

Jacek Galowicz

C++17 STL

Receptury

Helion 

Packt 

Tytuł oryginału: C++17 STL Cookbook

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-4501-0

Copyright © Packt Publishing 2017. First published in the English language under the title 'C++17 STL Cookbook – (9781787120495)'

Polish edition copyright © 2018 by Helion SA
All rights reserved.

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

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

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

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

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/cpp17r>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

Spis treści

O autorze	13
O redaktorze merytorycznym	14
Wprowadzenie	15
Rozdział 1. Nowe funkcje w C++17	21
Wprowadzenie	21
Użycie strukturalnych wiązań do rozpakowania wartości zwrótej	22
Jak to zrobić?	22
Jak to działa?	24
Co dalej?	24
Ograniczanie zasięgu zmiennej do konstrukcji if i switch	26
Jak to zrobić?	26
Jak to działa?	27
Co dalej?	28
Zalety stosowania nowych reguł inicjalizacji z użyciem składni opartej na nawiasach	29
Jak to zrobić?	29
Jak to działa?	30
Umożliwienie konstruktorowi automatycznego określenia typu klasy szablonu	31
Jak to zrobić?	31
Jak to działa?	31
Co dalej?	32
Użycie wyrażenia constexpr-if do uproszczenia decyzji podejmowanych podczas kompilacji	33
Jak to zrobić?	34
Jak to działa?	35
Co dalej?	36
Włączenie bibliotek w postaci samych nagłówków z użyciem osadzonych zmiennych	37
Jak to zrobić?	37
Jak to działa?	38
Co dalej?	39

Implementowanie za pomocą wyrażeń fold przydatnych funkcji pomocniczych	40
Jak to zrobić?	40
Jak to działa?	41
Co dalej?	41
Rozdział 2. Kontenery STL	47
Wprowadzenie	48
Magazyn danych znajdujących się obok siebie	48
Magazyn danych w postaci listy	49
Drzewo wyszukiwania	49
Tabela wartości hash	50
Adapter kontenera	50
Użycie stylu usuń – wyciąż w kontenerze std::vector	50
Jak to zrobić?	51
Jak to działa?	52
Co dalej?	54
Usuwanie w czasie O(1) elementów z nieposortowanego kontenera std::vector	54
Jak to zrobić?	55
Jak to działa?	57
Uzyskanie bezpiecznego dostępu do egzemplarzy std::vector	58
Jak to zrobić?	58
Jak to działa?	59
Co dalej?	60
Sortowanie egzemplarzy std::vector	60
Jak to zrobić?	60
Jak to działa?	62
Co dalej?	62
Efektywne i warunkowe wstawianie elementów do kontenera std::map	63
Jak to zrobić?	63
Jak to działa?	65
Co dalej?	66
Stosowanie nowej semantyki podpowiedzi podczas wstawiania elementów za pomocą std::map::insert	66
Jak to zrobić?	66
Jak to działa?	68
Co dalej?	68
Efektywne modyfikowanie kluczy elementów std::map	69
Jak to zrobić?	70
Jak to działa?	72
Co dalej?	72
Użycie kontenera std::unordered_map z niestandardowymi typami danych	73
Jak to zrobić?	73
Jak to działa?	75
Filtrowanie duplikatów w danych wejściowych użytkownika i wyświetlanie ich w kolejności alfabetycznej za pomocą kontenera std::set	76
Jak to zrobić?	76
Jak to działa?	77

Implementowanie za pomocą kontenera <code>std::stack</code> prostego kalkulatora RPN	79
Jak to zrobić?	80
Jak to działa?	82
Co dalej?	84
Implementowanie za pomocą kontenera <code>std::map</code> licznika częstotliwości występowania słów	85
Jak to zrobić?	85
Jak to działa?	87
Implementowanie za pomocą kontenera <code>std::set</code> narzędzia pomocniczego przeznaczonego do wyszukiwania bardzo długich zdań w tekście	88
Jak to zrobić?	89
Jak to działa?	91
Co dalej?	92
Implementowanie za pomocą kontenera <code>std::priority_queue</code> listy rzeczy do zrobienia	93
Jak to zrobić?	93
Jak to działa?	95
Rozdział 3. Iteratory	97
Wprowadzenie	97
Kategorie iteratorów	99
Tworzenie własnego zakresu, który można iterować	101
Jak to zrobić?	101
Jak to działa?	103
Tworzenie własnych iteratorów zgodnych z kategoriami iteratora STL	104
Jak to zrobić?	104
Jak to działa?	106
Co dalej?	107
Użycie adapterów iteratora do wypełniania ogólnych struktur danych	107
Jak to zrobić?	107
Jak to działa?	109
Implementowanie algorytmów w kategoriach iteratorów	110
Jak to zrobić?	111
Co dalej?	113
Iteracja w drugą stronę za pomocą adaptera iteratora odwrotnego	114
Jak to zrobić?	114
Jak to działa?	115
Zakończenie działania iteratora w zakresie za pomocą wartownika iteratora	116
Jak to zrobić?	117
Automatyczne sprawdzanie kodu iteratora	119
Jak to zrobić?	119
Jak to działa?	122
Co dalej?	123
Tworzenie własnego adaptera iteratora łączenia na zakładkę	123
Jak to zrobić?	125
Co dalej?	128

Rozdział 4. Wyrażenia lambda	131
Wprowadzenie	131
Definiowanie funkcji opartej na wyrażeniu lambda	133
Jak to zrobić?	133
Jak to działa?	136
Dodawanie polimorfizmu poprzez opakowanie wyrażenia lambda egzemplarzem <code>std::function</code>	138
Jak to zrobić?	138
Jak to działa?	140
Łączenie funkcji za pomocą konkatencji	141
Jak to zrobić?	142
Jak to działa?	143
Tworzenie skomplikowanych predykatów z logiczną koniunkcją	144
Jak to zrobić?	145
Co dalej?	146
Wywoływanie wielu funkcji dla tych samych danych wejściowych	146
Jak to zrobić?	147
Jak to działa?	148
Implementowanie funkcji <code>transform_if()</code> za pomocą algorytmu <code>std::accumulate</code> i wyrażeń lambda	150
Jak to zrobić?	150
Jak to działa?	152
Generowanie w trakcie kompilacji iloczynu kartezjańskiego par dla dowolnych danych wejściowych	155
Jak to zrobić?	156
Jak to działa?	158
Rozdział 5. Podstawy algorytmów biblioteki STL	161
Wprowadzenie	161
Kopiowanie elementów między kontenerami	163
Jak to zrobić?	164
Jak to działa?	166
Sortowanie kontenera	167
Jak to zrobić?	167
Jak to działa?	170
Usuwanie określonych elementów z kontenera	171
Jak to zrobić?	171
Jak to działa?	173
Przekształcanie zawartości kontenera	174
Jak to zrobić?	174
Jak to działa?	176
Wyszukiwanie elementów w uporządkowanych i nieuporządkowanych wektorach	176
Jak to zrobić?	177
Jak to działa?	180
Ograniczanie za pomocą <code>std::clamp</code> wartości wektora do określonego zakresu liczbowego	182
Jak to zrobić?	182
Jak to działa?	185

Wyszukiwanie za pomocą <code>std::search</code> wzorca w ciągu tekstowym i wybór optymalnej implementacji	186
Jak to zrobić?	186
Jak to działa?	188
Próbkowanie ogromnego wektora	189
Jak to zrobić?	190
Jak to działa?	192
Generowanie permutacji sekwencji danych wejściowych	193
Jak to zrobić?	193
Jak to działa?	194
Implementowanie narzędzia łączenia słowników	195
Jak to zrobić?	195
Jak to działa?	197
Rozdział 6. Zaawansowane przykłady użycia algorytmów biblioteki STL	199
Wprowadzenie	200
Implementowanie klasy drzewa trie za pomocą algorytmów STL	201
Jak to zrobić?	201
Jak to działa?	204
Implementowanie za pomocą drzewa trie generatora sugestii danych wejściowych używanych podczas wyszukiwania	206
Jak to zrobić?	206
Jak to działa?	210
Co dalej?	210
Implementowanie wzoru przekształcenia Fouriera za pomocą algorytmów STL	211
Jak to zrobić?	212
Jak to działa?	217
Obliczanie błędu sumy dwóch wektorów	218
Jak to zrobić?	218
Jak to działa?	220
Implementowanie procedury generującej dane ASCII dla zbioru Mandelbrota	221
Jak to zrobić?	223
Jak to działa?	226
Opracowanie własnego algorytmu — podział danych	227
Jak to zrobić?	228
Jak to działa?	229
Co dalej?	230
Połączenie użytecznych algorytmów biblioteki STL — zbieranie danych	231
Jak to zrobić?	231
Jak to działa?	233
Usuwanie nadmiarowych białych znaków znajdujących się między słowami	235
Jak to zrobić?	235
Jak to działa?	237
Kompresja i dekompresja ciągów tekstowych	238
Jak to zrobić?	238
Jak to działa?	240
Co dalej?	241

Rozdział 7. Ciągi tekstowe, klasy strumieni i wyrażenia regularne	243
Wprowadzenie	244
Tworzenie, konkatenacja i przekształcanie ciągów tekstowych	245
Jak to zrobić?	246
Jak to działa?	248
Usuwanie białych znaków z początku i końca ciągu tekstowego	248
Jak to zrobić?	249
Jak to działa?	250
Komfortowe użycie klasy <code>std::string</code> bez kosztów związanych z tworzeniem obiektów <code>std::string</code>	251
Jak to zrobić?	252
Jak to działa?	254
Odczyt wartości z danych wejściowych dostarczonych przez użytkownika	254
Jak to zrobić?	255
Jak to działa?	257
Zliczanie wszystkich słów w pliku	258
Jak to zrobić?	258
Jak to działa?	260
Formatowanie danych wyjściowych za pomocą manipulatorów strumienia wejścia – wyjścia	260
Jak to zrobić?	261
Jak to działa?	264
Inicjalizacja skomplikowanych obiektów na podstawie pliku źródłowego	266
Jak to zrobić?	266
Jak to działa?	268
Wypełnianie kontenera za pomocą iteratorów <code>std::istream</code>	269
Jak to zrobić?	269
Jak to działa?	272
Proste wyświetlanie danych za pomocą iteratorów <code>std::ostream</code>	273
Jak to zrobić?	273
Jak to działa?	276
Przekierowywanie sekcji kodu do pliku danych wyjściowych	277
Jak to zrobić?	278
Jak to działa?	280
Tworzenie własnych klas ciągu tekstowego za pomocą dziedziczenia po klasie <code>std::char_traits</code>	281
Jak to zrobić?	282
Jak to działa?	286
Tokenizowanie danych wejściowych za pomocą biblioteki wyrażeń regularnych	287
Jak to zrobić?	287
Jak to działa?	289
Wygodne formatowanie liczb w locie w zależności od kontekstu	291
Jak to zrobić?	291
Przechwytywanie na podstawie błędów <code>std::istream</code> wyjątków możliwych do odczytania	293
Jak to zrobić?	294
Jak to działa?	296

Rozdział 8. Klasy narzędziowe	297
Wprowadzenie	298
Konwertowanie między różnymi jednostkami czasu za pomocą std::ratio	298
Jak to zrobić?	299
Jak to działa?	301
Co dalej?	303
Konwertowanie między bezwzględnymi i względnymi wartościami czasu za pomocą std::chrono	304
Jak to zrobić?	304
Jak to działa?	306
Bezpieczne sygnalizowanie awarii za pomocą typu std::optional	307
Jak to zrobić?	307
Jak to działa?	310
Użycie funkcji wraz z krotkami	311
Jak to zrobić?	311
Jak to działa?	313
Szybkie opracowywanie struktur danych za pomocą std::tuple	313
Jak to zrobić?	314
Jak to działa?	318
Zastąpienie void* przez std::any dla zwiększenia bezpieczeństwa typu	320
Jak to zrobić?	321
Jak to działa?	323
Przechowywanie różnych typów za pomocą std::variant	323
Jak to zrobić?	324
Jak to działa?	327
Automatyczna obsługa zasobów za pomocą std::unique_ptr	329
Jak to zrobić?	329
Jak to działa?	332
Automatyczna obsługa współdzielonej pamięci na stercie za pomocą std::shared_ptr	333
Jak to zrobić?	333
Jak to działa?	336
Co dalej?	337
Praca ze słabymi wskaźnikami do współdzielonych obiektów	338
Jak to zrobić?	339
Jak to działa?	341
Uproszczenie obsługi zasobów przestarzałych API za pomocą sprytnych wskaźników	342
Jak to zrobić?	343
Jak to działa?	344
Współdzielenie różnych wartości składowych tego samego obiektu	345
Jak to zrobić?	346
Jak to działa?	347
Generowanie liczb losowych i wybór odpowiedniego silnika do generowania tego rodzaju liczb	348
Jak to zrobić?	349
Jak to działa?	353
Generowanie liczb losowych i umożliwienie bibliotece STL określenia szczegółów rozkładu	354
Jak to zrobić?	354
Jak to działa?	359

Rozdział 9. Programowanie równoległe i współbieżność	361
Wprowadzenie	362
Automatyczne stosowanie programowania równoległego w kodzie utworzonego za pomocą standardowych algorytmów	363
Jak to zrobić?	363
Jak to działa?	365
Uśpienie programu na podany okres czasu	369
Jak to zrobić?	369
Jak to działa?	370
Uruchamianie i zatrzymywanie wątków	371
Jak to zrobić?	371
Jak to działa?	373
Przeprowadzanie bezpiecznego pod względem wyjątków nakładania blokady współdzielonej za pomocą <code>std::unique_lock</code> i <code>std::shared_lock</code>	375
Jak to zrobić?	375
Jak to działa?	378
Zapobieganie zakleszczeniom dzięki stosowaniu algorytmu <code>std::scoped_lock</code>	381
Jak to zrobić?	382
Jak to działa?	384
Synchronizacja jednoczesnego użycia algorytmu <code>std::cout</code>	384
Jak to zrobić?	385
Jak to działa?	386
Bezpieczne odkładanie inicjalizacji za pomocą <code>std::call_once</code>	388
Jak to zrobić?	389
Jak to działa?	390
Przesunięcie zadania do wykonywania w tle za pomocą <code>std::async</code>	390
Jak to zrobić?	391
Jak to działa?	393
Co dalej?	395
Implementacja wzorca producent – konsument za pomocą <code>std::condition_variable</code>	395
Jak to zrobić?	396
Jak to działa?	398
Implementacja wzorca producent – konsument za pomocą <code>std::condition_variable</code>	400
Jak to zrobić?	400
Jak to działa?	404
Równoległe generowanie za pomocą <code>std::async</code> danych ASCII dla zbioru Mandelbrota	406
Jak to zrobić?	406
Jak to działa?	409
Implementacja za pomocą <code>std::future</code> niewielkiej biblioteki automatycznej programowania równoległego	410
Jak to zrobić?	411
Jak to działa?	415

Rozdział 10. System plików	419
Wprowadzenie	419
Implementowanie programu przeprowadzającego normalizację ścieżki dostępu	420
Jak to zrobić?	420
Jak to działa?	422
Co dalej?	422
Pobieranie kanonicznej ścieżki dostępu na podstawie względnej ścieżki dostępu	423
Jak to zrobić?	423
Jak to działa?	426
Wyświetlanie wszystkich plików znajdujących się w danym katalogu	426
Jak to zrobić?	427
Jak to zrobić?	430
Implementowanie programu wyszukującego dane i działającego podobnie jak narzędzie grep	431
Jak to zrobić?	431
Jak to działa?	433
Co dalej?	434
Implementowanie programu automatycznie zmieniającego nazwy plików	434
Jak to zrobić?	435
Implementowanie programu obliczającego wielkość katalogu	437
Jak to zrobić?	437
Jak to działa?	439
Obliczanie danych statystycznych dotyczących typów plików	440
Jak to zrobić?	440
Implementowanie narzędzia zmniejszającego wielkość katalogu poprzez zastąpienie powielonych plików dotaczeniami symbolicznymi	442
Jak to zrobić?	443
Jak to działa?	445
Co dalej?	446
Skorowidz	447

Nowe funkcje w C++17

W tym rozdziale omówię następujące receptury:

- Użycie strukturalnych wiązań do rozpakowania wartości zwrótniej
- Ograniczanie zasięgu zmiennej do konstrukcji `if` i `switch`
- Zalety stosowania nowych reguł inicjalizacji z użyciem składni opartej na nawiasach
- Umożliwienie konstruktorowi automatycznego określenia typu klasy szablonu
- Użycie wyrażenia `constexpr-if` do uproszczenia decyzji podejmowanych podczas kompilacji
- Włączenie bibliotek w postaci samych nagłówków z użyciem osadzonych zmiennych
- Implementowanie za pomocą wyrażzeń `fold` przydatnych funkcji pomocniczych

Wprowadzenie

Język C++ zyskał wiele usprawnień wraz z wydaniem standardów C++11, C++14 i C++17. Obecnie jest to zupełnie inny język w porównaniu do tego, którym był jeszcze dekadę temu. Standard C++ nie dotyczy jedynie języka, który musi być obsługiwany przez kompilatory, ale również standardową bibliotekę szablonów (ang. *standard template library* — STL).

Na podstawie wielu przykładów przedstawionych w tej książce dowiesz się, jak najlepiej wykorzystać bibliotekę STL. W tym rozdziale skoncentruję się na najważniejszych nowych funkcjach wprowadzonych w najnowszym wydaniu C++. Ich opanowanie bardzo pomoże w tworzeniu ekspresyjnego, łatwego w odczycie i późniejszej konserwacji kodu.

Zobaczysz, jak za pomocą strukturalnych wiązań komfortowo uzyskać dostęp do poszczególnych elementów składowych par, krotek i struktur, a także jak ograniczyć zasięg zmiennej, wykorzystując do tego nowe możliwości w zakresie inicjalizacji zmiennej w konstrukcjach `if` i `switch`. Syntaktyczna dwuznaczność wprowadzona przez standard C++11 wraz z nową składnią inicjalizacji z użyciem nawiasu wyszukującą ten sam inicjalizator dla list została poprawiona dzięki *nowym regułom inicjalizacji z użyciem składni opartej na nawiasach*. Konkretny *typ* egzemplarza klasy szablonu może być *ustalony* na podstawie rzeczywistych argumentów konstruktora. Jeżeli poszczególne specjalizacje szablonu klasy będą powodowały powstawanie zupełnie odmiennego kodu, teraz można to łatwo wyrazić za pomocą wyrażenia `constexpr-if`. Obsługa parametrów wariadycznych w funkcjach szablonu stała się w wielu przypadkach znacznie łatwiejsza dzięki pojawieniu się nowych *wyrażeń fold*. Wreszcie znacznie bardziej komfortowe jest definiowanie statycznych, dostępnych globalnie obiektów w bibliotekach w postaci jedynie nagłówków z użyciem nowych możliwości deklarowania osadzonych zmiennych, co wcześniej było możliwe jedynie w funkcjach.

Część przykładów w tym rozdziale może być bardziej interesująca dla programistów opracowujących biblioteki niż zajmujących się tworzeniem aplikacji. Część wymienionych funkcji przedstawiłem tutaj dla porządku, nie musisz dokładnie zrozumieć wszystkich przykładów w tym rozdziale, aby móc opanować materiał zaprezentowany w pozostałej części książki.

Użycie strukturalnych wiązań do rozpakowania wartości zwrotnej

Standard C++17 oferuje nową funkcję łączącą w sobie syntaktyczny lukier z automatycznym ustaleniem typu: **strukturalne wiązanie**. Pomaga ono w przypisywaniu poszczególnym zmiennym wartości pochodzących z par, krotek i struktur. W innych językach programowania to zadanie nosi nazwę **rozpakowania**.

Jak to zrobić?

Zastosowanie strukturalnego wiązania w celu przypisania wielu wartości pochodzących z większej struktury zawsze ma postać pojedynczego kroku. Najpierw zobacz, jak taka operacja odbywała się przed wprowadzeniem standardu C++17. Następnie przejdę do kilku przykładów pokazujących, jak można to zrobić w C++17.

- Zaczynam od uzyskania dostępu do poszczególnych wartości w `std::pair`. Przyjmuję założenie o istnieniu funkcji matematycznej, `divide_remainder()`, która akceptuje *dzielną* (parametr `dividend`) i *dzielnik* (parametr `divisor`) oraz zwraca wynik dzielenia liczb wraz z resztą. Wspomniane wartości są zwracane za pomocą algorytmu `std::pair`:

```
std::pair<int, int> divide_remainder(int dividend, int divisor);
```

Spójrz na sposób, w jaki uzyskujemy dostęp do poszczególnych wartości znajdujących się w wyniku działania funkcji:

```
const auto result (divide_remainder(16, 3));
    std::cout << "16 / 3 to "
                << result.first << " plus reszta z dzielenia "
                << result.second << '\n';
```

Zamiast stosować podejście przedstawione powyżej poszczególne wartości można przypisać zmiennym za pomocą ekspresyjnych nazw, które są znacznie łatwiejsze w odczycie. Spójrz na ten fragment kodu:

```
auto [fraction, remainder] = divide_remainder(16, 3);
std::cout << "16 / 3 to "
            << fraction << " plus reszta z dzielenia "
            << remainder << '\n';
```

- Strukturalne wiązanie działa również z algorytmem `std::tuple`. Przyjmujemy założenie o istnieniu przedstawionej poniżej funkcji, która zwraca pobrane z internetu informacje o kursie akcji:

```
std::tuple<std::string,
           std::chrono::system_clock::time_point, unsigned>
stock_info(const std::string &name);
```

Przypisanie wyniku poszczególnym zmiennym odbywa się dokładnie tak samo jak w poprzednim przykładzie:

```
const auto [name, valid_time, price] = stock_info("INTC");
```

- Strukturalne wiązanie sprawdza się również w przypadku pracy z niestandardowymi strukturami. Przyjmuję założenie o istnieniu następującej struktury:

```
struct employee {
    unsigned id;
    std::string name;
    std::string role;
    unsigned salary;
};
```

Dostęp do poszczególnych elementów składowych można uzyskać za pomocą strukturalnego wiązania. Alternatywne podejście polega na użyciu pętli, przy założeniu o istnieniu wektora, jak pokazałem tutaj:

```
int main()
{
    std::vector<employee> employees {
        /* Inicjalizacja odbyła się w innym miejscu */;
    for (const auto &[id, name, role, salary] : employees) {
        std::cout << "Nazwisko: " << name
                  << "Stanowisko: " << role
                  << "Wynagrodzenie: " << salary << '\n';
    }
}
```

Jak to działa?

Strukturalne wiązanie jest zawsze stosowane z użyciem następującego wzorca:

```
auto [zmienna1, zmienna2, ...] = <wyrażenie pary, krotki, struktury lub tablicy>;
```

- Lista zmiennych, `zmienna1`, `zmienna2` itd., musi dokładnie odpowiadać liczbie zmiennych znajdujących się w wyrażeniu, z którego będą pochodziły przypisywane wartości.
- Wyrażenie `<pary, krotki, struktury lub tablicy>` musi mieć jedną z następujących postaci:
 - Para: `std::pair`.
 - Krotka: `std::tuple`.
 - Struktura. Żaden element składowy *nie może* być statyczny, a wszystkie muszą być zdefiniowane w tej *samej klasie bazowej*. Pierwszy zadeklarowany element składowy będzie przypisany pierwszej zmiennej, drugi element składowy — drugiej zmiennej itd.
 - Tablica o stałej wielkości.
- Typem może być `auto`, `const auto`, `const auto&` lub nawet `auto&&`.

Ze względu między innymi na *wydajność* staraj się podczas używania odwołań zminimalizować liczbę niepotrzebnych operacji kopiowania.

Jeżeli podasz zbyt *dużą* lub *małą* liczbę zmiennych w nawiasie kwadratowym, wówczas kompilator wygeneruje komunikat błędu:

```
std::tuple<int, float, long> tup {1, 2.0, 3};
auto [a, b] = tup; // To polecenie nie działa
```

Powyższe polecenie próbuje skopiować dane z trzyelementowej krotki do jedynie dwóch zmiennych. Kompilator natychmiast wychwyci ten błąd i wygeneruje odpowiedni komunikat:

```
error: type 'std::tuple<int, float, long>' decomposes into 3 elements, but
only 2 names were provided
auto [a, b] = tup;
```

Co dalej?

Za pomocą strukturalnych wiązań można natychmiast uzyskać dostęp do wielu podstawowych struktur danych STL bez konieczności zmiany czegokolwiek. Na przykład tutaj przedstawiłem pętlę, która wyświetla wszystkie elementy zdefiniowane w egzemplarzu `std::map`:

```
std::map<std::string, size_t> animal_population {
    {"ludzi", 7000000000},
    {"kurczaków", 17863376000},
};
```



```

    {"wielbłądów", 24246291},
    {"owiec", 1086881528},
    /* ... */
};

for (const auto &[species, count] : animal_population) {
    std::cout << "Mamy " << count << " " << żyjących
                << " na naszej planecie.\n";
}

```

Ten konkretny przykład działa, ponieważ w trakcie iteracji kontenera `std::map` powstaje węzeł `std::pair<const typ_klucza, typ_wartości>` dla każdego kroku iteracji. Dokładniej mówiąc, te węzły są rozpakowywane za pomocą funkcji wiązania — `typ_klucza` to ciąg tekstowy będący nazwą populacji (`string`), natomiast `typ_wartości` to wielkość populacji podana jako `size_t` — w celu uzyskania do nich dostępu w treści pętli.

Przed wprowadzeniem standardu C++17 podobny efekt można było otrzymać po zastosowaniu wywołania `std::tie()`:

```

int remainder;
std::tie(std::ignore, remainder) = divide_remainder(16, 5);
std::cout << "16 % 5 to " << remainder << '\n';

```

Przykład ten pokazuje, jak parę wynikową rozpakować na postać dwóch zmiennych. Wywołanie `std::tie()` oferuje znacznie mniejsze możliwości niż strukturalne wiązanie w tym sensie, że wszystkie zmienne, do których będą dołączane wartości, trzeba *wcześniej* zdefiniować. Jednak z drugiej strony przykład pokazuje zaletę wywołania `std::tie()` *nieoferowaną* przez strukturalne wiązanie: wartość `std::ignore` działa w charakterze imitacji wartości. To będzie wartość reszty z dzielenia usuwana w tym przykładzie, ponieważ jest w tym miejscu niepotrzebna.

Podczas używania strukturalnego wiązania nie ma imitacji wartości, więc wszystkie wartości są przypisywane nazwanym zmiennym. Tego rodzaju operacja, a następnie zignorowanie niepotrzebnej wartości, jest mimo tego efektywna, ponieważ kompilator bardzo łatwo potrafi zoptymalizować nieużywane wiązania.

We wcześniejszych wydaniach standardu C++ funkcja `divide_remainder()` mogła być zaimplementowana w następujący sposób za pomocą parametrów danych wyjściowych:

```

bool divide_remainder(int dividend, int divisor,
                    int &fraction, int &remainder);

```

Uzyskanie dostępu do wartości odbywało się wówczas z użyciem kodu, takiego jak ten:

```

int fraction, remainder;
const bool success {divide_remainder(16, 3, fraction, remainder)};
if (success) {
    std::cout << "16 / 3 to " << fraction << " plus reszta z dzielenia "
                << remainder << '\n';
}

```

Wielu programistów nadal woli to podejście zamiast zwrotu skomplikowanych struktur danych w stylu na przykład pary lub krotki. Uznają oni, że przedstawiony kod będzie działał *szybciej* ze względu na uniknięcie pośredniego kopiowania wartości. Jednak w nowoczesnych kompilatorach kopiowanie *nie* zachodzi, co oznacza zoptymalizowanie operacji.

Pomijając brakujące możliwości języka C, dostarczanie skomplikowanych struktur danych za pomocą wartości zwrótej było przez długi czas uznawane za powolną operację ze względu na konieczność inicjalizacji obiektu, a następnie skopiowania wartości do zmiennej, która po stronie wywołującego powinna zawierać wartość zwrótną. Nowoczesne kompilatory oferują optymalizację wartości zwrótej (ang. *return value optimization* — RVO), co pozwala na pominięcie pośrednich operacji kopiowania.

Ograniczanie zasięgu zmiennej do konstrukcji if i switch

W dobrym tonie jest maksymalne ograniczanie zasięgu zmiennej. Jednak czasami trzeba najpierw otrzymać wartość, która będzie mogła być dalej przetwarzana tylko po spełnieniu określonego warunku.

Do tego celu standard C++17 oferuje konstrukcje if i switch wraz z odpowiednimi procedurami inicjalizującymi.

Jak to zrobić?

W tej recepturze składnię inicjalizatora wykorzystam w obu obsługiwanych kontekstach, aby pokazać, jak dzięki niej kod może stać się znacznie bardziej przejrzysty.

- Konstrukcja if. Przyjmuję założenie, że za pomocą metody find() egzemplarza std::map trzeba znaleźć znak w mapowaniu znaków.

```
if (auto itr (character_map.find(c)); itr != character_map.end()) {
    // Iterator *itr jest poprawny i zostanie wykorzystany do pewnych operacji
} else {
    // Iterator itr jest iteratorem końcowym; nie należy przeprowadzać dereferencji
}
// Iterator itr jest tutaj niedostępny
```

- Konstrukcja switch. Oto jak przedstawia się operacja pobrania znaku z danych wejściowych i jednocześnie sprawdzenie wartości w konstrukcji switch, na przykład w celu zachowania kontroli nad grą:

```
switch (char c (getchar())); c) {
    case 'a': move_left(); break;
    case 's': move_back(); break;
    case 'w': move_fwd(); break;
```

```

    case 'd': move_right(); break;
    case 'q': quit_game(); break;

    case '0'...'9': select_tool('0' - c); break;

    default:
        std::cout << "Nieprawidłowe dane wejściowe: " << c << '\n';
}

```

Jak to działa?

Konstrukcje `if` i `switch` wraz z inicjalizatorami to w zasadzie lukier syntaktyczny. Dwa przedstawione poniżej fragmenty kodu są swoimi odpowiednikami.

Kod stosowany *przed* wydaniem standardu C++17:

```

{
    auto var (init_value);
    if (condition) {
        // Zmienna var jest dostępna w gałęzi A
    } else {
        // Zmienna var jest dostępna w gałęzi B
    }
    // Zmienna var jest nadal dostępna
}

```

Kod stosowany *od chwili* wydania standardu C++17:

```

if (auto var (init_value); condition) {
    // Zmienna var jest dostępna w gałęzi A
} else {
    // Zmienna var jest dostępna w gałęzi B
}
// Zmienna var nie jest już dostępna

```

To samo ma zastosowanie w przypadku konstrukcji `switch`.

Kod stosowany *przed* wydaniem standardu C++17:

```

{
    auto var (init_value);
    switch (var) {
    case 1: ...
    case 2: ...
    ...
    }
    // Zmienna var jest nadal dostępna
}

```

Kod stosowany *od chwili* wydania standardu C++17:

```
switch (auto var (init_value); var) {
  case 1: ...
  case 2: ...
  ...
}
// Zmienna var nie jest już dostępna
```

Ta funkcja jest niezwykle użyteczna i pozwala maksymalnie zmniejszyć zasięg zmiennej. Przed wydaniem standardu C++17 osiągnięcie pokazanego efektu było możliwe jedynie po ujęciu kodu w dodatkowy nawias klamrowy, co możesz zobaczyć w przykładach zatytułowanych „Kod stosowany przed wydaniem standardu C++17”. Krótszy cykl życiowy zmniejsza liczbę zmiennych w zasięgu, co z kolei przekłada się na większą przejrzystość kodu i jego łatwiejszą refaktoryzację.

Co dalej?

Kolejnym użytecznym przypadkiem jest ograniczenie zasięgu selekcji o znaczeniu krytycznym. Spójrz na ten fragment kodu:

```
if (std::lock_guard<std::mutex> lg {my_mutex}; some_condition) {
  // Dowlolna operacja
}
```

W pierwszej chwili wydaje się, że zostanie utworzony egzemplarz `std::lock_guard`. To jest klasa akceptująca egzemplarz `mutex` jako argument konstruktora. Ten kod powoduje nałożenie na mutex *blokady* w konstruktorze, a po opuszczeniu przez niego zasięgu destruktor powoduje *zwolnienie* tej blokady. Dzięki temu praktycznie nie można *zapomnieć* o odblokowaniu mutexu. Przed wydaniem standardu C++17 konieczne było użycie dodatkowej pary nawiasów w celu ustalenia zasięgu i miejsca zwolnienia blokady.

Innym interesującym przypadkiem jest zasięg słabych wskaźników. Spójrz na następujący fragment kodu:

```
if (auto shared_pointer (weak_pointer.lock()); shared_pointer != nullptr) {
  // Współdzielony obiekt nadal istnieje
} else {
  // Zmienna shared_pointer jest dostępna, choć to będzie wskaźnik typu null
}
// Zmienna shared_pointer nie jest już dostępna
```

To jest kolejny przykład, w którym zmienna (tutaj `shared_pointer`) niepotrzebnie wycieka do bieżącego zasięgu. Ta zmienna okazuje się zupełnie bezużyteczna poza blokiem konstrukcji warunkowej `if`.

Konstrukcja `if` wraz z inicjalizatorem jest szczególnie użyteczna podczas pracy z *przestarzałym* API wykorzystującym parametry danych wyjściowych:

```
if (DWORD exit_code; GetExitCodeProcess(process_handle, &exit_code)) {
    std::cout << "Kod wyjścia procesu to: " << exit_code << '\n';
}
// Poza konstrukcją warunkową if zmienna exit_code jest nieprzydatna
```

GetExitCodeProcess() to funkcja API jądra systemu Windows. Jej wartością zwrótną jest kod wyjścia danego procesu, ale tylko wtedy, gdy uchwyt do procesu jest prawidłowy. Po opuszczeniu bloku konstrukcji warunkowej zmienna exit_code jest bezużyteczna i dlatego nie potrzebujemy jej już w żadnym zasięgu.

Możliwość inicjalizacji zmiennych w blokach konstrukcji if okazuje się bardzo użyteczna w wielu sytuacjach, zwłaszcza podczas pracy z już przestarzałymi API, które używają parametrów danych wyjściowych.

Staraj się maksymalnie ograniczać zakres w poleceniach inicjalizacyjnych konstrukcji if i switch. Dzięki temu tworzony kod będzie zwięźlejszy, łatwiejszy w odczycie, a podczas refaktoryzacji będzie można łatwiej nim operować.

Zalety stosowania nowych reguł inicjalizacji z użyciem składni opartej na nawiasach

Wraz z wydaniem standardu C++11 pojawiła się nowa składnia inicjalizatora, wykorzystująca nawias klamrowy {}. Jej przeznaczeniem było umożliwienie *agregowania* inicjalizacji, a także używania zwykłych wywołań konstruktora. Niestety zbyt łatwo można było wyrazić nie to, co trzeba, w przypadku połączenia nowej składni z typem zmiennej auto. W standardzie C++17 wprowadzono rozbudowany zestaw reguł inicjalizatora. W tej recepturze pokażę, jak prawidłowo zainicjalizować zmienne, wykorzystując do tego składnię wprowadzoną w standardzie C++17.

Jak to zrobić?

Zmienna jest inicjalizowana w jednym kroku. Stosując składnię inicjalizatora, mamy dwie odmiennie sytuacje:

- Użycie opartej na nawiasach składni inicjalizacji *bez* określenia typu auto:

```
// Trzy identyczne sposoby zainicjalizowania zmiennej typu int
int x1 = 1;
int x2 {1};
int x3 (1);
std::vector<int> v1 {1, 2, 3}; // Wektor wraz z trzema liczbami całkowitymi: 1, 2, 3
std::vector<int> v2 = {1, 2, 3}; // To samo co powyżej
std::vector<int> v3 (10, 20); // Wektor wraz z dziesięcioma liczbami całkowitymi,
// z których każda ma wartość 20
```

- Użycie opartej na nawiasach składni inicjalizacji z określeniem typu auto:

```

auto v {1}; // Zmienna v jest typu int
auto w {1, 2}; // Błąd: w bezpośredniej inicjalizacji auto dozwolony
// jest tylko jeden element! (to jest nowość)
auto x = {1}; // Zmienna x ma postać std::initializer_list<int>
auto y = {1, 2}; // Zmienna y ma postać std::initializer_list<int>
auto z = {1, 2, 3.0}; // Błąd: nie można ustalić typu elementu

```

Jak to działa?

Bez ustalania typu auto działanie operatora {} nie powinno być zaskoczeniem, przynajmniej podczas inicjalizowania wartości zwykłych typów. Z kolei w trakcie inicjalizowania kontenerów, takich jak `std::vector`, `std::list` itd., ten inicjalizator spowoduje dopasowanie konstruktora `std::initializer_list` klasy kontenera. To się odbywa w zachłanny sposób, co oznacza brak możliwości dopasowania konstruktorów nieagregujących (konstruktor nieagregujący to zwykły konstruktor będący przeciwieństwem akceptującego listę inicjalizatora).

Na przykład `std::vector` oferuje konstruktor nieagregujący, który wypełnia dowolną liczbę elementów tą samą wartością: `std::vector<int> v(N, wartość)`. W przypadku polecenia `std::vector<int> v{N, wartość}` zostanie użyty konstruktor `initializer_list` inicjalizujący wektor wraz z dwoma elementami: `N` i `wartość`. To jest zachowanie, o którym należy wiedzieć.

Warto również pamiętać o pewnym drobiazgu dotyczącym operatora {} w porównaniu do wywołania konstruktora za pomocą zwykłego nawiasu (), a mianowicie braku niejawnej konwersji typu. W przypadku wywołań `int x(1.2);` i `int x = 1.2;` otrzymamy zmienną `x` zainicjalizowaną wraz z wartością 1. Wartość zmiennoprzecinkowa została zaokrąglona w dół i skonwertowana na postać liczby całkowitej. Natomiast wywołanie `int x{1.2};` uniemożliwia kompilację z powodu próby *ściśle* dopasowania typu konstruktora.

Można się spierać, który z przedstawionych powyżej stylów inicjalizacji jest najlepszy.

Fani składni inicjalizacji opartej na nawiasie klamrowym uważają, że takie rozwiązanie konkretnie wskazuje typ zmiennej inicjalizowanej za pomocą wywołania konstruktora, a tego rodzaju wiersz kodu nie przeprowadza ponownej inicjalizacji czegokolwiek. Ponadto użycie nawiasu klamrowego spowoduje wybór jedynie dopasowanego konstruktora, podczas gdy składnia inicjalizacji opartej na nawiasie zwykłym próbuje dopasować najbliższy konstruktor — w tym celu może nawet przeprowadzić konwersję typu.

Dodatkowa reguła wprowadzona w standardzie C++17 wpływa na inicjalizację wraz z określeniem typu auto. W C++11 nastąpi poprawne ustalenie typu zmiennej `auto x{123};` jako `std::initializer_list<int>` z tylko jednym elementem, ale rzadko o to chodziło. Natomiast w standardzie C++17 to samo polecenie powoduje, że zmienna `x` jest typu `int`.

Pamiętaj o następujących regułach:

- Wywołanie `auto zmienna {jeden_element};` określa zmienną jako takiego samego typu jak `jeden_element`.
- Wywołanie `auto zmienna {element1, element2, ...};` jest nieprawidłowe i uniemożliwia kompilację kodu.
- Wywołanie `auto zmienna = {element1, element2, ...};` określa typ zmiennej jako `std::initializer_list<T>`, gdzie `T` to taki sam typ, jaki mają wszystkie elementy listy.

Standard C++17 znacznie utrudnia przypadkowe zdefiniowanie inicjalizatora listy.

Wypróbowanie przedstawionych fragmentów kodu z różnymi kompilatorami działającymi w trybach zgodności z C++11 i C++14 pokaże, że niektóre z nich faktycznie określają zmienną `x` w wywołaniu `auto x {123};` jako typu `int`, podczas gdy inne — jako typu `std::initializer_list<int>`. Tworzenie kodu w taki sposób może prowadzić do problemów z jego przenoszeniem.

Umożliwienie konstruktorowi automatycznego określenia typu klasy szablonu

Wiele klas w C++ jest zwykle specjalizowanych dla określonych typów, które mogą być łatwo określone na podstawie typu zmiennych umieszczonych w wywołaniach konstruktorów. Przed wydaniem C++17 to nie była ustandaryzowana funkcja. C++17 pozwala kompilatorowi na *automatyczne* określenie typu klasy szablonu na podstawie wywołania konstruktora.

Jak to zrobić?

Bardzo dobrym przykładem jest tworzenie egzemplarzy typu `std::pair` i `std::tuple`. Mogą być one specjalizowane i tworzone w jednym kroku:

```
std::pair my_pair (123, "abc"); // std::pair<int, const char*>
std::tuple my_tuple (123, 12.3, "abc"); // std::tuple<int, double, const char*>
```

Jak to działa?

Oto przykład definicji klasy z automatycznym określaniem typu szablonu:

```
template <typename T1, typename T2, typename T3>
class my_wrapper {
    T1 t1;
```

```

T2 t2;
T3 t3;

public:
    explicit my_wrapper(T1 t1_, T2 t2_, T3 t3_)
        : t1{t1_}, t2{t2_}, t3{t3_}
    {}

    /* ... */
};

```

Przedstawiony powyżej fragment kodu to tylko jeszcze jedna klasa szablonu. W celu utworzenia jej egzemplarza wcześniej trzeba było używać następującego polecenia:

```
my_wrapper<int, double, const char *> wrapper {123, 1.23, "abc"};
```

Natomiast teraz można pominąć fragment dotyczący specjalizacji szablonu:

```
my_wrapper wrapper {123, 1.23, "abc"};
```

Przed wydaniem standardu C++17 takie podejście było możliwe jedynie poprzez *utworzenie funkcji pomocniczej*:

```

my_wrapper<T1, T2, T3> make_wrapper(T1 t1, T2 t2, T3 t3)
{
    return {t1, t2, t3};
}

```

Wykorzystując taką funkcję pomocniczą, można był osiągnąć efekt podobny do poprzedniego, czyli pominąć fragment dotyczący specjalizacji szablonu, jak tutaj:

```
auto wrapper (make_wrapper(123, 1.23, "abc"));
```

Biblioteka STL jest dostarczana z wieloma funkcjami pomocniczymi, takimi jak `std::make_shared()`, `std::make_unique()`, `std::make_tuple()` itd. W specyfikacji C++17 większość z nich można uznać za zbędne. Oczywiście pozostały one w celu zapewnienia wstecznej zgodności.

Co dalej?

Przed chwilą dość dokładnie przedstawiłem *niejawne określanie typu szablonu*. Jednak w pewnych sytuacjach nie można opierać się na niejawnym określaniu typu. Spójrz na ten oto fragment kodu:

```

template <typename T>
struct sum {
    T value;

    template <typename ... Ts>
    sum(Ts&& ... values) : value{(values + ...)} {}
};

```


Struktura `sum` akceptuje dowolną liczbę parametrów i dodaje je do siebie za pomocą wyrażenia `fold` (w dalszej części rozdziału przedstawię nieco więcej informacji na temat takich wyrażzeń). Obliczona suma będzie zapisana w zmiennej składowej o nazwie `value`. Mógłbyś w tym miejscu zapytać, jakiego typu jest `T`. Jeżeli nie chcesz wyraźnie zdefiniować typu `T`, będzie on zależał od typu wartości przekazanych konstruktorowi. W przypadku ciągów tekstowych typem `T` będzie `std::string`. Jeżeli konstruktor otrzyma liczby całkowite, typem będzie `int`. Natomiast w przypadku dostarczenia konstruktorowi liczb całkowitych, zmiennoprzecinkowych i podwójnej precyzji kompilator musi wybrać odpowiedni typ umożliwiający przechowywanie tych wartości bez utraty jakiegokolwiek informacji. Aby osiągnąć ten cel, można zastosować *podpowiedź niejawnego określania typu*:

```
template <typename ... Ts>
sum(Ts&& ... ts) -> sum<std::common_type_t<Ts...>>;
```

Podpowiedź niejawnego określania typu wskazuje kompilatorowi możliwość użycia egzemplarza `std::common_type` w celu ustalenia wspólnego typu obejmującego wszystkie otrzymane wartości. Spójrz na przykład użycia podpowiedzi niejawnego określania typu:

```
sum s          {1u, 2.0, 3, 4.0f};
sum string_sum {std::string{"abc"}, "def"};

std::cout << s.value          << '\n'
           << string_sum.value << '\n';
```

W pierwszym wierszu następuje utworzenie obiektu `sum` wraz z argumentami konstruktora typów `unsigned`, `double`, `int` i `float`. W wyniku użycia `std::common_type_t` kompilator wybierze `double` jako wspólny typ obejmujący wartości przekazane konstruktorowi, więc otrzymujemy egzemplarz `sum<double>`. W drugim wierszu przekazywany jest egzemplarz `std::string` i ciąg tekstowy w stylu C. Kierując się podpowiedzią niejawnego określania typu, kompilator przygotowuje egzemplarz typu `sum<std::string>`.

Po uruchomieniu powyższego fragmentu kodu nastąpi wygenerowanie wartości liczbowej 10 i ciągu tekstowego `abcdef`.

Użycie wyrażenia `constexpr-if` do uproszczenia decyzji podejmowanych podczas kompilacji

W kodzie szablonu bardzo często zachodzi potrzeba odmiennego wykonywania pewnych zadań w zależności od typu szablonu, dla którego jest on specjalizowany. Standard C++17 oferuje wyrażenie `constexpr-if`, które potrafi w takich sytuacjach *znacznie* uprościć kod.

Jak to zrobić?

W tej recepturze zaimplementuję niewielką funkcję pomocniczą szablonu klasy. Jest ona przeznaczona do pracy z różnymi specjalizacjami typu szablonu, ponieważ potrafi wybrać zupełnie inny fragment kodu w zależności od typu specjalizacji.

1. Pracę należy rozpocząć od utworzenia kodu niezależnego od specjalizacji.

W omawianym przykładzie to jest prosta klasa umożliwiająca dodanie wartości typu `U` do wartości elementu składowego typu `T` za pomocą funkcji `add()`:

```
template <typename T>
class addable
{
    T val;
public:
    addable(T v) : val{v} {}
    template <typename U>
    T add(U x) const {
        return val + x;
    }
};
```

2. Przyjmuję założenie, że typem `T` jest `std::vector<coś>`, natomiast typem `U` jest `int`. Jaki powinien być wynik dodania liczby całkowitej do wektora?

Powiedzmy, że chcę dodać tę liczbę całkowitą do każdego elementu wektora.

Taką operację przeprowadzam za pomocą pętli:

```
template <typename U>
T add(U x)
{
    auto copy (val); // Pobranie kopii elementu składowego wektora
    for (auto &n : copy) {
        n += x;
    }
    return copy;
}
```

3. Następnym i zarazem ostatnim krokiem jest *połączenie* obu światów. Jeżeli `T` jest wektorem elementów `U`, wówczas należy zastosować wariant oparty na pętli, w przeciwnym razie wystarczy zaimplementować *zwykłą* operację dodawania:

```
template <typename U>
T add(U x) const {
    if constexpr (std::is_same_v<T, std::vector<U>>) {
        auto copy (val);
        for (auto &n : copy) {
            n += x;
        }
        return copy;
    } else {
        return val + x;
    }
}
```

4. Klasa jest teraz gotowa do użycia. Spójrz, jak elegancko potrafi współpracować z różnymi typami danych, takimi jak `int`, `float`, `std::vector<int>`

i `std::vector<string>`:

```
addable<int>{1}.add(2);           // Wynik wynosi 3
addable<float>{1.0}.add(2);      // Wynik wynosi 3.0
addable<std::string>{"aa"}.add("bb"); // Wynik wynosi "aabb"

std::vector<int> v {1, 2, 3};
addable<std::vector<int>>{v}.add(10);
// Wynik wynosi std::vector<int>{11, 12, 13}

std::vector<std::string> sv {"a", "b", "c"};
addable<std::vector<std::string>>{sv}.add(std::string{"z"});
// Wynik wynosi {"az", "bz", "cz"}
```

Jak to działa?

Nowe wyrażenie `constexpr-if` działa tak samo jak zwykła konstrukcja warunkowa `if-else`. Różnica między nimi polega na tym, że w przypadku `constexpr-if` warunek będzie sprawdzony *podczas kompilacji*. Generowany przez kompilator kod uruchomieniowy programu nie będzie zawierał żadnych poleceń gałęzi `constexpr-if`. Innymi słowy: można pokusić się o stwierdzenie, że wyrażenie to działa podobnie jak makra preprocesora `#if` i `#else`. Jednak w przypadku tych makr kod nie musi być nawet doskonale sformatowany syntaktycznie. Natomiast wszystkie gałęzie w konstrukcji `constexpr-if` muszą być doskonale *sformatowane syntaktycznie*, choć jednocześnie gałęzie *nie* muszą być *syntaktycznie poprawne*.

W celu odróżnienia kodu odpowiedzialnego za dodanie wartości zmiennej `x` do wektora wykorzystam typ `std::is_same`. Wyrażenie `std::is_same<A, B>::value` przyjmuje boolowską wartość `true`, gdy `A` i `B` są tego samego typu. Warunek wykorzystany w tej recepturze ma postać `std::is_same<T, std::vector<U>::value>` i przyjmuje wartość `true`, nawet jeśli użytkownik specjalizuje klasę jako `T = std::vector<X>` i próbuje wywołać funkcję `add()` wraz z parametrem `U = X`.

Oczywiście pojedynczy blok `constexpr-if-else` może zawierać wiele poleceń. (Zwróć uwagę, że wartości `a` i `b` zależą od parametrów szablonu, a nie tylko od stałych w trakcie kompilacji).

```
if constexpr (a) {
    // Dowolna operacja
} else if constexpr (b) {
    // Inna dowolna operacja
} else {
    // Jeszcze inna dowolna operacja
}
```

W standardzie C++17 mamy wiele związanych z metaprogramowaniem sytuacji, w których utworzenie kodu i jego późniejszy odczyt będą łatwiejsze, jeśli użyjesz wyrażenia `constexpr-if`.

Co dalej?

Aby się przekonać, jak dużym usprawnieniem w języku C++ jest konstrukcja `constexpr-if`, spójrz na tę samą implementację przygotowaną jeszcze *przed* wprowadzeniem standardu C++17:

```
template <typename T>
class addable
{
    T val;

public:
    addable(T v) : val{v} {}
};

template <typename U>
std::enable_if_t<!std::is_same<T, std::vector<U>>::value, T>
add(U x) const { return val + x; }

template <typename U>
std::enable_if_t<std::is_same<T, std::vector<U>>::value,
                std::vector<U>>
add(U x) const {
    auto copy (val);
    for (auto &n : copy) {
        n += x;
    }
    return copy;
};
```

Wprawdzie bez użycia wyrażenia `constexpr-if` powyższa klasa będzie obsługiwała wszystkie interesujące nas typy, ale jej kod jest niepotrzebnie skomplikowany. Jak to rozwiązanie działa w praktyce?

Sama implementacja *dwóch różnych* funkcji `add()` wygląda na prostą. Natomiast niezwykle skomplikowanie przedstawiają się deklaracje ich wartości zwrrotnych wraz ze sztuczką w postaci wyrażenia takiego jak `std::enable_if_t<warunek, typ>`, które będzie podanego typu, jeśli warunek przyjmie wartość `true`. W przeciwnym razie `std::enable_if_t` nie określi niczego. Normalnie taka sytuacja jest uznawana za błąd, ale już wkrótce dowiesz się, dlaczego tutaj tak się nie dzieje.

W przypadku drugiej funkcji `add()` ten sam warunek jest używany w odwrotny sposób. Dlatego w danej sytuacji tylko jedna z dwóch istniejących implementacji może przyjąć wartość `true`.

Kiedy kompilator napotyka różne funkcje szablonu o tej samej nazwie i musi wybrać tylko jedną z nich, do gry wchodzi ważna reguła określana mianem **SFINAE** (ang. *substitution failure is not an error*), co oznacza: **niepowodzenie podczas podstawiania nie jest błędem**. W takim przypadku kompilator nie wygeneruje błędu, jeśli wartość zwrrotna dowolnej z wymienionych funkcji nie będzie mogła być ustalona na podstawie nieprawidłowego wyrażenia szablonu (którym tutaj jest `std::enable_if`, ponieważ jego warunek przyjmuje wartość `false`). Kompilator po prostu szuka dalej i wypróbuje *inne* implementacje funkcji. Na tym polega sztuczka, dzięki której działa przedstawione rozwiązanie.

Mnóstwo z tym kłopotu. To doskonale pokazuje, że przedstawione zadanie można znacznie łatwiej wykonać dzięki użyciu kodu zgodnego ze standardem C++17.

Włączenie bibliotek w postaci samych nagłówków z użyciem osadzonych zmiennych

W języku C++ zawsze istniała możliwość deklarowania poszczególnych funkcji jako *osadzonych*. Standard C++17 pozwala deklarować *zmiennne* jako osadzone. Dzięki temu można znacznie łatwiej zaimplementować biblioteki w postaci samych nagłówków, co wcześniej wymagało stosowania sztuczek.

Jak to zrobić?

W tej recepturze utworzę prostą klasę, która może działać w charakterze elementu składowego typowej biblioteki w postaci samych nagłówków. Celem jest dostarczenie statycznego elementu składowego i zainicjalizowanie go globalnie za pomocą słowa kluczowego `inline`. Takie podejście było niemożliwe przed wydaniem standardu C++17.

1. Klasa `process_monitor` powinna zawierać statyczne elementy składowe i jednocześnie być dostępna globalnie. Takie rozwiązanie wymagałoby podwójnego definiowania symboli.

```
//foo_lib.hpp
class process_monitor {
public:
    static const std::string standard_string
        {"pewien statyczny dostępny globalnie ciąg tekstowy"};
};
process_monitor global_process_monitor;
```

2. Jeżeli teraz plik zawierający powyższą klasę będzie dołączony w wielu plikach `.cpp`, aby przeprowadzić ich kompilację i linkowanie, wówczas etap linkowania zakończy się niepowodzeniem. Rozwiązaniem problemu jest dodanie słowa kluczowego `inline`:

```
//foo_lib.hpp
class process_monitor {
public:
    static const inline std::string standard_string
        {"pewien statyczny dostępny globalnie ciąg tekstowy"};
};

inline process_monitor global_process_monitor;
```

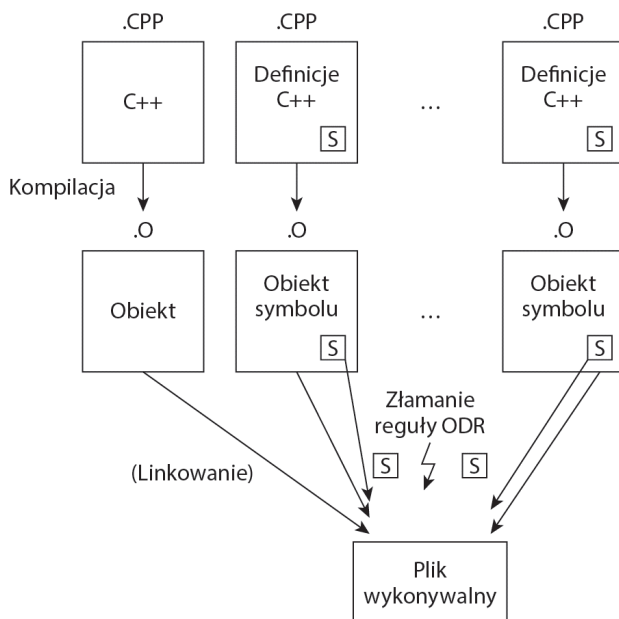
Voilà, o to chodziło!

Jak to działa?

Program w C++ często składa się z wielu plików kodu źródłowego (mają one rozszerzenie `.cpp` lub `.cc`). Najczęściej są one pojedynczo komplikowane na postać plików modułów lub obiektów (w takim przypadku rozszerzeniem pliku jest `.o`). Ostatnim krokiem jest linkowanie wszystkich plików modułów i obiektów w celu wygenerowania pojedynczego pliku wykonywalnego bądź biblioteki współdzielonej lub statycznej.

Jeżeli na etapie linkowania definicja danego symbolu zostanie znaleziona *wielokrotnie*, wówczas mamy do czynienia z błędem. Przyjmując założenie o istnieniu funkcji z sygnaturą taką jak `int foo();`. Kiedy dwa moduły definiują tę samą funkcję, która z nich jest prawidłowa? Linker nie może wybierać losowo implementacji. Cóż, mógłby, ale jest mało prawdopodobne, aby takiego podejścia oczekiwał jakikolwiek programista.

Tradycyjnym sposobem dostarczania globalnie dostępnych funkcji jest ich *deklarowanie* w plikach nagłówkowych, które mogą być dołączane przez dowolny moduł C++ potrzebujący tych funkcji. Definicja każdej funkcji dostępnej globalnie będzie umieszczona *raz* w oddzielnym pliku modułu. Następnie te pliki będą linkowane wraz z modułami, w których mają być używane dane funkcje. To podejście jest określane mianem **reguły ODR** (ang. *one definition rule*), czyli **reguły pojedynczej definicji**. W sposób graficzny tę regułę pokazałem na rysunku 1.1.



Rysunek 1.1. Reguła pojedynczej definicji

Jeżeli to byłoby jedyne rozwiązanie, wówczas zabrakłoby możliwości dostarczania bibliotek w postaci samych nagłówków. Biblioteki okazują się niezwykle użyteczne, ponieważ mogą być dołączane w dowolnym pliku programu C++ za pomocą polecenia `#include`, po którego wykonaniu są natychmiast dostępne. W celu użycia biblioteki w postaci innej niż samego nagłówka programista musi zaadaptować skrypt kompilacji, aby linker dołączył moduły biblioteki do plików modułów programu. To jest szczególnie niewygodne w przypadku bibliotek zawierających niewielką liczbę małych funkcji.

Z pomocą przychodzi tutaj słowo kluczowe `inline`, które można zastosować do zdefiniowania wyjątku umożliwiającego użycie w różnych modułach *wielu* definicji tego samego symbolu. Jeżeli linker znajdzie wiele symboli o takiej samej sygnaturze, ale zadeklarowanych jako osadzone, wówczas wybierze pierwszy, uznając, że pozostałe symbole mają taką samą definicję. Zagwarantowanie takich samych definicji dla wszystkich osadzonych symboli jest w zasadzie *obietnicą* składaną przez programistę.

Powracam teraz do omawianej tutaj receptury — linker znajdzie symbol `process_monitor::↪standard_string` w każdym module importującym `foo_lib.hpp`. Bez słowa kluczowego `inline` nie będzie wiedział, który z nich powinien zostać użyty, więc spowoduje wygenerowanie błędu. To samo dotyczy również symbolu `global_process_monitor`. Który z nich jest prawidłowy?

Po zadeklarowaniu obu symboli za pomocą słowa kluczowego `inline` nastąpi akceptacja pierwszego wystąpienia każdego symbolu i *odrzućenie* wszystkich kolejnych wystąpień.

Przed wprowadzeniem standardu C++17 jedynym przejrzystym sposobem implementacji przedstawionego powyżej rozwiązania było dostarczenie symbolu za pomocą dodatkowego pliku modułu C++. To wymagało od użytkowników biblioteki dołączania tego pliku na etapie linkowania.

Słowo kluczowe `inline` tradycyjnie miało również *inną* funkcję — wskazywało kompilatorowi możliwość usunięcia funkcji poprzez pobranie jej implementacji i bezpośrednie wstawienie w miejscu wywołania tej funkcji. W ten sposób kod zawierał o jedno wywołanie funkcji mniej, co często mogło przekładać się na większą wydajność działania. Jeżeli funkcja jest niewielka, podzespół wynikowy również będzie mniejszy (przy założeniu, że liczba poleceń odpowiedzialnych za wywołanie tej funkcji, umieszczenie i pobranie jej ze stosu itd. jest większa niż faktyczny kod funkcji). Natomiast w przypadku, gdy osadzona funkcja jest duża, wielkość pliku binarnego ulegnie zwiększeniu i ostatecznie może się zdarzyć, że kod nie będzie działał szybciej. Dlatego kompilator traktuje słowo kluczowe `inline` jedynie jako wskazówkę i może eliminować wywołania funkcji poprzez osadzanie ich w kodzie. Czasami może też osadzać w kodzie pewne funkcje, nawet jeśli *nie* zostały przez programistę określone jako osadzone.

Co dalej?

Jednym z możliwych rozwiązań problemu przed wprowadzeniem standardu C++17 było użycie funkcji statycznej zwracającej odwołanie do obiektu `static`. Spójrz na fragment kodu:

```

class foo {
public:
    static std::string& standard_string() {
        static std::string s {"Dowolny standardowy ciąg tekstowy."};
        return s;
    }
};

```

W ten sposób można całkowicie prawidłowo dołączyć plik nagłówkowy w wielu modułach i nadal zapewnić sobie wszędzie dostęp do dokładnie tego samego egzemplarza. Jednak wspomniany obiekt *nie* był tworzony *natychmiast* po uruchomieniu programu, a dopiero po pierwszym wywołaniu funkcji getter. W niektórych przypadkach to może stanowić problem. Wyobraź sobie sytuację, w której konstruktor statycznego, globalnie dostępnego obiektu ma wykonać pewne ważne zadanie (podobnie jak w przypadku przykładowej biblioteki w tej recepturze) *w trakcie uruchamiania programu*. Jednak ze względu na to, że wywołanie następuje prawie na końcu działania programu, wykonanie zadania się nie udało.

Innym rozwiązaniem jest zdefiniowanie niebędącej szablonem klasy foo jako klasy szablonu, aby w ten sposób móc skorzystać z tych samych zalet, które oferują szablony.

Obu strategii można uniknąć w kodzie źródłowym zgodnym ze standardem C++17.

Implementowanie za pomocą wyrażeń fold przydatnych funkcji pomocniczych

Od chwili wydania standardu C++11 mamy pakiety parametrów wariadycznych, co pozwala zaimplementować funkcje akceptujące dowolną liczbę parametrów. Czasami te parametry są łączone w jednym wyrażeniu w celu pobrania wyniku wykonania funkcji. To zadanie stało się znacznie łatwiejsze w C++17 dzięki pojawieniu się wyrażeń fold.

Jak to zrobić?

Przystępuję do zaimplementowania funkcji pobierającej dowolną liczbę parametrów i zwracającej ich sumę.

1. Pracę rozpoczynam od zdefiniowania sygnatury funkcji:

```

template <typename ... Ts>
    auto sum(Ts ... ts);

```

2. W tym momencie mam pakiet parametrów ts, funkcja powinna rozwinąć wszystkie parametry i obliczyć ich sumę za pomocą wyrażenia fold. Jeżeli w celu zastosowania wszystkich wartości pakietu parametrów zostanie użyty operator (w omawianym przykładzie to operator +) wraz z wielokropkiem, wówczas wyrażenie należy ująć w nawias:


```
template <typename ... Ts>
auto sum(Ts ... ts)
{
    return (ts + ...);
}
```

3. W tym momencie można użyć następującego wywołania:

```
int the_sum {sum(1, 2, 3, 4, 5)}; //Wartość: 15
```

4. Przedstawione rozwiązanie działa nie tylko z liczbami całkowitymi, ale również z każdym innym typem implementującym funkcję operator+(), na przykład std::string:

```
std::string a {"Witaj, "};
std::string b {"świecie!"};
std::cout << sum(a, b) << '\n'; //Dane wyjściowe: Witaj, świecie!
```

Jak to działa?

W przedstawionym rozwiązaniu po prostu zastosowałem dla parametrów funkcji rekurencyjny operator dwuargumentowy +. Takie działanie jest określane mianem **foldingu**. Standard C++17 oferuje obsługę **wyrażeń fold**, dzięki którym tę samą ideę można wyrazić za pomocą mniejszej ilości kodu.

Przedstawiony rodzaj wyrażenia jest nazywany **foldingiem jednoargumentowym**. Standard C++17 oferuje obsługę foldingu pakietów parametrów wraz z następującymi operatorami dwuargumentowymi: +, -, *, /, %, ^, &, |, =, <, >, <<, >>, +=, -=, *=, /=, %=, ^=, &=, |=, <<=, >>=, ==, !=, <=, >=, &&, ||, ,, .*, ->*

Warto w tym miejscu wspomnieć, że w omawianym tutaj przykładowym fragmencie kodu nie ma żadnego znaczenia, który z dwóch zapisów zostanie użyty — (ts + ...) czy (... + ts). Jednak istnieje różnica, która może mieć znaczenie w innych sytuacjach: jeśli wielokropek znajduje się *po prawej* stronie operatora, wówczas mamy do czynienia z *prawym foldingiem*, natomiast wielokropek po lewej stronie operatora oznacza istnienie *lewego* foldingu.

W omawianym przykładzie lewy folding zostanie rozwinięty do postaci 1 + (2 + (3 + (4 + 5))), natomiast prawy do postaci (((1 + 2) + 3) + 4) + 5. W zależności od użytego operatora to może mieć znaczenie. W przypadku dodawania liczb nie ma żadnego znaczenia.

Co dalej?

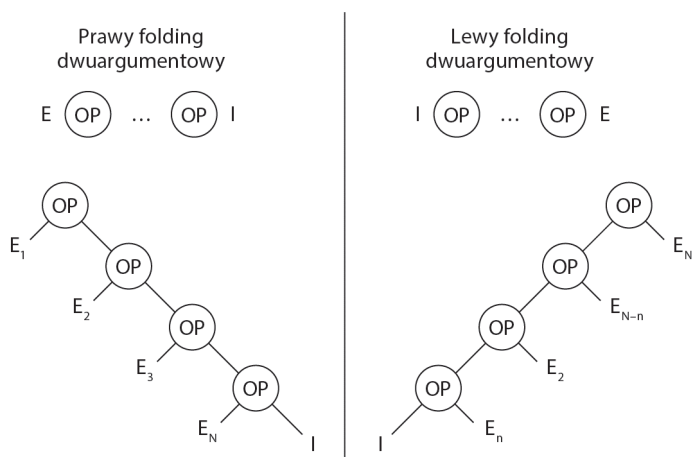
W przypadku wywołania funkcji sum() *bez* parametrów parametr wariadyczny nie zawiera żadnych wartości, które mogłyby zostać poddane foldingowi. W większości operatorów taka sytuacja spowoduje wygenerowanie błędu (choć nie we wszystkich, do tego jeszcze powrócę w niniejszym rozdziale). Trzeba wówczas zdecydować, czy błąd ma pozostać, czy ma zostać zwrócona pusta suma. Oczywiście jest, że suma niczego wynosi 0.

Oto jak to można zrobić:

```
template <typename ... Ts>
auto sum(Ts ... ts)
{
    return (ts + ... + 0);
}
```

W ten sposób wynik wywołania funkcji `sum()` wynosi 0, natomiast wywołanie `sum(1, 2, 3)` zostaje rozwinięte do postaci `(1 + (2 + (3 + 0)))`. Takie wyrażenie wraz z wartością początkową jest określane mianem **foldingu dwuargumentowego**.

W tym przypadku zapis również nie ma znaczenia — `(ts + ... + 0)` lub `(0 + ... + ts)` — choć wpływa na rodzaj foldingu dwuargumentowego: może być *prawy* lub *lewy*. Spójrz na rysunek 1.2.



Rysunek 1.2. Przykłady lewego i prawego foldingu dwuargumentowego

Podczas użycia foldingu dwuargumentowego w celu zaimplementowania obsługi sytuacji, w której nie zostanie podany parametr, element *identyczności* jest najczęściej ważny. W omawianym przykładzie dodanie 0 do dowolnej liczby niczego nie zmienia, więc 0 staje się elementem identyczności. Z powodu tej właściwości za pomocą operatora `+` lub `-` można używać 0 w dowolnym wyrażeniu fold. W przypadku wywołania funkcji bez parametrów dodanie 0 powoduje otrzymanie wyniku wynoszącego 0. Z matematycznego punktu widzenia to jest poprawne. Natomiast z perspektywy implementacji na podstawie wymagań konieczne jest zdefiniowanie, co należy uznać za poprawne.

Ta sama reguła ma zastosowanie w przypadku mnożenia. Tutaj elementem identyczności jest 1:

```
template <typename ... Ts>
auto product(Ts ... ts)
{
    return (ts * ... * 1);
}
```

Wynikiem wywołania funkcji `product(2, 3)` jest 6, natomiast wynik wywołania tej samej funkcji, ale bez parametrów wynosi 1.

Operatory logiczne **i** (`&&`) oraz **lub** (`||`) są dostarczane wraz z *wbudowanymi* elementami identyczności. Folding pustego pakietu parametrów dla operatora `&&` powoduje wygenerowanie wartości `true`, natomiast w przypadku operatora `||` będzie to wartość `false`.

Kolejny operator, który domyślnie stosuje pewne wyrażenie w przypadku użycia operatora bez parametrów, to przecinek. Jeżeli użyjesz go bez parametrów, domyślnie wywoła `void()`.

Przejdę teraz do kilku innych niewielkich funkcji pomocniczych, które można zaimplementować za pomocą omówionej funkcjonalności.

Dopasowanie zakresu względem poszczególnych elementów

Oto funkcja, która informuje, czy pewien zakres zawiera *co najmniej* jedną z wartości podanych jako parametr wariadyczny:

```
template <typename R, typename ... Ts>
auto matches(const R& range, Ts ... ts)
{
    return (std::count(std::begin(range), std::end(range), ts) + ...);
}
```

Ta funkcja pomocnicza wykorzystuje funkcję `std::count()` biblioteki STL i pobiera trzy parametry. Dwa pierwsze to iteratory *początkowy* i *końcowy* pewnego zakresu, trzeci zaś zawiera *wartość* porównywaną ze wszystkimi elementami we wskazanym zakresie. Metoda `std::count()` zwraca liczbę elementów w zakresie, które są identyczne z elementem podanym w trzecim parametrze wywołania `std::count()`.

W naszym wyrażeniu `fold` zawsze znajdują się te same iteratory *początkowy* i *końcowy* zakresu przekazywanego funkcji `std::count()`. Natomiast jako trzeciego parametru używamy po jednej wartości pochodzącej z pakietu parametru wariadycznego. Ostatecznie funkcja sumuje wyniki otrzymane dla poszczególnych wartości parametru wariadycznego i przekazuje je komponentowi wywołującemu.

Omawianej tutaj funkcji pomocniczej można użyć w następujący sposób:

```
std::vector<int> v {1, 2, 3, 4, 5};

matches(v,          2, 5);           // Wynik: 2
matches(v,         100, 200);       // Wynik: 0
matches("abcdefg", 'x', 'y', 'z'); // Wynik: 0
matches("abcdefg", 'a', 'd', 'f'); // Wynik: 3
```

Jak widzisz, funkcja pomocnicza `matches()` jest dość wszechstronna — można ją wywołać bezpośrednio dla wektorów, a nawet ciągów tekstowych. Działa również z metodami inicjalizacyjnymi list, czyli egzemplarzami `std::list`, `std::array`, `std::set` itd.

Sprawdzenie, czy operacja wstawienia wielu elementów do zbioru zakończyła się pomyślnie

Utworzę teraz funkcję pomocniczą wstawiającą dowolną liczbę parametrów wariadycznych do zbioru `std::set` i zwracającą informacje o tym, czy wstawienie wszystkich elementów zakończyło się *pomyślnie*:

```
template <typename T, typename ... Ts>
bool insert_all(T &set, Ts ... ts)
{
    return (set.insert(ts).second && ...);
}
```

W jaki sposób działa powyższy fragment kodu? Funkcja `insert()` zbioru `std::set` ma następującą sygnaturę:

```
std::pair<iterator, bool> insert(const typ_wartości& wartość);
```

Zgodnie z dokumentacją, w trakcie operacji wstawiania elementu, wartością zwrótną funkcji `insert()` jest para iterator i zmienna boolowska. Jeżeli wstawienie się powiodło, zmienna boolowska przyjmie wartość `true`, a iterator będzie wskazywał *nowy element* w zbiorze. W przeciwnym razie iterator wskazuje *istniejący* element, który może *kolidować* z elementem przeznaczonym do wstawienia.

Po operacji wstawienia omawiana tutaj funkcja pomocnicza uzyskuje dostęp do właściwości `.second`, która jest po prostu zmienną boolowską odzwierciedlającą sukces lub niepowodzenie. Jeżeli wszystkie operacje wstawienia doprowadziły do przypisania wartości `true` zmiennej boolowskiej w każdej parze wyniku działania funkcji, oznacza to, że wstawienie elementów zakończyło się powodzeniem. Wyrażenie `fold` za pomocą operatora `&&` łączy wyniki wszystkich operacji wstawienia i zwraca ostateczny wynik.

Omawianej tutaj funkcji pomocniczej można użyć w następujący sposób:

```
std::set<int> my_set {1, 2, 3};

insert_all(my_set, 4, 5, 6); //Wartością zwrótną jest true
insert_all(my_set, 7, 8, 2); //Wartością zwrótną jest false, ponieważ element 2 powoduje kolizję z istniejącym
```

Jeżeli spróbujesz wstawić na przykład trzy elementy, a drugi z nich nie może zostać wstawiony, wówczas wyrażenie `&& ...` skróci wykonywanie operacji i wstrzyma przetwarzanie pozostałych elementów:

```
std::set<int> my_set {1, 2, 3};

insert_all(my_set, 4, 2, 5); //Wartością zwrótną jest false
// Teraz zbiór zawiera elementy {1, 2, 3, 4}, ale bez 5!
```

Sprawdzenie, czy wszystkie parametry znajdują się w danym zakresie

Skoro można ustalić, czy *jedna* zmienna znajduje się w określonym zakresie, za pomocą wyrażenia `fold` to samo można sprawdzić dla wielu zmiennych:

```
template <typename T, typename ... Ts>
bool within(T min, T max, Ts ...ts)
{
    return ((min <= ts && ts <= max) && ...);
}
```

Wyrażenie `(min <= ts && ts <= max)` sprawdza, czy poszczególne wartości parametru wariacyjnego znajdują się w zakresie definiowanym przez `min` i `max` (łącznie z `min` i `max`). Zdecydowałem się na użycie operatora `&&`, aby zmniejszyć liczbę wyników boolowskich do jednego. Jeżeli wszystkie wyniki cząstkowe przyjęły wartość `true`, wynik końcowy również będzie miał wartość `true`.

Oto omawiana funkcja pomocnicza w akcji:

```
within( 10, 20, 1, 15, 30); //--> false
within( 10, 20, 11, 12, 13); //--> true
within(5.0, 5.5, 5.1, 5.2, 5.3) //--> true
```

Ta funkcja pomocnicza jest niezwykle wszechstronna, ponieważ jedynym ograniczeniem nakładanym na typ jest możliwość *porównania* za pomocą operatora `<=`. To wymaganie jest spełnione między innymi przez egzemplarz `std::string`, jak pokazałem w następującym fragmencie kodu.

```
std::string aaa {"aaa"};
std::string bcd {"bcd"};
std::string def {"def"};
std::string zzz {"zzz"};

within(aaa, zzz, bcd, def); //--> true
within(aaa, def, bcd, zzz); //--> false
```

Umieszczenie wielu elementów w wektorze

Istnieje również możliwość utworzenia funkcji pomocniczej nie zmniejszającej liczby wyników, ale przetwarzającej wiele akcji tego samego rodzaju. Przykładem może być tutaj wstawianie elementów do kontenera `std::vector`, co nie powoduje zwrotu żadnego wyniku — wywołanie `std::vector::insert()` sygnalizuje błąd poprzez zgłoszenie wyjątku.

```
template <typename T, typename ... Ts>
void insert_all(std::vector<T> &vec, Ts ... ts)
{
    (vec.push_back(ts), ...);
}

int main()
```

```
{
    std::vector<int> v {1, 2, 3};
    insert_all(v, 4, 5, 6);
}
```

Zwróć uwagę na użycie operatora `,` w celu rozwinięcia parametru wariadycznego na poszczególne wywołania `vec.push_back(...)` bez zastosowania foldingu dla rzeczywistego wyniku. Prezentowana tutaj funkcja działa również w przypadku *pustego* parametru wariadycznego, ponieważ operator `,` ma niejawnie zdefiniowany element identyczności, `void()`, oznaczający *nie rób nic*.

Skorowidz

A

adapter
 iteratora, 107, 123
 odwrotnego, 114
 std::back_insert_iterator, 109
 std::front_insert_iterator, 109
 std::insert_iterator, 109
 std::istream_iterator, 110
 std::ostream_iterator, 110
 kontenera, 50
agregowanie inicjalizacji, 29
algorytm, *Patrz także* funkcja
 Boyera i Moore'a, 189
 compress(), 239
 decompress(), 239
 gather(), 231, 233
 generate(), 364
 max_element(), 350
 next_permutation(), 194
 przekształcenia Fouriera, 211
 remove_multi_whitespace(), 235
 scoped_lock, 381
 split(), 227, 229
algorytmy
 biblioteki STL, 161, 167, 366
 kompresji, 238
 sortowania, 170
 w kategoriach iteratorów, 110
 wyszukiwania, 176
aliasing, 190
ASCII, 221, 406
automatyczna
 obsługa współdzielonej pamięci, 333
 obsługa zasobów, 329
 zmiana nazwy pliku, 434

automatyczne
 określanie typu, 31
 sprawdzanie kodu iteratora, 119
 stosowanie programowania równoległego, 363
awaria
 sygnalizowanie, 307

B

bezpieczeństwo typu, 320
białe znaki, 235
 spacja, 249
 tabulator, 249
 znak nowego wiersza, 249
biblioteka
 Boost, 129
 ranges, 129
 STL, 32
 kontenery, 47
 włączenie, 37
blok
 constexpr-if-else, 35
 try, 295
blokady, 379
 współdzielone, 375
błąd
 kompilacji, 106, 122
 sumy wektorów, 218
błędy strumienia, 293
bufor stringstream, 385

C

ciąg Fibonacciego, 111
ciąg tekstowy
 kompresja i dekompresja, 238
 konkatenacja, 245, 248

ciąg tekstowy
 przekształcanie, 245
 tworzenie, 245
 usuwanie białych znaków, 235, 248
 własne klasy, 281

constexpr-if, 33, 35

czas
 bezwzględne wartości, 304
 jednostki, 298
 uśpienie programu, 369
 względne wartości, 304

D

debugowanie, 119, 121

dekompresja ciągów tekstowych, 238

dereferencja, 78

deserializacja obiektów, 268, 272

destruktor, 279

dołączenia symboliczne, 442

domknięcie, 131

dostęp do wektora, 58

drzewo, 201
 generator wypowiedzi, 206
 wyszukiwania, 49, 68

dynamiczne alokowanie, 333

dziedziczenie, 281

E

egzemplarz, *Patrz* klasa

F

fala kwadratowa, 214

FIFO, first in, first out, 93

filtrowanie, 154, 171
 duplikatów, 76

fold, 40

folding, 41, 153
 dwuargumentowy, 42
 jednoargumentowy, 41

formatowanie
 danych wyjściowych, 260
 liczb, 291
 wejścia – wyjścia, 267

fraktal, 221

funkcja, *Patrz także* algorytm
 absolute_path(), 425
 accumulate(), 272

async_adapter(), 415

asynchrone(), 413, 415, 417

at(), 59

begin(), 118

canonical(), 422, 424, 425

cartesian(), 157

concat(), 413

compress(), 240

cout.flush(), 387

create(), 413

create_foo(), 343

create_hard_link(), 445

create_symlink(), 445

current_path(), 424, 425

cv.wait(), 397

decompress(), 240

destroy_foo(), 343

end(), 118

eq(), 283

equivalence(), 425

evaluate_rpn(), 83

exclusive_throw(), 377

ext_stats(), 442

extract(), 71, 72

f(), 412

file_info(), 429

file_size(), 439

filesystem::exists(), 421

filesystem::canonical(), 421

filesystem::exists(), 422

find(), 186, 274

find_first_not_of(), 252

Foo::create_foo(), 344

gather(), 233

gather_sort(), 233

gen_cosine(), 217

get(), 394

get_input(), 300

getline(), 432

histogram(), 394

insert(), 71, 72

insert_sorted(), 61

is_sorted(), 61

key(), 72

lower_bound(), 62

lt(), 283

make_shared(), 336

make_unique(), 336

matches(), 433

multicall(), 147

now(), 302
 operator<<(), 318, 385
 operator>>(), 268, 270
 population_higher_than(), 179
 predicate(), 399
 print(), 202, 253, 254
 print_cout(), 386
 print_exclusive(), 376
 process_item(), 331
 push(), 82, 94
 q.pop(), 95
 q.push(), 95
 q.top(), 95
 quick_remove_at(), 55, 57
 recursive_directory_iterator(), 433, 441
 reduce_dupes(), 445
 regex_replace(), 435
 relative_path(), 426
 release(), 332
 remove_suffix(), 252
 scale(), 225
 setw(), 261
 signal_from_generator(), 217
 size_string(), 438, 440
 sleep_for(), 370
 sleep_until(), 370
 status(), 427
 std::accumulate(), 150, 213, 217, 271
 std::async, 393, 406, 413, 415
 std::basic_string, 282
 std::binary_search(), 180
 std::call_once, 388
 std::cerr, 277
 std::char_traits, 281
 std::chrono, 304
 std::chrono::duration, 299
 std::complex, 212, 222
 std::condition_variable, 362, 395, 400
 std::copy(), 164
 std::copy_if(), 174
 std::equal_range(), 176, 180
 std::filesystem::path, 420
 std::find(), 176, 180, 228, 238
 std::find_if(), 180
 std::function, 138
 std::future, 410
 std::inner_product(), 221
 std::is_sorted(), 170
 std::lower_bound(), 180
 std::merge(), 196
 std::optional, 307
 std::ostream, 246
 std::ostream_iterator, 273
 std::ostrstream, 247
 std::partial_sort(), 167, 170
 std::partition(), 170
 std::ratio, 298
 std::regex_iterator, 290
 std::remove(), 171, 174
 std::remove_copy(), 174
 std::remove_copy_if(), 173
 std::remove_if(), 171
 std::replace(), 172, 174
 std::replace_copy(), 174
 std::shared_lock, 375
 std::shared_ptr, 333
 std::shuffle(), 170
 std::sort(), 167, 171
 std::stream_iterator, 77
 std::string, 249, 251
 std::string_view, 248
 std::stringstream, 248
 std::thread, 362
 std::transform(), 174, 217
 std::tuple, 313
 std::unique, 237
 std::unique_lock, 375
 std::unique_ptr, 329
 std::upper_bound(), 180
 std::variant, 323
 strerror(), 295
 string::find_first_of(), 250
 string::substring(), 251
 sum_min_max_avg(), 318
 system_complete(), 424, 425
 this_thread::sleep_for(), 370
 this_thread::sleep_until(), 370
 to_iteration_count(), 227
 toupper(), 247
 transform_if(), 150
 trie::subtrie(), 210
 try_emplace(), 63, 65, 66, 444
 try_lock(), 384
 twice(), 413
 unique_lock(), 376
 weak_ptr_info(), 340
 word_num(), 275
 wordcount(), 258
 x.join(), 374
 zip(), 319

funkcje

- binarnego drzewa wyszukiwania, 179
- foldingu, 153
- łączenie, 141
- oparte na wyrażeniu lambda, 133
- osadzone, 37
- pomocnicze, 40
- wywoływanie, 146
- z krotkami, 311

G

generator

- fraktala, 221
- podpowiedzi, 206

generowanie

- iloczynu kartezjańskiego, 155
- liczb losowych, 348, 354
- permutacji, 193
- równoległe fraktala, 406
- sygnału, 217
- wartości hash, 443

H

- harmonogram działania programu, 416
- hash, 73
- Haskell, 124
- hermetyzowanie kodu, 133

I

iloczyn kartezjański, 155

implementowanie

- algorytmów w kategoriach iteratorów, 110
- fraktala, 221
- funkcji `transform_if()`, 150
- kalkulatora RPN, 79
- klasy drzewa trie, 201
- licznika, 85
- listy, 93
- obiektu wyszukiwania, 187
- podpowiedzi, 206
- programu
 - do wyszukiwania zdań, 88
 - łączenia słowników, 195
 - normalizującego ścieżkę dostępu, 420
 - obliczającego wielkość katalogu, 437

- wyszukującego dane, 431
- zmieniającego nazwy plików, 434
- zmniejszającego wielkość katalogu, 442
- wzorca producent – konsument, 395, 400
- wzoru przekształcenia Fouriera, 211

inicjalizacja

- odłożona, 388
- skomplikowanych obiektów, 266

inicjalizator, 29

iterator, 97, 164

- danych wejściowych, 100
- danych wyjściowych, 100
- dostępu losowego, 100
- dwukierunkowy, 100
- modyfikowalny, 100
- podpowiedzi, 68
- poruszający się tylko do przodu, 100
- sąsiadujący, 100
- `std::istream`, 269
- `std::ostream`, 273
- strumieni danych, 276
- własnego zakresu, 101

iteratory

- adaptery, 107
- automatyczne sprawdzanie kodu, 119
- kończenie działania, 116
- odwrotne, 114
- opakowanie, 108
- tworzenie, 104
- użycie wartownika, 116
- własny adapter, 123

J

jednostki czasu, 298

język

- Haskell, 124
- Python, 124

K

kalkulator RPN, 79

kanoniczna ścieżka dostępu, 423

katalog

- bieżący, 420, 429
- nadrzędny, 420
- obliczanie wielkości, 437
- zmniejszanie wielkości, 442

kategorie iteratorów, 99

klasa

- allocator<char>, 286
- char_traits<char>, 286
- filesystem::path, 423
- format_guard, 292
- lock_guard, 379
- mutex, 378
- recursive_mutex, 378
- recursive_timed_mutex, 378
- scoped_lock, 379, 384
- shared_mutex, 378
- shared_timed_mutex, 378
- sregex_token_iterator, 288
- std::async, 406
- std::basic_string, 282
- std::call_once, 388
- std::cerr, 277
- std::char_traits, 281
- std::chrono, 304
- std::chrono::duration, 299
- std::complex, 212, 222
- std::condition_variable, 362, 395, 400
- std::filesystem::path, 420
- std::function, 138
- std::future, 410
- std::optional, 307
- std::ostream, 246
- std::ostream_iterator, 273
- std::ostringstream, 247
- std::ratio, 298
- std::regex_iterator, 290
- std::shared_lock, 375
- std::shared_ptr, 333
- std::stream_iterator, 77
- std::string, 249, 251
- std::string_view, 248
- std::stringstream, 248
- std::thread, 362
- std::tuple, 313
- std::unique_lock, 375
- std::unique_ptr, 329
- std::variant, 323
- std::timed_mutex, 378
- std::unique_lock, 379

klasy

- biblioteki STL, 244
- blokad, 379
- muteksu, 378
- narzędziowe, 297

kolejka, 93

- dwustronna, 49
- priorytetowa
 - lista, 93
 - odwołanie do elementu, 95
 - usuwanie pierwszego elementu, 95
 - wstawianie elementu, 95

komponenty sprawdzania poprawności, 122

kompresja ciągów tekstowych, 238

koniunkcja, 144

konkatenacja

- ciągów tekstowych, 245, 248
- funkcji, 141

konstrukcja

- constexpr-if, 33, 35
- if, 26
- switch, 26
- try-catch, 423
- warunkowa, 332

konstruktor, 278

- domyślny, 278

konsumenty, 401

kontener

- std::deque, 49
- std::forward_list, 49
- std::list, 49
- std::map, 49, 63, 68, 70, 85
- std::multimap, 49, 91
- std::multiset, 49
- std::priority_queue, 50, 93
- std::queue, 50, 93
- std::set, 49, 76, 88
- std::stack, 50, 79
- std::std array, 48
- std::unordered_map, 50, 73
- std::unordered_multiset, 50
- std::unordered_set, 50
- std::vector, 48, 50, 54, 58

kontenery

- adapter kontenera, 50
- asocjacyjne, 49
- drzewo wyszukiwania, 49
- listy, 49
- przekształcanie zawartości, 174
- sortowanie, 167
- tabela wartości hash, 50
- tablice, 48
- usuwanie elementów, 171
- wypełnianie, 269

konwerter stoull, 350
 konwertowanie jednostek czasu, 298, 304
 kopiowanie elementów między kontenerami, 163
 krotki, 311, 313
 funkcja operator <<(), 318
 funkcja zip(), 319

L

liczby
 losowe, 348, 354
 zespolone, 222
 licznik, 85, 349
 odwołań, 337
 linkowanie, 37, 38, 39
 lista, 49, 93
 przechwytywania, 136
 constexpr, 137
 exception atrybut, 137
 mutable, 137
 typ wartości zwrotnej, 137
 logiczna koniunkcja, 144

Ł

łączenie
 algorytmów, 231
 funkcji, 141
 słowników, 195

M

magazyn danych
 listy, 49
 tablice, 48
 makro _GLIBCXX_DEBUG, 120
 manipulatory strumienia wejścia – wyjścia, 260
 mapa
 algorytm wyszukiwania, 68
 licznik słów, 85
 modyfikowanie kluczy elementów, 69
 podpowiedzi wstawiania, 66
 wstawianie elementów, 63
 wyszukiwanie elementów, 66
 mapowanie
 operacji, 84
 wartości typu string, 84
 metoda, *Patrz* funkcja

migawka czasu, 300
 modyfikator
 boolalpha, 265
 dec, 264
 defaultfloat, 264
 endl, 265
 ends, 265
 fixed, 264
 flush, 265
 hexfloat, 264
 internal, 264
 left, 264
 noboolalpha, 265
 noshowbase, 265
 noshowpoint, 264
 noshowpos, 264
 noskipws, 265
 nounitbuf, 265
 nouppercase, 265
 quoted(string), 265
 right, 264
 scientific, 264
 setbase(int n), 264
 setfill(char c), 264
 setprecision(int n), 264
 setw(int n), 264
 showbase, 265
 showpoint, 264
 showpos, 264
 skipws, 265
 unitbuf, 265
 uppercase, 265
 ws, 265
 modyfikowanie kluczy elementów mapy, 69
 muteks, 376, 378

N

narzędzie grep, 431
 nawias klamrowy, 29
 nazwy plików, 436
 niejawne określanie typu, 33
 niestandardowy operator porównania, 88
 normalizacja ścieżki dostępu, 420
 numer błędu, 295

O

obiekt std
 function<void()>, 140
 obiekty
 tworzenie, 280
 usuwanie, 280
 współdzielenie wartości składowych, 345
 współdzielone, 338
 zegara, 302
 obliczanie
 błędu sumy, 218
 danych statystycznych, 440
 wielkości katalogu, 437
 obraz ASCII, 221
 obsługa
 błędu, 60, 81
 stosu, 82
 wektoryzacji, 363
 współbieżności, 217
 współdzielonej pamięci, 333
 wyjątków, 294, 296
 wyrażeń lambda, 133
 zasobów, 329
 zasobów przestarzałych API, 342
 odczyt
 danych wejściowych, 210
 wartości z danych wejściowych, 254
 odkładanie inicjalizacji, 388
 ODR, one definition rule, 38
 odwrotna notacja polska, RPN, 79
 ograniczanie wartości wektora, 182
 określanie typu
 automatyczne, 31
 niejawne, 33
 opakowanie
 iteratora, 108
 kontenera, 109
 wyrażenia lambda, 138
 opcje debugowania, 120
 operand, 81
 operator
 [], 59
 |, 434
 <, 95
 >>, 83
 dereferencji, 101
 inkrementacji, 101
 porównania niestandardowy, 88
 operatory rzutowania, 149

P

parametr wariadyczny, 43
 permutacja sekwencji danych, 193
 pętla for, 210
 pliki
 .cc, 38
 .o, 38
 dane statystyczne, 440
 wyświetlanie, 426
 zliczanie słów, 258
 wypowiedzi, 206
 wstawiania, 66
 podział
 ciągu tekstowego, 228
 danych, 227
 polecenie for, 103, 116
 polimorfizm, 138
 polityka
 launch::async, 394
 launch::deferred, 394
 launch::async, 394
 launch::deferred, 394
 parallel_policy, 367
 parallel_unsequenced_policy, 367
 sequenced_policy, 367
 polityki
 uruchamiania funkcji, 394
 wykonywania, 367
 predykat, 144
 filtrowanie, 154
 producenty, 401
 programowanie
 funkcyjne, 133
 równoległe, 361, 363, 366, 410
 programy
 czasowe uśpienie, 369
 wielowątkowe, 384
 próbkowanie ogromnego wektora, 189
 przechowywanie różnych typów, 323
 przechwytywanie
 błędów, 293
 ciągu tekstowego, 132
 przez referencję, 136
 przez wartość, 136
 przeciążanie funkcji, 266
 przekierowanie
 do pliku, 279
 sekcji kodu, 277
 strumienia, 278

przekształcenie
 ciągów tekstowych, 245
 Fouriera, 211, 217
 jednostki czasu, 303
 przesunięcie zadania, 390
 Python, 124

R

RAII, resource acquisition is initialization, 280
 reguła
 ODR, 38
 pojedynczej definicji, 38
 SFINAE, 36
 rozkład
 bernoulli_distribution, 360
 Bernoulliego, 357
 discrete_distribution, 360
 liczb losowych, 354
 normal_distribution, 356, 360
 piecwise_constant_distribution, 356
 piecwise_linear_distribution, 357
 uniform_int_distribution, 356, 360
 rozpakowanie, 22
 rozwijanie funkcji, 135
 równoległe generowanie, 406
 RPN, reverse Polish notation, 79
 rzutowanie, 149

S

SFINAE, 36
 silnik
 liczb losowych, 348
 std::default_random_engine, 353
 skierowany graf cykliczny, 410
 składnia inicjalizatora, 29
 słaby wskaźnik, 338, 341
 słownik
 łączenie, 195
 słowo kluczowe
 const, 71
 inline, 37, 39
 sortowanie
 egzemplarzy, 60
 kontenera, 167
 spacja, 249
 sprawdzenie parametrów w zakresie, 45
 sprytny wskaźnik, 342, 347

sterta, 333
 STL, standard template library, 15
 stos
 kalkulator RPN, 79
 operand, 83
 usunięcie elementu, 83
 wstawianie elementu, 82
 strukturalne wiązanie, 22
 struktury danych, 161
 strumień wejścia – wyjścia, 257, 260
 suma wektorów, 218
 sygnalizowanie awarii, 307
 synchronizacja jednoczesnego
 użycia algorytmu, 384
 system plików, 419

Ś

ścieżka dostępu
 kanoniczna, 423
 względna, 423
 znormalizowana, 420

T

tabela wartości hash, 50
 tablica, 48
 tabulator, 249
 token, 80, 83
 tokenizacja
 ciągów tekstowych, 89
 danych wejściowych, 255, 287
 transformacja, 154
 tryb debugowania, 119
 tworzenie
 adaptera iteratora, 123
 ciągów tekstowych, 245
 iteratorów, 104
 klas ciągu tekstowego, 281
 obiektów, 280
 predykatów, 144
 typ danych
 std::any, 320, 323, 327
 std::optional, 310
 std::variant, 323, 327
 typy
 niestandardowe, 73
 plików, 440

U

unia, 324
 uruchamianie wątków, 371
 usuwanie

- białych znaków, 248
- elementów z kontenera, 171
- elementu wektora, 54
- nadmiarowych białych znaków, 235
- obiektów, 280

 uśpienie programu, 369

W

wartość hash, 73
 wartownik iteratora, 116
 wątki

- oczekiwanie, 372
- uruchamianie, 371–374
- zakleszczenie, 381
- zatrzymywanie, 371–374

 wektor, 45, 48, 50

- bezpieczny dostęp, 58
- liczb typu double, 219
- liczb typu int, 219
- nieuporządkowany, 176
- ograniczanie wartości, 182
- próbkowanie, 189
- przekształcenia Fouriera, 213
- sortowanie elementów, 60
- sumowanie, 218
- uporządkowany, 176
- usuwanie bez przesuwania, 54
- usuwanie elementu, 50
- wstawianie elementów, 45
- wymazanie elementu, 52
- wyszukiwanie elementów, 176
- zmiennych licznika, 349

 wektoryzacja, 368
 włączenie bibliotek, 37
 wskaźnik

- null, 339
- shared_ptr, 329, 333
- void*, 320

 wskaźniki

- blokowanie, 340
- do współdzielonych obiektów, 338
- słabe, 338, 341

- sprytne, 342, 347
- współdzielone, 341

 współbieżność, 217, 361
 współdzielenie wartości składowych obiektu, 345
 wstawianie parametrów wariadycznych, 44
 wyjątek, 293, 296, *Patrz także* obsługa błędów

- out of range, 84

 wykres

- fali, 220
- sygnałów, 216

 wypełnianie kontenera, 269
 wyrażenia regularne, 287, 289
 wyrażenie

- constexpr-if, 33, 35
- fold, 40

 wyrażenie lambda, 88, 131

- implementowanie funkcji transform_if(), 150
- opakowanie obiektem std function, 138
- wywoływanie wielu funkcji, 146

 wyszukiwanie, 206

- ciągu tekstowego, 186
- danych, 431
- długich zdań, 88
- elementów, 176
- elementów mapy, 66

 wyświetlanie

- danych za, 273
- plików, 426

 wywoływanie wielu funkcji, 146
 wzajemne wykluczenie, 378
 względna ścieżka dostępu, 423
 wzorzec producent – konsument, 395, 400
 wzór

- na zbiór Mandelbrota, 222
- przekształcenia Fouriera, 211

Z

zadania wykonywane w tle, 390
 zakleszczenie, 381
 zamortyzowana złożoność, 69
 zasięg zmiennej, 26
 zatrzymywanie wątków, 371
 zbieranie danych, 231
 zbiór

- filtrowanie duplikatów, 76
- Mandelbrota, 221, 406
- wstawianie elementów, 44
- wyszukiwanie długich zdań, 88

zegar

 high_resolution_clock, 302

 steady_clock, 302

 system_clock, 302

zliczanie słów, 258

złączenie, 374

zmienianie nazwy plików, 434

zmienna globalna errno, 295

zmiennie

 inicjalizacja, 29

 ograniczanie zasięgu, 26

znak nowego wiersza, 249

Ż

źródło błędu, 295

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



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

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

C++17 i STL. Elegancka klasyka i potężne możliwości.

C++ pozwala zarówno na tworzenie interfejsów wysokiego poziomu, jak i na pisanie kodu działającego na niskim poziomie. Sprawdza się, gdy trzeba zapewnić wysoką wydajność i niskie obciążenie. Język ten jest konsekwentnie rozwijany: jego kolejne specyfikacje, C++14 i C++17, przyniosły wiele znakomitych udoskonaleń. Aby w pełni wykorzystać ten potencjał, należy używać C++ łącznie z biblioteką STL. Jest to standardowa biblioteka języka, dzięki której C++ jest idealny do implementowania oprogramowania o wysokiej jakości i dużej wydajności. Zalety C++ sprawiają, że jest wykorzystywany niemal w każdej dziedzinie. Niestety, wielu programistów nie używa STL.

Dzięki tej książce poznasz użyteczność biblioteki standardowej (STL) w C++17 w praktyce, co pozwoli Ci na tworzenie efektywniejszego i w pełni przenośnego kodu źródłowego. Najpierw poznasz nowe funkcje języka, co pozwoli Ci na zrozumienie reguł rządzących C++, oraz funkcje biblioteki standardowej i sposób jej działania. Podczas pracy nad praktycznymi i łatwymi do wykorzystania recepturami poznasz podstawowe koncepcje STL, takie jak kontener, algorytm, klasa narzędziowa, wyrażenie lambda, iterator i wiele innych. Dowiesz się, jak działają najnowsze funkcje wprowadzone w standardzie C++17. Dzięki temu zaoszczędzisz czas i wysiłek podczas programowania, a Twój kod stanie się prostszy i zdecydowanie bardziej elegancki.

W książce między innymi:

- nowości w standardzie C++17
- kontenery STL i koncepcja iteratorów
- wyrażenia lambda i zaawansowane algorytmy biblioteki STL
- ciągi tekstowe, strumień wejścia-wyjścia i wyrażenia regularne
- programowanie równoległe i współbieżność

Jacek Galowicz do niedawna implementował sterowniki jądra w C i C++. Tworzył też grafiki 3D i bazy danych, a także zajmował się komunikacją sieciową. Aktualnie pracuje w firmach Intel oraz FireEye, w których programuje systemy operacyjne o dużej wydajności i wysokim poziomie bezpieczeństwa, wykorzystując wirtualizację Intel x86. Jego ogromną pasją są nowoczesne implementacje C++ na niskim poziomie i łączenie w kodzie wysokiej wydajności z eleganckim stylem.

 helion.pl	<i>Sprawdź nasze szkolenia</i> SZKOLENIA  AKADEMIA IT & BUSINESS WWW.SZKOLENIA.HELION.PL	KOD KORZYŚCI Ślepnij po więcej! ▶  ISBN 978-83-283-4501-0  9 788328 345010
 helion.pl		
 HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl		
INFORMATYKA W NAJLEPSZYM WYDANIU		Cena: 79,00 zł

Packt