowanie jest uważane za błąd, ponieważ przekazywanie zniszczonej flagi `once_flag` albo nie jest możliwe, albo powoduje niezdefiniowane zachowanie programu.

futura umożliwia przekazanie danych z jednego wątku do innego tylko raz. W rzeczywistości głównym przeznaczeniem futur jest obsługa wartości zwracanych przez wątki bądź zgłaszanych przez nie wyjątków.

W tym podrozdziale zaprezentujemy i omówimy zmienne warunkowe, które można wykorzystać do synchronizacji logicznych zależności w przepływie danych pomiędzy wątkami więcej niż jeden raz.

18.6.1. Przeznaczenie zmiennych warunkowych

W punkcie 18.5.1 zaprezentowaliśmy naiwny sposób realizacji mechanizmu pozwalającego na oczekiwanie jednego wątku na drugi za pomocą *flagi gotowości*, która sygnalizowała, kiedy jeden wątek przygotował dane lub dostarczył funkcjonalności dla innego wątku. To zazwyczaj oznacza, że wątek oczekujący *odpytuje* inny wątek, aby dowiedzieć się, czy są dostępne potrzebne dane lub czy jest spełniony określony warunek wstępny:

```
bool readyFlag;
std::mutex readyFlagMutex;

// oczekiwanie, aż flaga readyFlag będzie miała wartość true
{
    std::unique_lock<std::mutex> ul(readyFlagMutex);
    while (!readyFlag) {
        ul.unlock();
        std::this_thread::yield(); // wskazówka przekazania sterowania do następnego wątku
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        ul.lock();
    }
} // zwolnienie blokady
```

Niemniej jednak takie *odpytywanie* o spełnienie warunku zazwyczaj nie jest dobrym rozwiązaniem. Jak napisano w [*Williams:C++Conc*]:

> Wątek oczekujący pochłania cenny czas przetwarzania, wielokrotnie sprawdzając flagę, a kiedy zablokuje muteks, to wątek ustawiający flagę gotowości jest zablokowany. Ponadto trudno jest prawidłowo ustawić czas uśpienia: zbyt krótki czas uśpienia pomiędzy sprawdzeniami spowoduje, że wątek będzie marnotrawił czas przetwarzania na sprawdzenia, zbyt długi czas oczekiwania spowoduje, że wątek będzie uśpiony nawet wtedy, gdy zadanie, na które czeka, już się zakończyło — co wprowadza niepotrzebne opóźnienia.

Lepszym rozwiązaniem jest skorzystanie ze *zmiennych warunkowych*, które biblioteka standardowa C++ udostępnia za pośrednictwem pliku nagłówkowego `<condition_variable>`. Zmienna warunkowa jest zmienną, za pośrednictwem której wątek może obudzić jeden bądź więcej wątków oczekujących.

Ogólnie rzecz biorąc, zmienna warunkowa działa w następujący sposób:

- Należy włączyć zarówno plik nagłówkowy `<mutex>`, jak i `<condition_variable>`, aby zadeklarować muteks i zmienną warunkową:

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

std::mutex readyMutex;
std::condition_variable readyCondVar;
```

- Wątek (lub jeden z wielu wątków), który sygnalizuje spełnienie warunku, musi wywołać:

  ```
  readyCondVar.notify_one(); // powiadomienie jednego z oczekujących wątków
  ```

  albo:

  ```
  readyCondVar.notify_all(); // powiadomienie wszystkich oczekujących wątków
  ```

- Każdy wątek oczekujący na spełnienie warunku musi wywołać:

  ```
  std::unique_lock<std::mutex> l(readyMutex);
  readyCondVar.wait(l);
  ```

Zatem wątek, który dostarcza danych lub przygotowuje funkcjonalność, wywołuje funkcję `notify_one()` lub `notify_all()` dla zmiennej warunkowej. Jest to moment obudzenia dla jednego bądź wszystkich wątków oczekujących.

Do tej pory wszystko powinno być jasne. Brzmi prosto. Ale jest coś więcej. Po pierwsze, należy zwrócić uwagę, że aby oczekiwać na zmienną warunkową, potrzebny jest muteks oraz obiekt `unique_lock` zaprezentowany po raz pierwszy w punkcie 18.5.1. Obiekt `lock_guard` jest niewystarczający, ponieważ funkcja oczekująca może blokować i odblokowywać muteks.

Ponadto zmienne warunkowe mogą powodować tak zwane *fałszywe wybudzenia*. Oznacza to, że oczekiwanie na zmienną warunkową może zwrócić sterowanie, nawet jeśli powiadomienie o zmiennej warunkowej nie zostało przekazane. Cytując Anthony'ego Williamsa z pozycji [*Williams:CondVar*]: „Fałszywych wybudzeń nie da się przewidzieć; z punktu widzenia użytkownika zdarzają się one losowo. Pomimo to fałszywe wybudzenia powszechnie występują, jeśli biblioteka obsługi wątków nie może niezawodnie zapewnić, że wątek oczekujący nie przegapi powiadomienia. Ponieważ niezauważone powiadomienie spowodowałoby, że zmienna warunkowa stałaby się bezużyteczna, biblioteka obsługi wątków wybudza wątek ze stanu oczekiwania, zamiast podejmować ryzyko".

Zatem wybudzenie niekoniecznie oznacza, że wymagany warunek jest spełniony. Zamiast tego po wybudzeniu należy skorzystać z kodu, który powinien zweryfikować, czy oczekiwany warunek jest rzeczywiście prawdziwy. Dlatego należy na przykład sprawdzić, czy dostarczone dane są rzeczywiście dostępne, albo skorzystać z czegoś w rodzaju flagi gotowości. Do ustawienia i odpytywania o dostarczone dane lub flagę gotowości można wykorzystać ten sam muteks.

18.6.2. Pierwszy kompletny przykład wykorzystania zmiennych warunkowych

Poniższy kod prezentuje kompletny przykład zastosowania zmiennych warunkowych:

```
// concurrency/condvar1.cpp

#include <condition_variable>
#include <mutex>
#include <future>
#include <iostream>
bool readyFlag;
std::mutex readyMutex;
std::condition_variable readyCondVar;

void thread1()
{
    // wykonanie operacji, których wątek thread2 potrzebuje do przygotowania
    std::cout << "<return>" << std::endl;
    std::cin.get();
    // zasygnalizowanie, że wątek thread1 zrealizował działania przygotowawcze
    {
        std::lock_guard<std::mutex> lg(readyMutex);
        readyFlag = true;
    } // zwolnienie blokady
    readyCondVar.notify_one();
}
void thread2()
{
    // oczekiwanie, aż flaga gotowości będzie miała wartość true (readyFlag ma wartość true)
    {
        std::unique_lock<std::mutex> ul(readyMutex);
        readyCondVar.wait(ul, []{ return readyFlag; });
    } // zwolnienie blokady

    // wykonanie operacji, które powinny być wykonane po przygotowaniu działań przez wątek thread1
    std::cout << "zrobione" << std::endl;
}

int main()
{
    auto f1 = std::async(std::launch::async,thread1);
    auto f2 = std::async(std::launch::async,thread2);
}
```

Po włączeniu potrzebnych plików nagłówkowych do zrealizowania komunikacji pomiędzy wątkami potrzebujemy trzech rzeczy:

1. Obiektu na dane dostarczone do przetwarzania lub flagi sygnalizującej, że warunek rzeczywiście jest spełniony (tutaj: `readyFlag`).
2. Muteksu (tutaj: `readyMutex`).
3. Zmiennej warunkowej (tutaj `readyCondVar`).

Wątek dostarczający `thread1()` blokuje muteks `readyMutex`, aktualizuje warunek (obiekt na dane lub na flagę gotowości), odblokowuje muteks i powiadamia zmienną warunkową:

```
{
    std::lock_guard<std::mutex> lg(readyMutex);
    readyFlag = true;
} // zwolnienie blokady
readyCondVar.notify_one();
```

Zwróćmy uwagę, że samo powiadomienie nie musi się znaleźć wewnątrz chronionego obszaru blokady.

Wątek oczekujący (konsumujący lub przetwarzający) blokuje muteks za pomocą blokady `unique_lock` (punkt 18.5.1), oczekuje na powiadomienie, jednocześnie sprawdzając warunek, i zwalnia blokadę:

```
{
    std::unique_lock<std::mutex> ul(readyMutex);
    readyCondVar.wait(ul, []{ return readyFlag; });
} // zwolnienie blokady
```

W tym przypadku funkcja składowa `wait()` w odniesieniu do zmiennych warunkowych jest wykorzystywana w następujący sposób: przekazujemy blokadę `ul` dla muteksu `readyMutex` jako pierwszy argument oraz wyrażenia lambda jako *obiekt wywoływalny* (patrz podrozdział 4.4), sprawdzając warunek przekazany jako drugi argument. Efekt jest taki, że funkcja `wait()` wewnętrznie wywołuje pętlę do czasu, aż przekazany obiekt wywoływalny zwróci `true`. Zatem kod ma taki sam skutek jak kod zamieszczony poniżej, w którym pętla niezbędna do obsługi fałszywych wybudzeń jest jawnie widoczna:

```
{
    std::unique_lock<std::mutex> ul(readyMutex);
    while (!readyFlag) {
        readyCondVar.wait(ul);
    }
} // zwolnienie blokady
```

Zwróćmy uwagę, że musimy tu użyć blokady `unique_lock` i nie możemy użyć obiektu `lock_guard`, ponieważ wewnętrznie składowa `wait()` jawnie odblokowuje i blokuje muteks.

Można argumentować, że jest to zły przykład użycia zmiennych warunkowych, ponieważ można korzystać z futur w celu zablokowania do momentu, aż dotrą określone dane. W związku z tym zaprezentujemy drugi przykład.

18.6.3. Wykorzystanie zmiennych warunkowych do zaimplementowania kolejki dla wielu wątków

W tym przykładzie trzy wątki przesyłają wartości do kolejki, z której dwa inne wątki odczytują dane i je przetwarzają:

```
// concurrency/condvar2.cpp

#include <condition_variable>
#include <mutex>
#include <future>
#include <thread>
#include <iostream>
#include <queue>
std::queue<int> queue;
std::mutex queueMutex;
```

```
std::condition_variable queueCondVar;

void provider (int val)
{
    // przesłanie do kolejki różnych wartości (od val do val+5 z limitem czasu wynoszącym 5 milisekund)
    for (int i=0; i<6; ++i) {
        {
            std::lock_guard<std::mutex> lg(queueMutex);
            queue.push(val+i);
        } // zwolnienie blokady
        queueCondVar.notify_one();

        std::this_thread::sleep_for(std::chrono::milliseconds(val));
    }
}

void consumer (int num)
{
    // odczytanie wartości, jeśli są dostępne (argument num identyfikuje konsumenta)
    while (true) {
        int val;
        {
            std::unique_lock<std::mutex> ul(queueMutex);
            queueCondVar.wait(ul,[]{ return !queue.empty(); });
            val = queue.front();
            queue.pop();
        } // zwolnienie blokady
        std::cout << "konsument " << num << ": " << val << std::endl;
    }
}

int main()
{
    // uruchomienie trzech dostawców dla wartości 100+, 300+ i 500+
    auto p1 = std::async(std::launch::async,provider,100);
    auto p2 = std::async(std::launch::async,provider,300);
    auto p3 = std::async(std::launch::async,provider,500);

    // uruchomienie dwóch konsumentów wyświetlających wartości
    auto c1 = std::async(std::launch::async,consumer,1);
    auto c2 = std::async(std::launch::async,consumer,2);
}
```

Mamy tu globalną kolejkę (patrz podrozdział 12.2), która jest wykorzystywana współbieżnie i chroniona przez muteks oraz zmienną warunkową:

```
std::queue<int> queue;
std::mutex queueMutex;
std::condition_variable queueCondVar;
```

Muteks zapewnia, że operacje odczytu i zapisu są niepodzielne, a zmienna warunkowa służy do wybudzania wątków przetwarzania oraz do powiadamiania ich o tym, że dostępne są nowe wartości.

Danych dostarczają trzy wątki, które umieszczają wartości w kolejce:

```
{
    std::lock_guard<std::mutex> lg(queueMutex);
```

```
        queue.push(val+i);
    } // zwolnienie blokady
    queueCondVar.notify_one();
```

Za pomocą funkcji `notify_one()` budzą jeden z oczekujących wątków w celu przetwarzania następnej wartości. Zauważmy raz jeszcze, że to wywołanie nie musi być częścią chronionej sekcji, zatem zamykamy blok, w którym wcześniej zadeklarowaliśmy strażnika blokady.

Wątki oczekujące na nowe wartości do przetwarzania działają w następujący sposób:

```
int val;
{
    std::unique_lock<std::mutex> ul(queueMutex);
    queueCondVar.wait(ul,[]{ return !queue.empty(); });
    val = queue.front();
    queue.pop();
} // zwolnienie blokady
...
```

W tym kodzie, zgodnie z interfejsem kolejki (patrz podrozdział 12.2), potrzebujemy trzech wywołań w celu pobrania kolejnej wartości z kolejki: funkcja `empty()` sprawdza, czy wartość jest dostępna. Wywołanie funkcji `empty()` spełnia rolę podwójnego sprawdzenia warunku w celu obsługi fałszywych wybudzeń w funkcji `wait()`. Funkcja `front()` odpytuje o następną wartość, natomiast funkcja `pop()` pobiera tę wartość. Wszystkie trzy znajdują się wewnątrz chronionego obszaru unikatowej blokady `ul`. Jednak przetwarzanie wartości zwróconej przez funkcję `front()` odbywa się później. Ma to na celu zminimalizowanie czasu trwania blokady.

Oto możliwy wynik działania tego programu:

```
konsument 1: 300
konsument 1: 100
konsument 2: 500
konsument 1: 101
konsument 2: 102
konsument 1: 301
konsument 2: 103
konsument 1: 104
konsument konsument 1: 105
2: 501
konsument 1: 302
konsument 2: 303
konsument 1: 502
konsument 2: 304
konsument 1: 503
konsument 2: 305
konsument 1: 504
konsument 2: 505
```

Zwróćmy uwagę, że wyjścia z dwóch konsumentów nie są zsynchronizowane, zatem może się zdarzyć, że niektóre znaki mogą się przeplatać. Zwróćmy także uwagę na to, że kolejność, w jakiej są powiadamiane współbieżne wątki oczekujące, nie została zdefiniowana.

W taki sam sposób możemy wywołać funkcję `notify_all()` w przypadku, gdy wiele konsumentów będzie musiało przetwarzać te same dane. Typowym przykładem może być system sterowany zdarzeniami, gdzie zdarzenie musi być opublikowane do wszystkich zarejestrowanych konsumentów.

Zwróćmy także uwagę, że w przypadku zmiennych warunkowych dysponujemy interfejsem pozwalającym na czekanie maksymalnie przez określony czas: funkcja `wait_for()` czeka przez określony czas, natomiast funkcja `wait_until()` czeka do czasu, aż zostanie osiągnięty określony punkt w czasie.

18.6.4. Zmienne warunkowe w szczegółach

Plik nagłówkowy `<condition_variable>` daje dostęp do dwóch klas obsługi zmiennych warunkowych: `condition_variable` i `condition_variable_any`.

Klasa condition_variable

Zgodnie z tym, co napisano w podrozdziale 18.6, klasa `std::condition_variable` jest dostarczana przez bibliotekę standardową C++ w celu umożliwienia wybudzenia jednego bądź kilku wątków oczekujących na spełnienie określonego warunku (wykonanie niezbędnych operacji przygotowawczych albo dostarczenie potrzebnych danych). Na tę samą zmienną warunkową może czekać wiele wątków. Kiedy warunek zostanie spełniony, wątek może powiadomić o tym jeden bądź wszystkie oczekujące wątki.

Ze względu na *fałszywe wybudzenia* samo powiadomienie wątku po spełnieniu warunku nie jest wystarczające. Wątki oczekujące po wybudzeniu mogą i powinny skorzystać z możliwości podwójnego sprawdzenia, czy warunek jest spełniony.

W tabeli 18.10 szczegółowo opisano interfejs klasy `condition_variable` dostarczony przez standardową bibliotekę C++. Klasa `condition_variable_any` dostarcza tego samego interfejsu z wyjątkiem funkcji `native_handle()` i `notify_all_at_thread_exit()`.

TABELA 18.10. Operacje klasy condition_variable

Operacja	Efekt
condvar *cv*	Domyślny konstruktor. Tworzy zmienną warunkową.
cv.~***condvar***()	Niszczy zmienną warunkową.
cv.notify_one()	Wybudza jeden z oczekujących wątków.
cv.notify_all()	Wybudza wszystkie oczekujące wątki.
cv.wait(*ul*)	Oczekuje na powiadomienie z wykorzystaniem unikatowej blokady *ul*.
cv.wait(*ul*,*pred*)	Po wybudzeniu oczekuje na powiadomienie z wykorzystaniem unikatowej blokady *ul* do czasu, aż predykat *pred* zwróci `true`.

TABELA 18.10. Operacje klasy condition_variable (ciąg dalszy)

Operacja	Efekt
`cv.wait_for(ul,duration)`	Oczekuje na powiadomienie z wykorzystaniem unikatowej blokady `ul` przez czas `duration`.
`cv.wait_for(ul,duration,pred)`	Po wybudzeniu oczekuje na powiadomienie z wykorzystaniem unikatowej blokady `ul` przez czas `duration` albo do czasu, aż predykat `pred` zwróci `true`.
`cv.wait_until(ul,timepoint)`	Oczekuje na powiadomienie z wykorzystaniem unikatowej blokady `ul` do punktu w czasie `timepoint`.
`cv.wait_until(ul,timepoint,pred)`	Po wybudzeniu oczekuje na powiadomienie z wykorzystaniem unikatowej blokady `ul` do punktu w czasie `timepoint` albo do czasu, aż predykat `pred` zwróci `true`.
`cv.native_handle()`	Zwraca specyficzny dla platformy typ `native_handle_type` w przypadku nieprzenośnych rozszerzeń.
`notify_all_at_thread_exit(cv,ul)`	Na końcu wątku wywołującego wybudza wszystkie wątki oczekujące na zmienną warunkową `cv`, z wykorzystaniem unikatowej blokady `ul`.

Jeśli stworzenie zmiennej warunkowej nie jest możliwe, konstruktor może zgłosić wyjątek `std::system_error` (patrz punkt 4.3.1) z kodem błędu `resource_unavailable_try_again`, który jest odpowiednikiem kodu POSIX `errno EAGAIN` (patrz punkt 4.3.2). Kopie i przypisania nie są dozwolone.

Powiadomienia są automatycznie synchronizowane, dzięki czemu wywołania `notify_one()` i `notify_all()` nie powodują problemów.

Wszystkie wątki oczekujące na zmienną warunkową muszą korzystać z tego samego muteksu, który powinien być zablokowany przez blokadę `unique_lock` w chwili, gdy zostanie wywołana jedna ze składowych `wait()`. W przeciwnym razie program może działać w nieoczekiwany sposób.

Zwróćmy uwagę, że konsumenci zmiennej warunkowej zawsze działają na muteksach, które są zazwyczaj zablokowane. Jedynie funkcje oczekujące czasowo odblokowują muteks, wykonując jedną z poniższych niepodzielnych operacji[23]:

1. Odblokowanie muteksu i wejście w stan oczekiwania.
2. Odblokowanie oczekiwania.
3. Ponowne zablokowanie muteksu.

Oznacza to, że predykaty przekazywane do funkcji oczekujących zawsze są wywoływane w warunkach blokady, więc mogą bezpiecznie korzystać z obiektu chronionego przez muteks[24]. Wywołania do blokowania i odblokowywania muteksu mogą powodować zgłaszanie odpowiednich wyjątków (patrz punkt 18.5).

Wywołana bez predykatu zarówno funkcja `wait_for()`, jak i `wait_until()` zwraca poniższe wartości *klasy wyliczeniowej* (patrz punkt 3.1.13):

[23] Problem z zastosowaniem naiwnego podejścia w rodzaju *„zablokuj, sprawdź stan, odblokuj, czekaj"* polega na tym, że powiadomienia, które nastąpiły pomiędzy *odblokuj* a *czekaj*, mogą być utracone.

[24] Dziękujemy Bartoszowi Milewskiemu za zwrócenie uwagi na tę kwestię.

- `std::cv_status::timeout`, jeśli upłynął limit czasu;
- `std::cv_status::no_timeout`, jeśli nastąpiło powiadomienie.

Jeśli funkcje `wait_for()` i `wait_until()` zostaną wywołane z predykatem, zwracają wynik predykatu (odpowiedź na pytanie, czy jest spełniony warunek).

Globalna funkcja `notify_all_at_thread_exit(`*cv*`,`*l*`)` służy do wywołania funkcji `notify_all()` w momencie, kiedy wątek wywołujący zakończy działanie. W tym celu wątek ten czasowo blokuje odpowiednią blokadę *l*, która musi wykorzystać ten sam muteks, który wykorzystują wszystkie wątki oczekujące. W celu uniknięcia zakleszczeń wątek powinien zakończyć działanie bezpośrednio po wywołaniu funkcji `notify_all_at_thread_exit()`. Zatem to wywołanie służy tylko do posprzątania przed powiadomieniem oczekujących wątków. Nigdy nie powinno ono zablokować działania programu[25].

Klasa condition_variable_any

Oprócz klasy `std::condition_variable` standardowa biblioteka C++ dostarcza klasy `std::condition_variable_any`, która nie wymaga używania obiektu klasy `std::unique_lock` jako blokady. Zgodnie z tym, co napisano w dokumentacji standardowej biblioteki C++: „Jeśli typ blokady inny niż jeden ze standardowych typów muteksów lub wrapper `unique_lock` dla standardowego typu muteksu zostaną użyte z klasą `condition_variable_any`, użytkownik musi zadbać o wykonanie niezbędnej synchronizacji dotyczącej predykatu powiązanego z egzemplarzem klasy `condition_variable_any`". W rzeczywistości obiekt musi spełnić tzw. wymagania *BasicLockable*, które wymagają dostarczenia synchronizowanych funkcji składowych `lock()` i `unlock()`.

18.7. Atomowe typy danych

W pierwszym przykładzie użycia zmiennych warunkowych (patrz punkt 18.6.1) użyliśmy zmiennej `readyFlag` typu `Boolean`, aby umożliwić wątkowi zasygnalizowanie, że dane lub operacje zostały przygotowane dla innego wątku. Można by się teraz zastanawiać, po co w dalszym ciągu potrzebujemy tu muteksu? Jeśli mamy wartość typu `Boolean`, to dlaczego nie możemy zezwolić na to, aby jeden wątek modyfikował wartość, a drugi współbieżnie ją sprawdzał? W chwili, kiedy wątek dostarczający ustawi wartość `Boolean` na `true`, wątek obserwujący powinien móc to zobaczyć i wykonać odpowiednie działania.

[25] Typowy przykład to sygnalizacja końca działania odłączonego wątku (patrz punkt 18.2.1). Dzięki użyciu funkcji `notify_all_at_thread_exit()` możemy zapewnić, aby lokalne obiekty wątku zostały zniszczone, zanim program główny (główny wątek) przetworzy fakt zakończenia działania odłączonego wątku.

Jak napisano w podrozdziale 18.4, mamy w tym przypadku dwa problemy:

1. Ogólnie rzecz biorąc, odczyt i zapis nawet dla podstawowych typów danych nie jest niepodzielny. W związku z tym możemy odczytać zmienną `Boolean`, która nie jest do końca zapisana, a to zgodnie ze standardem skutkuje niezdefiniowanym zachowaniem.
2. Wygenerowany kod może zmodyfikować kolejność operacji. W związku z tym kod dostarczający może ustawić flagę gotowości, zanim zostaną dostarczone dane, a wątek konsumujący może przetworzyć dane przed sprawdzeniem wartości flagi gotowości.

W przypadku użycia muteksu oba problemy są rozwiązane, ale użycie muteksu może być stosunkowo kosztowną operacją, zarówno jeśli chodzi o niezbędne zasoby, jak i o czas trwania wyłącznego dostępu do danych. Zatem zamiast muteksów i blokad być może warto użyć atomowych typów danych.

W tym podrozdziale najpierw zaprezentuję *wysokopoziomowy interfejs* atomowych typów danych, który dostarcza atomowych operacji wykorzystujących domyślne gwarancje związane z kolejnością dostępu do pamięci. Te domyślne gwarancje zapewniają *sekwencyjną spójność*, co oznacza, że wewnątrz wątku operacje atomowe na pewno zostaną wykonane w takiej kolejności, w jakiej zostały zaprogramowane. W związku z tym problemy zmienionej kolejności instrukcji zaprezentowane w punkcie 18.4.3 nie mają zastosowania. Na końcu tego podrozdziału zaprezentuję *niskopoziomowy interfejs* atomowych typów danych: operacje, w których nie obowiązują ścisłe gwarancje kolejności.

Należy pamiętać, że biblioteka standardowa C++ nie rozróżnia wysokopoziomowego i niskopoziomowego interfejsu atomowych typów danych. Termin *niskopoziomowy* został wprowadzony przez Hansa Boehma, jednego z autorów biblioteki. Czasami jest on również nazywany *słabym* (ang. *weak* lub *relaxed*) interfejsem typów atomowych, natomiast wysokopoziomowy interfejs jest czasem nazywany *normalnym* bądź *silnym*.

18.7.1. Przykład użycia atomowych typów danych

Spróbujmy przekształcić przykład z punktu 18.6.1 na program wykorzystujący atomowe typy danych:

```
#include <atomic>      // do obsługi typów atomowych
...
std::atomic<bool> readyFlag(false);

void thread1()
{
    // wykonanie operacji, których wątek thread2 potrzebuje do przygotowania
    ...
    readyFlag.store(true);
}
void thread2()
{
    // oczekiwanie, aż flaga readyFlag będzie miała wartość true (wątek thread1 jest zakończony)
```

```
    while (!readyFlag.load()) {
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
}

    // wykonanie operacji, które powinny być wykonane po przygotowaniu działań przez wątek thread1
    ...
}
```

Najpierw włączamy plik nagłówkowy `<atomic>`, w którym są zadeklarowane atomowe typy danych:

```
#include <atomic>
```

Następnie deklarujemy atomowy obiekt, korzystając z szablonu klasy `std::atomic<>`:

```
std::atomic<bool> readyFlag(false);
```

Ogólnie rzecz biorąc, w roli parametru szablonu można użyć dowolnego typu wskaźnikowego lub całkowitego.

Zwróćmy uwagę, że *zawsze* powinniśmy zainicjować obiekty atomowe, ponieważ domyślny konstruktor nie w pełni je inicjuje (nie oznacza to, że początkowa wartość jest niezdefiniowana, ale że blokada jest niezainicjowana)[26]. Dla obiektów atomowych o statycznym czasie trwania należy użyć stałej do inicjalizacji. O ile jest wykorzystywany domyślny konstruktor, wtedy jedyną operacją, jaka jest następnie dozwolona, jest wywołanie globalnej funkcji `atomic_init()` w następujący sposób:

```
std::atomic<bool> readyFlag;
...
std::atomic_init(&readyFlag,false);
```

Ten sposób inicjalizacji służy do tego, aby można było pisać kod, który kompiluje się także w języku C (patrz punkt 18.7.3).

Dwie najważniejsze operacje wykonywane z atomowymi typami danych to `store()` i `load()`:

- operacja `store()` przypisuje nową wartość;
- operacja `load()` zwraca bieżącą wartość.

Ważne jest to, że dla tych operacji mamy gwarancję niepodzielności, zatem nie potrzebujemy muteksu do ustawienia flagi gotowości tak jak w przypadku, gdy nie używaliśmy typów atomowych. Zatem w pierwszym wątku zamiast wywołania:

```
{
    std::lock_guard<std::mutex> lg(readyMutex);
    readyFlag = true;
} // zwolnienie blokady
```

możemy po prostu napisać:

```
readyFlag.store(true);
```

[26] Dziękujemy Lawrence'owi Crowlowi za zwrócenie uwagi na tę kwestię.

Natomiast w drugim wątku zamiast instrukcji:

```
{
    std::unique_lock<std::mutex> l(readyFlagMutex);
    while (!readyFlag) {
        l.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        l.lock();
    }
} // zwolnienie blokady
```

musimy zaimplementować tylko następujący fragment:

```
while (!readyFlag.load()) {
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
}
```

Jednak jeśli korzystamy ze zmiennych warunkowych, to w dalszym ciągu potrzebujemy muteksu do skonsumowania zmiennej warunkowej:

```
// oczekiwanie, aż wątek thread1 będzie gotowy (readyFlag ma wartość true)
{
    std::unique_lock<std::mutex> l(readyMutex);
    readyCondVar.wait(l, []{ return readyFlag.load(); });
} // zwolnienie blokady
```

W przypadku atomowych typów danych w dalszym ciągu możemy używać przydatnych „zwyczajnych" operacji, takich jak przypisanie, automatyczna konwersja do typów całkowitych, inkrementacja, dekrementacja itd.:

```
std::atomic<bool> ab(false);
ab = true;
if (ab) {
    ...
}

std::atomic<int> ai(0);
int x = ai;
ai = 10;
ai++;
ai-=17;
```

Zwróćmy jednak uwagę na to, że zapewnienie niepodzielności wiąże się z tym, że standardowe zachowania mogą być nieco różne. Na przykład operator przypisania zwraca przypisaną wartość zamiast referencji do zmiennej atomowego typu danych, do której przypisano tę wartość. Szczegółowe informacje na ten temat można znaleźć w punkcie 18.7.2.

Przyjrzyjmy się kompletnemu przykładowi użycia zmiennych atomowych:

```
// concurrency/atomics1.cpp

#include <atomic> // do obsługi typów atomowych
#include <future> // do wykorzystania funkcji async() i futur
#include <thread> // do wykorzystania klasy this_thread
#include <chrono> // do zmiennych opisujących czas trwania
#include <iostream>
```

```
long data;
std::atomic<bool> readyFlag(false);

void provider ()
{
    // po odczytaniu znaku
    std::cout << "<return>" << std::endl;
    std::cin.get();

    // dostarczenie pewnych danych
data = 42;

    // zasygnalizowanie gotowości
    readyFlag.store(true);
}

void consumer ()
{
    // oczekiwanie na gotowość i wykonanie innej operacji
    while (!readyFlag.load()) {
        std::cout.put('.').flush();
        std::this_thread::sleep_for(std::chrono::milliseconds(500));
    }

    // przetwarzanie dostarczonych danych
    std::cout << "\nwartość : " << data << std::endl;
}

int main()
{
    // uruchomienie dostawcy i konsumenta
    auto p = std::async(std::launch::async,provider);
    auto c = std::async(std::launch::async,consumer);
}
```

W tym przykładzie wątek `provider()` najpierw dostarcza pewnych danych, a następnie wykorzystuje funkcję `store()`, aby zasygnalizować, że dane zostały dostarczone:

```
data = 42;              // dostarczenie danych
readyFlag.store(true); //zasygnalizowanie gotowości
```

Operacja `store()` realizuje tak zwaną operację *zwolnienia* odpowiedniej lokalizacji w pamięci. Domyślnie operacja ta daje pewność, że wszystkie wcześniejsze operacje na pamięci — atomowe lub nie — staną się widoczne dla innych wątków, zanim stanie się widoczny efekt operacji `store()`.

Z kolei wątek `consumer()` wykonuje pętlę złożoną z operacji `load()` i przetwarza w niej dane:

```
while (!readyFlag.load()) { // pętla do czasu gotowości
...
}
std::cout << data << std::endl; // i przetwarzanie dostarczonych danych
```

Operacja `load()` realizuje tak zwaną operację *uzyskania* odpowiedniej lokalizacji w pamięci. Domyślnie operacja ta daje pewność, że wszystkie kolejne operacje na

pamięci — atomowe lub nie — staną się widoczne dla innych wątków po wykonaniu operacji `load()`.

W konsekwencji ze względu na to, że ustawienie danych *odbywa się, zanim* wątek `provider()` zapisze wartość `true` w zmiennej `readyFlag`, natomiast przetwarzanie danych `data` odbywa się po załadowaniu przez wątek `consumer()` wartości `true` jako wartości zmiennej `readyFlag`, to przetwarzanie zmiennej `data` na pewno będzie wykonywane po dostarczeniu danych.

Tę gwarancję uzyskujemy dlatego, ponieważ we wszystkich operacjach na atomowych typach danych wykorzystujemy domyślną *kolejność dostępu do pamięci* zwaną `memory_order_seq_cst` (nazwa pochodzi od *sequential consistent memory order* — dosł. *sekwencyjna spójna kolejność pamięci*). W przypadku korzystania z niskopoziomowych operacji na atomowych typach danych możemy „osłabić" tę gwarancję kolejności (szczegółowe informacje można znaleźć w punkcie 18.7.4).

18.7.2. Atomowe typy danych i ich interfejs wysokiego poziomu w szczegółach

W pliku nagłówkowym `<atomic>` szablon klasy `std::atomic<>` dostarcza ogólnych operacji na atomowych typach danych. Szablon ten można zastosować w odniesieniu do dowolnego prostego typu danych. Wersje specjalistyczne są dostępne dla typu `bool`, wszystkich typów całkowitoliczbowych oraz wskaźników:

```
template<typename T> struct atomic; // podstawowy szablon klasy
template<> struct atomic<bool>;     // jawne specjalizacje
template<> struct atomic<int>;
...
template<typename T> struct atomic<T*>; // częściowa specjalizacja dla wskaźników
```

Wysokopoziomowe operacje dostępne dla atomowych typów danych zestawiono w tabeli 18.11. Jeśli to możliwe, mają one bezpośrednie odwzorowania na odpowiadające im instrukcje CPU. W kolumnie **proste** oznaczono operacje dostępne dla obiektów klasy `std::atomic<bool>` oraz atomowych typów odpowiadających innym typom prostym; w kolumnie **typy *int*** oznaczono operacje dostępne dla klasy `std::atomic<>` typów całkowitoliczbowych; w kolumnie **typy *wsk*** oznaczono operacje dostarczone dla klasy `std::atomic<>` użytej w odniesieniu do typów wskaźnikowych.

Zwróćmy uwagę na kilka spraw związanych z tą tabelą:

- Ogólnie rzecz biorąc, operacje zwracają kopie, a nie elementy.
- Domyślny konstruktor nie inicjuje całkowicie zmiennej (obiektu). Jedyną dozwoloną operacją po stworzeniu obiektu jest wywołanie funkcji `atomic_init()` w celu zainicjowania obiektu (patrz punkt 18.7.1).
- Konstruktor dla wartości odpowiedniego typu nie jest atomowym typem danych.
- Wszystkie funkcje poza konstruktorami są przeciążone dla wartości `volatile` i `non-volatile`.

TABELA 18.11. Wysokopoziomowe operacje na wartościach atomowych typów danych

Operacja	proste	typy int	typy wsk	Efekt
`atomic a=val`	Tak	Tak	Tak	Inicjuje obiekt wartością *val* (to nie jest operacja atomowa)
`atomic a; atomic_init(&a,val)`	Tak	Tak	Tak	jw. (bez `atomic_init()`, argument *a* nie jest inicjowany)
`a.is_lock_free()`	Tak	Tak	Tak	`true`, jeśli wewnętrznie typ nie korzysta z blokad
`a.store(val)`	Tak	Tak	Tak	Przypisuje *val* (zwraca `void`)
`a.load()`	Tak	Tak	Tak	Zwraca kopię wartości `a`
`a.exchange(val)`	Tak	Tak	Tak	Przypisuje wartość *val* i zwraca kopię starej wartości `a`
`a.compare_exchange_strong(exp, des)`	Tak	Tak	Tak	Operacja CAS (patrz poniżej)
`a.compare_exchange_strong(exp, des)`	Tak	Tak	Tak	Słaba operacja CAS
`a = val`	Tak	Tak	Tak	Przypisuje wartość *val* i zwraca kopię `val`
`a.operator atomic()`	Tak	Tak	Tak	Zwraca kopię wartości `a`
`a.fetch_add(val)`		Tak	Tak	Atomowa operacja `t+=val` (zwraca kopię nowej wartości)
`a.fetch_sub(val)`		Tak	Tak	Atomowa operacja `t-=val` (zwraca kopię nowej wartości)
`a += val`		Tak	Tak	To samo co `t.fetch_add(val)`
`a -= val`		Tak	Tak	To samo co `t.fetch_sub(val)`
`++a, a++`		Tak	Tak	Wywołuje instrukcję `t.fetch_add(1)` i zwraca kopię argumentu `a` lub `a+1`
`--a, a--`		Tak	Tak	Wywołuje instrukcję `t.fetch_sub(1)` i zwraca kopię argumentu `a` lub `a-1`
`a.fetch_and(val)`		Tak		Atomowa operacja `a&=val` (zwraca kopię nowej wartości)
`a.fetch_or(val)`		Tak		Atomowa operacja `a|=val` (zwraca kopię nowej wartości)
`a.fetch_xor(val)`		Tak		Atomowa operacja `a^=val` (zwraca kopię nowej wartości)
`a &= val`		Tak		To samo co `a.fetch_sub(val)`
`a |= val`		Tak		To samo co `a.fetch_add(val)`
`a ^= val`		Tak		To samo co `a.fetch_xor(val)`

Na przykład dla typu `atomic<int>` zadeklarowane są następujące operacje przypisania:

```
namespace std {
    // specjalizacja klasy std::atomic<> dla int:
    template<> struct atomic<int> {
        public:
            // zwykłe operatory przypisania nie są dostępne:
            atomic& operator=(const atomic&) = delete;
            atomic& operator=(const atomic&) volatile = delete;
            // ale przypisanie dla danych int jest dostępne; zwraca przekazany argument:
            int operator= (int) volatile noexcept;
            int operator= (int) noexcept;
            ...
};
}
```

Za pomocą funkcji `is_lock_free()` można sprawdzić, czy atomowy typ danych wewnętrznie korzysta z blokad po to, by był atomowy. Jeśli nie, to mamy natywne, sprzętowe wsparcie dla atomowych typów danych (jest to wymaganie wstępne dla używania atomowych typów danych w procedurach obsługi sygnałów).

Operacje `compare_exchange_strong()` oraz `compare_exchange_weak()` są określane jako operacje CAS (od ang. *compare-and-swap* — dosł. *porównaj i przestaw*). Procesory często udostępniają te atomowe operacje do porównywania zawartości lokalizacji pamięci z określoną wartością. Tylko wtedy, gdy te wartości są takie same, następuje zmodyfikowanie tej lokalizacji pamięci na podaną nową wartość. W ten sposób uzyskujemy gwarancję, że nowa wartość będzie obliczona na podstawie aktualnych informacji. Efekt przypomina działanie poniższego pseudokodu:

```
bool compare_exchange_strong (T& expected, T desired)
{
    if (this->load() == expected) {
        this->store(desired);
        return true;
    }
    else {
        expected = this->load();
        return false;
    }
}
```

Zatem jeśli wartość została w międzyczasie zaktualizowana przez inny wątek, to funkcja zwraca `false`, a nowa wartość jest zapisana w argumencie `expected`.

Słaba forma funkcji może fałszywie nie powieść się, tzn. zwraca `false`, nawet kiedy wartość `expected` jest dostępna. Czasami jednak słaba forma jest bardziej wydajna od silnej wersji.

18.7.3. Interfejs atomowych typów danych w stylu języka C

Wraz z propozycją atomowych typów danych dla języka C++ opracowano analogiczną propozycję dla języka C. Założono, że interfejs w stylu języka C powinien zapewniać tę samą semantykę, ale bez wykorzystywania specyficznych własności języka C++, takich jak szablony, referencje i funkcje składowe. Z tego względu dla całego interfejsu atomowych typów danych języka C++ istnieje odpowiednik w stylu języka C, który również zaproponowano jako rozszerzenie do standardu C.

Na przykład można zadeklarować zmienną typu `atomic<bool>` jako `atomic_bool`, a zamiast z funkcji `store()` i `load()` korzystać z funkcji globalnych, które używają wskaźnika do obiektu:

```
std::atomic_bool ab;          // odpowiednik std::atomic<bool> ab
std::atomic_init(&ab,false); // patrz punkt 18.7.1
...
std::atomic_store(&ab,true); // odpowiednik wywołania: ab.store(true)
...
if (std::atomic_load(&ab)) { // odpowiednik wywołania: if (ab.load())
...
}
```

Jednak do języka C dodano inny interfejs z wykorzystaniem przyrostków `_Atomic` i `_Atomic()`, dlatego interfejs w stylu języka C, ogólnie rzecz biorąc, jest przydatny tylko dla kodu, który ma się kompilować zarówno w środowisku C, jak i C++.

Jednak atomowe typy danych w stylu języka C są dość powszechnie używane w C++. W tabeli 18.12 wyszczególniono najważniejsze nazwy atomowych typów danych. Częściej są one wykorzystywane dla mniej popularnych typów, takich jak `atomic_int_fast32_t` dla typu `atomic<int_fast32_t>`.

TABELA 18.12. Niektóre typy danych w stylu języka C odpowiadające typom std::atomic<>

Typ danych w stylu języka C	Odpowiednik typu w stylu języka C++
`atomic_bool`	`atomic<bool>`
`atomic_char`	`atomic<char>`
`atomic_schar`	`atomic<signed char>`
`atomic_uchar`	`atomic<unsigned char>`
`atomic_short`	`atomic<short>`
`atomic_ushort`	`atomic<unsigned short>`
`atomic_int`	`atomic<int>`
`atomic_uint`	`atomic<unsigned int>`
`atomic_long`	`atomic<long>`
`atomic_ulong`	`atomic<unsigned long>`
`atomic_llong`	`atomic<long long>`
`atomic_ullong`	`atomic<unsigned long long>`

TABELA 18.12. Niektóre typy danych w stylu języka C odpowiadające typom std::atomic<> (ciąg dalszy)

Typ danych w stylu języka C	Odpowiednik typu w stylu języka C++
atomic_char16_t	atomic<char16_t>
atomic_char32_t	atomic<char32_t>
atomic_wchar_t	atomic<wchar_t>
atomic_intptr_t	atomic<intptr_t>
atomic_uintptr_t	atomic<uintptr_t>
atomic_size_t	atomic<size_t>
atomic_ptrdiff_t	atomic<ptrdiff_t>
atomic_intmax_t	atomic<intmax_t>
atomic_uintmax_t	atomic<uintmax_t>

Zwróćmy uwagę, że dla współdzielonych wskaźników (patrz punkt 5.2.1) dostępne są specjalne atomowe operacje. Powodem jest to, że deklaracja postaci `atomic ↪<shared_ptr<T>>` nie jest możliwa, ponieważ współdzielony wskaźnik nie jest trywialnie kopiowalny. Operacje na atomowych typach danych mają nazwy zgodne z konwencją nazewnictwa interfejsu w stylu języka C. Szczegółowe informacje na ten temat można znaleźć w punkcie 5.2.4.

18.7.4. Niskopoziomowy interfejs atomowych typów danych

Niskopoziomowy interfejs atomowych typów danych to stosowanie operacji na atomowych typach danych w sposób, który nie gwarantuje sekwencyjnej spójności. Zatem kompilatory i sprzęt mogą (częściowo) zmieniać kolejność dostępu do atomowych typów danych (patrz punkt 18.4.3).

Ostrzegam jeszcze raz: chociaż zaprezentowałem taki przykład, ten obszar jest istnym „polem minowym". Potrzeba bardzo wiele doświadczenia, by wiedzieć, kiedy warto podjąć ryzyko zmiany kolejności dostępu do pamięci, a nawet eksperci często popełniają błędy w tej dziedzinie[27].

Eksperci korzystający z tej własności powinni zapoznać się z materiałami wymienionymi w pozycjach [*N2480:MemMod*] i [*BoehmAdve:MemMod*] oraz ogólnie rzecz biorąc — ze wszystkimi materiałami wyszczególnionymi w [*Boehm:C++MM*].

[27] Specjalne podziękowania należą się Hansowi Boehmowi i Bartoszowi Milewskiemu za pomoc w zrozumieniu tej kwestii oraz ich wsparcie w doborze właściwego słownictwa. Wszelkie uchybienia są wyłącznie moim błędem.

Przykład użycia niskopoziomowego interfejsu atomowych typów danych

Przeanalizujmy drugi przykład użycia atomowych typów danych zaprezentowany w punkcie 18.7.1, gdzie zadeklarowaliśmy atomową flagę w celu zarządzania dostępem do niektórych danych:

```
long data;
std::atomic<bool> readyFlag(false);
```

oraz wątek dostarczający danych:

```
data = 42;              // dostarczenie danych
readyFlag.store(true); // zasygnalizowanie gotowości
```

a także wątek konsumujący dane:

```
while (!readyFlag.load()) { // pętla do czasu gotowości
...
}
std::cout << data << std::endl; // i przetwarzanie dostarczonych danych
```

Ponieważ skorzystaliśmy z domyślnej kolejności dostępu do pamięci, która zapewnia *sekwencyjną spójność*, powyższy przykład działa tak, jak opisano w punkcie 18.7.1. W rzeczywistości wywołaliśmy następujące instrukcje:

```
data = 42;
readyFlag.store(true,std::memory_order_seq_cst);
```

oraz

```
while (!readyFlag.load(std::memory_order_seq_cst)) {
    ...
}
std::cout << data << std::endl;
```

Zatem każda operacja ma opcjonalny argument pozwalający na przekazanie kolejności dostępu do pamięci. Domyślnie jest to wartość `std::memory_order_seq_cst` (*sekwencyjna spójna kolejność dostępu do pamięci*).

Poprzez przekazanie innych wartości opisujących kolejność dostępu do pamięci możemy osłabić gwarancje dotyczące kolejności dostępu. Na przykład w naszym przypadku wystarczy wymagać, aby dostawca nie opóźniał operacji po zapisaniu zmiennej atomowej oraz aby konsument nie przesuwał w przód operacji następujących po załadowaniu zmiennej atomowej:

```
data = 42;
readyFlag.store(true,std::memory_order_release);
```

oraz

```
while (!readyFlag.load(std::memory_order_acquire)) {
    ...
}
std::cout << data << std::endl;
```

Jednak wyeliminowanie wszystkich ograniczeń dotyczących kolejności atomowych operacji skutkowałoby niezdefiniowanym zachowaniem:

```
// BŁĄD: niezdefiniowane zachowanie:
data = 42;
readyFlag.store(true,std::memory_order_relaxed);
```

Powodem jest to, że kolejność dostępu do pamięci `std::memory_order_relaxed` nie gwarantuje tego, że wszystkie wcześniejsze operacje na pamięci staną się widoczne dla innych wątków, zanim stanie się widoczny efekt operacji zapisywania do pamięci (`store`). Zatem dostawca mógł zapisać dane po ustawieniu flagi gotowości. Tym samym konsument mógł odczytać dane w czasie, gdy były one zapisywane — jest to sytuacja *wyścigu o dane*.

Zwróćmy także uwagę, że zmienną `data` można również zadeklarować jako atomową i wykorzystać wartość `std::memory_order_relaxed` jako kolejność dostępu do pamięci:

```
std::atomic<long> data(0);
std::atomic<bool> readyFlag(false);

// wątek dostawcy:
data.store(42,std::memory_order_relaxed);
readyFlag.store(true,std::memory_order_relaxed);

// wątek konsumenta:
while (!readyFlag.load(std::memory_order_relaxed)) {
    ...
}
std::cout << data.load(std::memory_order_relaxed) << std::endl;
```

Ściśle rzecz biorąc, to nie jest *niezdefiniowane zachowanie*, ponieważ nie mamy tu do czynienia z *wyścigiem o dane*. Jednak ten kod nie będzie działał zgodnie z oczekiwaniami, ponieważ wartość zmiennej `data` może być inna niż `42` (kolejność dostępu do danych nie jest gwarantowana). Jest to zachowanie polegające na tym, że zmienna `data` ma *nieokreśloną* wartość.

Użycie wartości `memory_order_relaxed` byłoby przydatne tylko wtedy, gdybyśmy mieli zmienne atomowe, gdzie odczyt i (lub) zapis zachodziłyby niezależnie od siebie. Przykładem może być globalny licznik, który może być inkrementowany lub dekrementowany przez różne wątki, a potrzebujemy tylko jego ostatecznej wartości po zakończeniu działania wszystkich wątków.

Przegląd operacji niskiego poziomu

W tabeli 18.13 wyszczególniono uzupełniające operacje niskopoziomowe dostępne dla atomowych typów danych. Jak można zauważyć, operacje ładowania, zapisywania, wymiany, CAS i pobierania dają możliwość podania kolejności dostępu do pamięci jako dodatkowego argumentu.

Dostępne są także dodatkowe funkcje pozwalające na ręczne zarządzanie dostępem do pamięci. Na przykład dostępne są funkcje `atomic_thread_fence()` i `atomic_ signal_fence()` pozwalające na ręczne zaprogramowanie płotów — tzn. barier do zmiany kolejności dostępu do pamięci.

TABELA 18.13. Uzupełniające niskopoziomowe operacje na wartościach atomowych typów danych

Operacja	proste	typy int	typy wsk
`a.store(val,mo)`	Tak	Tak	Tak
`a.load(mo)`	Tak	Tak	Tak
`a.exchange(val,mo)`	Tak	Tak	Tak
`a.compare_exchange_strong(exp,des,mo)`	Tak	Tak	Tak
`a.compare_exchange_strong(exp,des,mo1,mo2)`	Tak	Tak	Tak
`a.compare_exchange_weak(exp,des,mo)`	Tak	Tak	Tak
`a.compare_exchange_weak(exp,des,mo1,mo2)`	Tak	Tak	Tak
`a.fetch_add(val,mo)`		Tak	Tak
`a.fetch_sub(val,mo)`		Tak	Tak
`a.fetch_and(val,mo)`		Tak	
`a.fetch_or(val,mo)`		Tak	
`a.fetch_xor(val,mo)`		Tak	

Bez dodatkowych szczegółów

Nie objaśniam tego niskopoziomowego interfejsu w szczegółach, ponieważ ta własność jest przeznaczona dla prawdziwych ekspertów w dziedzinie współbieżności lub osób, które chcą zostać takimi ekspertami. W związku z tym z całą pewnością należy skorzystać z literatury poświęconej specjalnie tej tematyce.

Dobrym punktem startowym jest książka Anthony'ego Williamsa *C++ Concurrency in Action* (patrz [*Williams:C++Conc*]), zwłaszcza rozdziały 5. i 7. Inną pozycją godną polecenia jest skompilowana przez Hansa Boehma lista adresów URL do materiałów dotyczących modeli pamięci (patrz [*Boehm:C++MM*]).

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

Z

Ź

Ż

ODKRYJ POTĘGĘ C++!

Pomimo olbrzymiej konkurencji ze strony platformy .NET oraz języka Java język C++ wciąż jest niezastąpiony w wielu dziedzinach. Wszędzie tam, gdzie wymagane są najwyższa wydajność, pełna kontrola nad sprzętem oraz przewidywalność, C++ jest bezkonkurencyjny. Biblioteka standardowa C++ to zestaw klas i interfejsów, które w znaczny sposób zwiększają możliwości tego języka. Warto wykorzystać jej potencjał!

Biblioteka standardowa C++ poza wieloma niewątpliwymi zaletami ma jedną poważną wadę — jest trudna do opanowania. Właśnie dlatego potrzebny Ci jest ten podręcznik! W trakcie lektury poznasz nowe elementy języka C++ w wersji 11. Następnie dowiesz się, czym jest standardowa biblioteka szablonów (STL), oraz zobaczysz, jak wykorzystać w codziennej pracy: mapy, multimapy, iteratory, listy oraz wiele innych elementów. Na sam koniec nauczysz się poprawnie korzystać ze współbieżności oraz tworzyć aplikacje obsługujące różne wersje językowe. Każdy z komponentów biblioteki został dokładnie przedstawiony: z opisem przeznaczenia, przykładami oraz problemami, których może przysporzyć. Książka ta jest obowiązkową lekturą każdego programisty C++!

Dzięki tej książce:

- poznasz nowości języka C++ w wersji 11
- wykorzystasz możliwości kontenerów STL
- zrozumiesz zastosowanie iteratorów
- zobaczysz na praktycznych przykładach, jak działają komponenty
- błyskawicznie opanujesz możliwości biblioteki standardowej C++

helion.pl
księgarnia internetowa

Nr katalogowy: 18835

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Addison-Wesley
Pearson Education

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-246-5576-2
cena: 149,00 zł
9 788324 655762