



C++

Podróż po języku dla zaawansowanych

Wydanie III

Bjarne Stroustrup



Tytuł oryginału: A Tour of C++ (C++ In-Depth Series), 3rd Edition

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-8322-525-8

Authorized translation from the English language edition, entitled A Tour of C++, 3rd Edition by Bjarne Stroustrup, published by Pearson Education, Inc, publishing as Addison-Wesley Professional, Copyright © 2023 Pearson Education, Inc.

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 Pearson Education, Inc.

Polish language edition published by Helion S.A., Copyright © 2023.

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 wydawca 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 wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<https://helion.pl/user/opinie/cppoz3>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 230 98 63

e-mail: helion@helion.pl

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

Printed in Poland.

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

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

Spis treści

Wstęp	9
1. Podstawy	13
1.1 Wprowadzenie	13
1.2 Programy	14
1.3 Funkcje	16
1.4 Typy, zmienne i arytmetyka	18
1.5 Zakres i cykl istnienia	21
1.6 Stałe	22
1.7 Wskaźniki, tablice i referencje	24
1.8 Testy	27
1.9 Mapowanie sprzętowe	29
1.10 Porady	31
2. Typy zdefiniowane przez użytkownika	33
2.1 Wprowadzenie	33
2.2 Struktury	34
2.3 Klasy	35
2.4 Wyliczenia	37
2.5 Unie	39
2.6 Porady	40
3. Moduły	42
3.1 Wprowadzenie	42
3.2 Kompilacja rozdzielna	43
3.3 Przestrzenie nazw	49
3.4 Argumenty i wartości zwrótne funkcji	50
3.5 Porady	56

4 Spis treści

4. Obsługa błędów	57
4.1 Wprowadzenie	57
4.2 Wyjątki	58
4.3 Niezmienniki	59
4.4 Różne sposoby obsługi błędów	61
4.5 Asercje	63
4.6 Porady	66
5. Klasy	67
5.1 Wprowadzenie	67
5.2 Typy konkretne	68
5.3 Typy abstrakcyjne	74
5.4 Funkcje wirtualne	77
5.5 Hierarchie klas	78
5.6 Porady	84
6. Operacje podstawowe	86
6.1 Wprowadzenie	86
6.2 Kopiowanie i przenoszenie	89
6.3 Zarządzanie zasobami	94
6.4 Przeciążanie operatorów	95
6.5 Operacje standardowe	97
6.6 Literały zdefiniowane przez użytkownika	100
6.7 Porady	102
7. Szablony	103
7.1 Wprowadzenie	103
7.2 Typy parametryzowane	104
7.3 Operacje parametryzowane	109
7.4 Mechanizmy szablonów	115
7.5 Porady	118
8. Koncepcje i programowanie generyczne	119
8.1 Wprowadzenie	119
8.2 Koncepcje	120
8.3 Programowanie generyczne	129
8.4 Szablony zmienne	131
8.5 Model kompilacji szablonów	134
8.6 Porady	135
9. Podstawowe informacje o bibliotece	136
9.1 Wprowadzenie	136
9.2 Komponenty biblioteki standardowej	137

9.3	Organizacja biblioteki standardowej	138
9.4	Porady	142
10.	Łańcuchy i wyrażenia regularne	143
10.1	Wprowadzenie	143
10.2	Łańcuchy	144
10.3	Widoki łańcuchów	146
10.4	Wyrażenia regularne	148
10.5	Porady	155
11.	Wejście i wyjście	157
11.1	Wprowadzenie	157
11.2	Wyjście	158
11.3	Wejście	159
11.4	Stan wejścia i wyjścia	161
11.5	Wejście i wyjście typów zdefiniowanych przez użytkownika	162
11.6	Formatowanie wyjścia	163
11.7	Strumienie	167
11.8	Wejście i wyjście w stylu języka C	170
11.9	System plików	171
11.10	Porady	175
12.	Kontenery	177
12.1	Wprowadzenie	177
12.2	Typ vector	178
12.3	Listy	183
12.4	forward_list	184
12.5	Słowniki	184
12.6	Słowniki nieuporządkowane	185
12.7	Alokatory	187
12.8	Przegląd kontenerów	189
12.9	Porady	191
13.	Algorytmy	193
13.1	Wprowadzenie	193
13.2	Zastosowania iteratorów	195
13.3	Typy iteratorów	198
13.4	Predykaty	201
13.5	Przegląd algorytmów	202
13.6	Algorytmy równoległe	203
13.7	Porady	204

14. Zakresy	205
14.1 Wprowadzenie	205
14.2 Widoki	206
14.3 Generatory	208
14.4 Potoki	209
14.5 Koncepcje — informacje ogólne	210
14.6 Porady	215
15. Wskaźniki i kontenery	216
15.1 Wprowadzenie	216
15.2 Wskaźniki	217
15.3 Kontenery	223
15.4 Alternatywy	230
15.5 Porady	234
16. Narzędzia pomocnicze	235
16.1 Wprowadzenie	235
16.2 Czas	236
16.3 Adaptacja funkcji	238
16.4 Funkcje typów	240
16.5 source_location	245
16.6 move() i forward()	246
16.7 Manipulowanie bitami	247
16.8 Zamykanie programu	248
16.9 Porady	249
17. Liczby	250
17.1 Wprowadzenie	250
17.2 Funkcje matematyczne	251
17.3 Algorytmy numeryczne	252
17.4 Liczby zespolone	254
17.5 Liczby losowe	254
17.6 Arytmetyka wektorowa	256
17.7 Granice numeryczne	257
17.8 Aliasy typów	257
17.9 Stałe matematyczne	258
17.10 Porady	258
18. Współbieżność	260
18.1 Wprowadzenie	260
18.2 Zadania i wątki	261
18.3 Wspólne używanie danych	264
18.4 Oczekiwanie na zdarzenia	267

18.5	Komunikacja między zadaniami	268
18.6	Współprocedury	273
18.7	Porady	277
19.	Historia i zgodność	279
19.1	Historia	279
19.2	Ewolucja funkcjonalności C++	288
19.3	Zgodność C i C++	293
19.4	Bibliografia	298
19.5	Porady	301
A.	Moduł std	303
A.1	Wprowadzenie	303
A.2	Używaj tego, co masz w implementacji	304
A.3	Używaj nagłówków	304
A.4	Stwórz własny moduł std	305
A.5	Porady	305
S.	Skorowidz	306

Narzędzia pomocnicze

*Czas, którego marnowanie sprawia ci przyjemność,
nie jest czasem zmarnowanym.*
— Bertrand Russell

- Wprowadzenie
- Czas
 - Zegary. Kalendarze. Strefy czasowe
- Adaptacja funkcji
 - Lambdy jako adaptery. `mem_fn()`. `function`
- Funkcje typów
 - Predykaty typów. Własności warunkowe. Generatory typów. Typy powiązane
- `source_location()`
- `move()` i `forward()`
- Manipulacja bitami
- Zamykanie programu
- Porady

16.1 Wprowadzenie

Przypisanie komponentu biblioteki do grupy „Narzędzia” niewiele mówi użytkownikom. Oczywiście każdy komponent biblioteki jest w jakimś sensie narzędziem. W tym rozdziale opisuję elementy biblioteki, które mają kluczowe znaczenie w wielu sytuacjach, ale nie pasowały nigdzie indziej. Wiele z nich stanowi element budowy bardziej zaawansowanych narzędzi biblioteki, w tym innych komponentów.

16.2 Czas

Wszystko, co jest potrzebne do pracy z czasem, znajduje się w nagłówku `<chrono>` biblioteki standardowej:

- Zegary, `time_point` i `duration` do pomiaru czasu trwania różnych czynności oraz jako podstawa budowy wszystkiego, co wiąże się z czasem.
- `day`, `month`, `year` oraz `weekdays` do przedstawiania punktów w czasie `time_point` w kontekście życia codziennego.
- `time_zone` i `zoned_time` do pracy z różnicami w czasie w różnych częściach globu.

Zasadniczo każdy większy system wykorzystuje niektóre z tych narzędzi.

16.2.1 Zegary

Oto prosty przykład mierzenia czasu wykonywania jakichś czynności:

```
using namespace std::chrono; // w podrzędnej przestrzeni nazw std::chrono (zobacz 3.3)

auto t0 = system_clock::now();
do_work();
auto t1 = system_clock::now();

cout << t1-t0 << "\n"; // domyślna jednostka: 20223[1/00000000]s
cout << duration_cast<milliseconds>(t1-t0).count() << " ms\n"; // określona jednostka: 2 ms
cout << duration_cast<nanoseconds>(t1-t0).count() << " ns\n"; // określona jednostka: 2022300ns
```

Zegar zwraca `time_point` (punkt w czasie). Wynikiem odejmowania dwóch obiektów `time_point` jest czas trwania (okres). Domyślny operator `<<` typu `duration` dodaje jednostkę w formie przyrostka. Różne zegary zwracają wyniki w różnych jednostkach czasu, „tyknięciach zegara” (użyty przeze mnie mierzy czas w setkach nanosekund), dlatego często powinno się przekonwertować otrzymaną wartość na odpowiednią jednostkę. Do tego służy `duration_cast`.

Zegary umożliwiają wykonywanie szybkich pomiarów. Nigdy nie wypowiadaj się na temat „wydajności” kodu, dopóki nie przeprowadzisz pomiarów. Bardzo rzadko udaje się trafnie zgadnąć w tej kwestii. Szybkie i proste pomiary są lepsze niż ich brak, ale wydajność nowoczesnych komputerów to skomplikowany temat i dlatego nie należy zbyt mocno przywiązywać do wyników paru prostych testów. Testy zawsze wykonuj wielokrotnie, aby zmniejszyć ryzyko odchylenia wyniku przez jakieś rzadkie zdarzenie lub efekt działania pamięci podręcznej.

Przestrzeń nazw `std::chrono_literals` zawiera definicje przyrostków jednostek czasu (6.6). Na przykład:

```
this_thread::sleep_for(10ms+33us); // czekaj 10 milisekund i 33 mikrosekundy
```

Konwencjonalne nazwy symboliczne wyraźnie zwiększają czytelność i ułatwiają utrzymanie kodu.

16.2.2 Kalendarze

Podczas pracy z codziennymi wydarzeniami rzadko wykorzystujemy milisekundy. Najczęściej korzystamy z jednostek obejmujących lata, miesiące, dni, godziny, minuty, sekundy oraz dni tygodnia. Biblioteka standardowa obsługuje je wszystkie. Na przykład:

```
auto spring_day = April/7/2018;
cout << weekday(spring_day) << '\n';           // Sat
cout << format("{:%A}\n", weekday(spring_day)); // Saturday
```

W moim komputerze Sat jest domyślną reprezentacją znakową soboty. Nie podobał mi się ten skrót, więc użyłem formatu (11.6.2), aby uzyskać pełną nazwę dnia. Z niejasnych powodów %A oznacza „wydrukuj pełną nazwę dnia tygodnia”. Oczywiście April to miesiąc (kwiecień), a dokładniej mówiąc std::chrono::Month. Możemy też napisać:

```
auto spring_day = 2018y/April/7;
```

Przyrostek y odróżnia lata od zwykłych liczb całkowitych, które są używane w odniesieniu do dni miesiąca ponumerowanych od 1 do 31.

Istnieje możliwość zdefiniowania nieprawidłowej daty. W razie wątpliwości można użyć funkcji ok() do sprawdzenia:

```
auto bad_day = January/0/2024;
if (!bad_day.ok())
    cout << bad_day << " is not a valid day\n";
```

Oczywiście funkcja ok() jest najbardziej przydatna do sprawdzania dat uzyskanych w wyniku obliczeń.

Daty są składane dzięki przecięciu operatora / (ukośnik) przez typy year, month i int. Powstały w wyniku typ Year_month_day ma operacje konwersji na time_point i w drugą stronę, co umożliwia szybkie i precyzyjne wykonywanie operacji na datach. Na przykład:

```
sys_days t = sys_days{February/25/2022}; // punkt w czasie z precyzją dni
t += days(7);                             // tydzień po 25 lutego 2022 r.
auto d = year_month_day(t);                 // konwersja punktu w czasie z powrotem na kalendarz

cout << d << '\n';
// 2022-03-04
cout << format("{:%B}/{} / {} \n", d.month(), d.day(), d.year()); // Marzec/04/2022
```

Te obliczenia wymagają zmiany miesiąca i wiedzy na temat lat przestępnych. Domyślnie implementacja podaje datę w standardowym formacie ISO 8601. Aby otrzymać nazwę miesiąca w postaci słowa „Marzec”, musimy rozbić datę na poszczególne pola i zastosować detale formatujące (11.6.2). Z niejasnych powodów %B oznacza „wydrukuj pełną nazwę miesiąca”.

Takie operacje często można wykonywać w czasie kompilacji, dzięki czemu są one zaskakująco szybkie:

```
static_assert(weekday(April/7/2018) == Saturday); // true
```

Kalendarze są skomplikowane i subtelne. To typowe i odpowiednie dla „systemów” przeznaczonych dla „zwykłych ludzi” na przestrzeni wieków, a nie dla programistów, którzy chcieliby uprościć

swoją pracę. System kalendarzy biblioteki standardowej może być (i jest) rozszerzany o inne kalendarze, takie jak juliański, islamski, tajski itd.

16.2.3 Strefy czasowe

Jednym z największych wyzwań w pracy z czasem jest poprawne posługiwanie się strefami czasowymi. Trudno je zapamiętać, ponieważ są arbitralne oraz zmieniają się na wiele różnych sposobów, które nie są ustandaryzowane w ujęciu całej planety. Na przykład:

```
auto tp = system_clock::now(); // tp to time_point
cout << tp << '\n';           // 2021-11-27 21:36:08.2085095

zoned_time ztp { current_zone(), tp }; // 2021-11-27 16:36:08.2085095 EST
cout << ztp << '\n';

const time_zone est {"Europe/Copenhagen"};
cout << zoned_time{ &est, tp } << '\n'; // 2021-11-27 22:36:08.2085095 GMT+1
```

Obiekt typu `time_zone` reprezentuje czas względny w odniesieniu do standardu (zwanego GMT lub UTC) używanego przez `system_clock`. Biblioteka standardowa przeprowadza synchronizację z globalną bazą danych (IANA), aby zwracać poprawne wyniki. Ta synchronizacja może odbywać się automatycznie w systemie operacyjnym lub pod kontrolą administratora systemu. Nazwy stref czasowych są łańcuchami w stylu języka C w formie „kontynent/duże miasto”, np. "America/New_York", "Asia/Tokyo", "Africa/Nairobi". Obiekt typu `zoned_time` to `time_zone` wraz z `time_point`.

Strefy czasowe, tak jak kalendarze, odnoszą się do kwestii, które powinniśmy pozostawić bibliotece standardowej zamiast stosować własnoręcznie przygotowane rozwiązania. Na przykład: o której godzinie ostatniego dnia lutego 2024 r. w Nowym Jorku zmieni się data w New Delhi? Kiedy skończył się czas letni w Denver w amerykańskim stanie Kolorado w 2020 r.? Kiedy będzie następny rok przestępny? Biblioteka standardowa to wszystko „wie”.

16.3 Adaptacja funkcji

Gdy funkcja jest przekazywana jako argument funkcji, typ tego argumentu musi dokładnie odpowiadać wymaganiom określonym w deklaracji funkcji wywoływanej. Jeśli dany argument tylko „prawie spełnia wymagania”, programista ma do wyboru różne rozwiązania:

- użyć lambdy (16.3.1);
- za pomocą funkcji `std::mem_fn()` utworzyć obiekt funkcyjny z funkcji składowej (16.3.2);
- zdefiniować funkcję tak, aby przyjmowała `std::function` (16.3.3).

Jest jeszcze wiele innych sposobów, ale zazwyczaj jedna z powyższych metod okazuje się najlepsza.

16.3.1 Lambdy jako adaptery

Spójrz na klasyczny przykład rysowania wszystkich kształtów:

```
void draw_all(vector<Shape*>& v)
{
    for_each(v.begin(), v.end(), [](Shape* p) { p->draw(); });
}
```

Tak jak wszystkie algorytmy z biblioteki standardowej, `for_each()` wywołuje swój argument przy użyciu tradycyjnej składni wywołania funkcji $f(x)$, ale funkcja `draw()` klasy `Shape` stosuje tradycyjną notację obiektową $x \rightarrow f()$. Lambda bez problemu przełącza się między tymi dwiema notacjami.

16.3.2 `mem_fn()`

Dla danej funkcji składowej adapter funkcji `mem_fn(mf)` tworzy obiekt funkcyjny, który można wywoływać jako funkcję niebędącą składową klasy. Na przykład:

```
void draw_all(vector<Shape*>& v)
{
    for_each(v.begin(), v.end(), mem_fn(&Shape::draw));
}
```

Zanim w C++11 wprowadzono lambdy, funkcja `mem_fn()` i jej odpowiedniki stanowiły podstawowy mechanizm zamiany obiektowej metody wywoływania na funkcyjną.

16.3.3 `function`

Typ `function` z biblioteki standardowej może przechowywać dowolny obiekt, który można wywołać za pomocą operatora wywołania `()`. Innymi słowy, obiekt typu `function` jest obiektem funkcyjnym (6.3.2). Na przykład:

```
int f1(double);
function<int(double)> fct1 {f1}; // inicjalizacja do f1

int f2(string);
function fct2 {f2}; // typ fct2 to function<int(string)>

function fct3 = [](Shape* p) { p->draw(); }; // typ fct3 to function<void(Shape*)>
```

W przypadku `fct2` typ funkcji pozostawiłem do dedukcji z inicjalizatora — `int(string)`.

Oczywiście obiekty typu `function` są przydatne przy tworzeniu wywołań zwrotnych, przekazywaniu operacji w argumentach, przekazywaniu obiektów funkcyjnych itd. To jednak w porównaniu z bezpośrednimi wywołaniami może powodować pewien narzut. W szczególności, w przypadku obiektu `function`, którego rozmiar nie jest obliczany w czasie kompilacji, alokacja w pamięci wolnej może fatalnie wpłynąć na działanie aplikacji, w której krytyczne znaczenie ma wydajność. Rozwiązanie pojawi się w C++23 w postaci `move_only_function`.

Innym problemem jest to, że funkcja, będąc obiektem, nie uczestniczy w przeciążaniu. Jeśli chcesz przeciążać obiekty funkcyjne (także lambdy), możesz użyć typu `overloaded` (15.4.1).

16.4 Funkcje typów

Funkcja typu to taka funkcja, której wartość jest określana w czasie kompilacji, przyjmująca typ jako argument lub zwracająca typ. Biblioteka standardowa zawiera wiele funkcji typu, dzięki którym implementatorzy bibliotek (a także inni programiści) mogą pisać kod wykorzystujący aspekty języka, biblioteki standardowej i ogólnie kodu.

W odniesieniu do typów numerycznych szerokie spektrum informacji dostarcza `numeric_limits` z nagłówka `<limits>` (17.7). Na przykład:

```
constexpr float min = numeric_limits<float>::min(); // najmniejsza dodatnia liczba
// zmiennoprzecinkowa
```

Natomiast rozmiar obiektu można sprawdzić za pomocą wbudowanego operatora `sizeof` (1.4). Na przykład:

```
constexpr int sz_i = sizeof(int); // liczba bajtów w int
```

W nagłówku `<type_traits>` biblioteki standardowej znajduje się wiele funkcji do sprawdzania właściwości typów. Na przykład:

```
bool b = is_arithmetic_v<X>; // prawda, jeśli X jest jednym z wbudowanych typów arytmetycznych
using Res = invoke_result_t<decltype(f)>; // Res to int, jeśli f jest funkcją zwracającą int
```

Konstrukcja `decltype(f)` to wywołanie wbudowanej funkcji typu `decltype()`, która zwraca zadeklarowany typ swojego argumentu — tutaj `f`.

Niektóre funkcje typu tworzą nowe typy na podstawie danych wejściowych. Na przykład:

```
typename<typename T>
using Store = conditional_t<sizeof(T)<max, 0n_stack<T>, 0n_heap<T>>;
```

Jeśli pierwszym (logicznym) argumentem funkcji `conditional_t` jest `true`, to wynikiem jest pierwsza z alternatywnych opcji. W przeciwnym razie — druga. Zakładając, że `0n_stack` i `0n_heap` udostępniają te same funkcje dostępu do `T`, mogą alokować swoje `T` zgodnie z tym, co sugerują ich nazwy. Dzięki temu użytkowników `Store<X>` można dostosować odpowiednio do rozmiaru `X` obiektów. Korzyści w zakresie wydajności wynikające z tego wyboru alokacji mogą być znaczne. To jest prosty przykład tego, jak możemy tworzyć nowe funkcje typu na podstawie standardowych lub przy użyciu koncepcji.

Koncepcje są funkcjami typu. Używane w wyrażeniach są predykatami typów. Na przykład:

```
template<typename F, typename... Args>
auto call(F f, Args... a, Allocator alloc)
{
    if constexpr (invocable<F, alloc, Args...>) // potrzebny alokator?
        return f(f, alloc, a...);
    else
        return f(f, a...);
}
```

W wielu przypadkach koncepcje są najlepszymi funkcjami typu, ale większość biblioteki standardowej została napisana przed wprowadzeniem koncepcji i musi uwzględniać kod, w którym nie są one używane.

Konwencje notacyjne są mało przejrzyste. W bibliotece standardowej funkcje typu zwracające wartości są oznaczone przyrostkiem `_v`, a funkcje typu zwracające typy są oznaczone przyrostkiem `_t`. Jest to pozostałość po słabym typizowaniu języka C i czasach sprzed wprowadzenia koncepcji w C++. Żadna funkcja typu biblioteki standardowej nie zwraca i typu, i wartości, więc te przyrostki są zbędne. Dzięki koncepcjom, zarówno w bibliotece standardowej, jak i gdziekolwiek indziej, żaden przyrostek nie jest potrzebny ani używany.

Funkcje typu są częścią mechanizmu języka C++ do wykonywania obliczeń w czasie kompilacji umożliwiającego ściślejszą kontrolę typów i pozwalającego uzyskać większą wydajność niż ta, którą można byłoby osiągnąć bez niego. Posługiwanie się funkcjami typu i koncepcjami (rozdział 8., podrozdział 14.5) często nazywa się **metaprogramowaniem** lub (kiedy wykorzystywane są szablony) **metaprogramowaniem szablonowym**.

16.4.1 Predykaty typów

W nagłówku `<type_traits>` biblioteki standardowej znajduje się wiele prostych funkcji zwanych **predykatami typów** (ang. *type predicate*), które dostarczają podstawowe informacje o typach. W poniższej tabeli znajduje się opis kilku z nich.

Jednym z tradycyjnych zastosowań tych predykatów jest ograniczanie argumentów szablonów. Na przykład:

```
template<typename Scalar>
class complex {
    Scalar re, im;
public:
    static_assert(is_arithmetic_v<Scalar>, "Niestety obsługuję tylko liczby zespolone.");
    // ...
};
```

To jednak — podobnie jak inne tradycyjne zastosowania — można wyrazić prościej i bardziej elegancko za pomocą koncepcji:

```
template<Arithmetic Scalar>
class complex {
    Scalar re, im;
public:
    // ...
};
```

W wielu przypadkach predykaty typów takie jak `is_arithmetic` dla ułatwienia zostają zaszyte w definicjach koncepcji. Na przykład:

```
template<typename T>
concept Arithmetic = is_arithmetic_v<T>;
```

Co ciekawe nie istnieje koncepcja `std::arithmetic`.

Często możemy definiować koncepcje, które są bardziej ogólne niż predykaty typów z biblioteki standardowej. Wiele z nich odnosi się tylko do typów wbudowanych. Programista może zdefiniować koncepcję w odniesieniu do wymaganych operacji, co sugeruje definicja koncepcji `Number` (8.2.4):

Wybrane predykaty typów T, A i U są typami; wszystkie predykaty zwracają wartość logiczną	
<code>is_void_v<T></code>	Czy T jest void?
<code>is_integral_v<T></code>	Czy T jest typu całkowitoliczbowego?
<code>is_floating_point_v<T></code>	Czy T jest typu zmiennoprzecinkowego?
<code>is_class_v<T></code>	Czy T jest klasą (a nie unią)?
<code>is_function_v<T></code>	Czy T jest funkcją (a nie obiektem funkcyjnym lub wskaźnikiem do funkcji)?
<code>is_arithmetic_v<T></code>	Czy T jest typem całkowitoliczbowym lub zmiennoprzecinkowym?
<code>is_scalar_v<T></code>	Czy T jest typem arytmetycznym, wyliczeniem, wskaźnikiem lub wskaźnikiem do składowej?
<code>is_constructible_v<T, A...></code>	Czy T można utworzyć z listy argumentów A...?
<code>is_default_constructible_v<T></code>	Czy T można utworzyć bez jawnych argumentów?
<code>is_copy_constructible_v<T></code>	Czy T można utworzyć z innego T?
<code>is_move_constructible_v<T></code>	Czy T można przenieść lub skopiować do innego T?
<code>is_assignable_v<T,U></code>	Czy U można przypisać do T?
<code>is_trivially_copyable_v<T,U></code>	Czy U można przypisać do T bez operacji kopiowania zdefiniowanych przez użytkownika?
<code>is_same_v<T,U></code>	Czy T jest tym samym typem co U?
<code>is_base_of_v<T,U></code>	Czy U pochodzi od T lub czy U jest tym samym typem co U?
<code>is_convertible_v<T,U></code>	Czy T można niejawnie przekonwertować na U?
<code>is_iterator_v<T></code>	Czy T jest typem iteratora?
<code>is_invocable_v<T, A...></code>	Czy T można wywołać z listą argumentów A...?
<code>has_virtual_destructor_v<T></code>	Czy T ma destruktor wirtualny?

```
template<typename T, typename U = T>
concept Arithmetic = Number<T,U> && Number<U,T>;
```

Predykaty typów biblioteki standardowej najczęściej można spotkać głęboko w implementacji podstawowych usług, gdzie często wskazują przypadki optymalizacji. Na przykład część implementacji funkcji `std::copy(Iter, Iter, Iter2)` mogłaby optymalizować ważny przypadek związany z ciągłymi sekwencjami obiektów typów prostych, takich jak liczby całkowite:

```
template<class T>
void cpy1(T* first, T* last, T* target)
{
    if constexpr (is_trivially_copyable_v<T>)
        memcpy(first, target, (last - first) * sizeof(T));
    else
        while (first != last) *target++ = *first++;
}
```


Ta prosta optymalizacja bije swój nieoptymalizowany wariant o około 50% w niektórych implementacjach. Nie angażuj się w takie sprawy, dopóki nie sprawdzisz, czy standard nie robi czegoś lepiej. Ręcznie optymalizowany kod jest zazwyczaj trudniejszy w utrzymaniu od prostszych alternatyw.

16.4.2 Właściwości warunkowe

Spójrz na poniższą definicję „inteligentnego wskaźnika”:

```
template<typename T>
class Smart_pointer {
    T& operator*() const;
    T* operator->() const; // -> powinien działać tylko wtedy, gdy T jest klasą
};
```

Operator `->` powinien być zdefiniowany tylko wtedy, gdy `T` jest typem klasowym. Na przykład `Smart_pointer<vector<T>>` powinien mieć `->`, a `Smart_pointer<int>` — nie. Nie możemy użyć instrukcji `if` kompilacji, ponieważ nie jesteśmy wewnątrz funkcji. Dlatego piszemy:

```
template<typename T>
class Smart_pointer {
    //...
    T& operator*() const;
    T* operator->() const requires is_classv<T>; // -> zdefiniowany tylko wtedy, gdy T jest klasą
};
```

Predykat typu bezpośrednio wyraża ograniczenie dotyczące funkcji `operator->()`. W tym przypadku można by było użyć także koncepcji. W bibliotece standardowej nie ma koncepcji reprezentującej wymóg, że typ musi być klasowy (tzn. musi być klasą, strukturą lub unią), ale możemy taką zdefiniować:

```
template<typename T>
concept Class = is_class v<T> || is_union v<T>; // unie są klasami

template<typename T>
class Smart_pointer {
    //...
    T& operator*() const;
    T* operator->() const requires Class<T>; // -> jest zdefiniowany tylko, jeśli T jest klasą lub unią
};
```

Koncepcja często jest bardziej ogólna albo po prostu bardziej odpowiednia niż bezpośrednie użycie predykatu typu z biblioteki standardowej.

16.4.3 Generatory typów

Wiele funkcji typu zwraca typy, często nowe, które zostały przez nie obliczone. Takie funkcje nazywam **generatorami typów**, aby odróżnić je od predykatów typów. W standardzie jest ich kilka. Poniższa tabela przedstawia kilka przykładowych.

Wybrane generatory typów	
<code>R=remove_const_t<T></code>	R to typ T z usuniętą ewentualną deklaracją <code>const</code>
<code>R=add_const_t<T></code>	R to <code>const T</code>
<code>R=remove_reference_t<T></code>	Jeśli T jest referencją U&, to R odpowiada U, a w przeciwnym razie R odpowiada T
<code>R=add_lvalue_reference_t<T></code>	Jeśli T jest referencją l-wartościową, to R odpowiada T, a w przeciwnym razie R odpowiada T&
<code>R=add_rvalue_reference_t<T></code>	Jeśli T jest referencją r-wartościową, to R odpowiada T, a w przeciwnym razie R odpowiada T&&
<code>R=enable_if_t<b,T =void></code>	Jeśli b jest prawdą, to R odpowiada T, a w przeciwnym razie R jest niezdefiniowane
<code>R=conditional_t<b,T,U></code>	R odpowiada T, jeśli b jest prawdą, a w przeciwnym razie R odpowiada U
<code>R=common_type_t<T...></code>	Jeśli istnieje typ, na który można niejawnie przekonwertować wszystkie T, to R jest tym typem, a w przeciwnym razie R jest niezdefiniowane
<code>R=underlying_type_t<T></code>	Jeśli T jest wyliczeniem, to R jest jego podstawowym typem, a w przeciwnym razie występuje błąd
<code>R=invoke_result_t<T,A...></code>	Jeśli T można wywołać z argumentami A..., to R jest typem zwrrotnym, a w przeciwnym razie występuje błąd

Te funkcje typów są zazwyczaj używane w implementacjach narzędzi, a nie w zwykłym kodzie aplikacji. Z nich wszystkich `enable_if` występuje chyba najczęściej w kodzie napisanym przed wprowadzeniem koncepcji. Na przykład warunkowo włączany operator `->` dla inteligentnego wskaźnika tradycyjnie jest implementowany mniej więcej tak:

```
template<typename T>
class Smart_pointer {
    // ...
    T& operator*();
    enable_if<is_class v<T>,T&> operator->();    // -> jest zdefiniowany tylko, jeśli T jest klasą
};
```

Niezbyt przyjemnie się to czyta, a bardziej skomplikowane przypadki są jeszcze gorsze. Definicja `enable_if` bazuje na subtelnej cesze języka o nazwie **SFINAE** (ang. *Substitution Failure Is Not An Error* — nieudane podstawienie nie jest błędem). Poszukaj informacji na ten temat, jeśli ich potrzebujesz.

16.4.4 Typy powiązane

Wszystkie kontenery biblioteki standardowej (12.8) i wszystkie kontenery zdefiniowane na ich wzór mają pewne **typy powiązane**, takie jak ich typy wartości i iteratorów. W nagłówkach `<iterator>` i `<ranges>` biblioteki standardowej znajdują się następujące nazwy:

Wybrane generatory typów	
<code>range_value_t<R></code>	Typ elementów zakresu R
<code>iter_value_t<T></code>	Typ elementów wskazywanych przez iterator T
<code>iterator_t<R></code>	Typ iteratora zakresu R

16.5 source_location

Drukując na wyjściu zawartość stosu lub komunikat o błędzie, często chcemy dodać informację na temat lokalizacji w kodzie źródłowym. Do tego służy klasa `source_location` z biblioteki:

```
const source_location loc = source_location::current();
```

Funkcja `current()` zwraca obiekt klasy `source_location` wskazujący miejsce w kodzie źródłowym, w którym się znajduje. Klasa `source_location` ma funkcje składowe `file()` i `function_name()`, które zwracają łańcuchy w stylu C, oraz funkcje składowe `line()` i `column()`, które zwracają liczby całkowite bez znaku.

Wywołując je w funkcji, możemy utworzyć całkiem przydatny komunikat dziennika:

```
void log(const string& mess = "", const source_location loc = source_location::current())
{
    cout << loc.file_name()
         << '(' << loc.line() << ':' << loc.column() << ") "
         << loc.function_name() ": "
         << mess;
}
```

Wywołanie funkcji `current()` jest domyślnym argumentem, dzięki czemu otrzymujemy lokalizację wywołującego funkcji `log()`, a nie lokalizację samej funkcji `log()`:

```
void foo()
{
    log("Hello"); // myfile.cpp (17,4) foo: Hello
    // ...
}

int bar(const string& label)
{
    log(label); // myfile.cpp (23,4) bar: <<wartość label>>
    // ...
}
```

W kodzie napisanym przed powstaniem C++20 lub przeznaczonym do kompilacji w starszych kompilatorach w takiej sytuacji używane byłyby makra `__FILE__` i `__LINE__`.

16.6 move() i forward()

Wybór między przenoszeniem i kopiowaniem zwykle odbywa się zakulisowo (3.6). Kompilator wybiera przenoszenie, kiedy dany obiekt ma być zniszczony (jak przy zwracaniu), ponieważ ta operacja jest uważana za prostszą i mniej czasochłonną. Czasami jednak programista musi wyrazić swój zamiar wprost. Na przykład wskaźnik `unique_ptr` jest jedynym właścicielem obiektu, w związku z czym nie może być kopiowany. Jeśli więc jest potrzebny w innym miejscu, należy go przenieść. Na przykład:

```
void f1()
{
    auto p = make_unique<int>(2);
    auto q = p;      // błąd: nie można skopiować wskaźnika unique_ptr
    auto q = move(p); // teraz p zawiera nullptr
    // ...
}
```

Co zaskakujące, funkcja `std::move()` niczego nie przenosi, tylko rzutuje swój argument na referencję r-wartości, zaznaczając w ten sposób, że jej argument nie będzie już używany, a więc może zostać przeniesiony (6.2.2). Powinna zatem nazywać się `rvalue_cast`. Ma ona zastosowanie w kilku ważnych przypadkach. Weźmy na przykład prostą zamianę:

```
template <typename T>
void swap(T& a, T& b)
{
    T tmp {move(a)}; // konstruktor T „widzi” r-wartość i dokonuje przeniesienia
    a = move(b);    // przypisanie T „widzi” r-wartość i dokonuje przeniesienia
    b = move(tmp);  // przypisanie T „widzi” r-wartość i dokonuje przeniesienia
}
```

Nie powinniśmy wielokrotnie przenosić potencjalnie dużych obiektów i dlatego żądamy przeniesienia za pomocą funkcji `std::move()`.

Tak jak w przypadku innych rodzajów rzutowania, funkcja `std::move()` ma pewne kuszące, choć niebezpieczne zastosowania. Na przykład:

```
string s1 = "Witaj,";
string s2 = "świecie";
vector<string> v;
v.push_back(s1);      // argument const string&; push_back() wykona kopię
v.push_back(move(s2)); // użycie konstruktora przenoszącego
v.emplace_back(s1);  // alternatywa; umieszcza kopię s1 na nowej pozycji końcowej v (12.8)
```

W tym przykładzie obiekt `s1` zostaje skopiowany (przez funkcję `push_back()`), podczas gdy `s2` zostaje przeniesiony. Czasami (tylko czasami) może to przyspieszyć działanie funkcji `push_back()` w takiej sytuacji jak z `s2`. Sęk w tym, że obiekt, z którego nastąpiło przeniesienie, jest pozostawiany. Jeśli ponownie użyjemy `s2`, będziemy mieć kłopoty:

```
cout << s1[2]; // drukuje 't'
cout << s2[2]; // awaria?
```

Moim zdaniem to zastosowanie funkcji `std::move()` jest zbyt ryzykowne, aby było powszechne. Nie korzystaj z tej metody, dopóki nie wykażesz, że daje ona wyraźne korzyści pod względem wydajności. Później przez przypadek ktoś może użyć obiektu, z którego nastąpiło przeniesienie.

Kompilator „wie”, że wartość zwrrotna nie jest ponownie używana w funkcji, więc wywołanie wprost funkcji `std::move()`, np. `return std::move(x)`, jest zbędne, a nawet może utrudniać optymalizację.

Stan obiektu, z którego nastąpiło przeniesienie, jest generalnie nieokreślony, ale wszystkie typy biblioteki standardowej pozostawiają takie obiekty w stanie, w którym można je zniszczyć lub coś do nich przypisać. Byłoby niemądrze nie podążać tym tropem. W przypadku kontenera (np. `vector` lub `string`) obiekt, z którego nastąpiło przeniesienie, będzie „pusty”. W przypadku wielu typów domyślna wartość jest dobrym stanem określającym pustkę: ma odpowiednie znaczenie i nie zajmuje dużo zasobów.

Ważnym zastosowaniem przenoszenia jest dalsze przekazywanie argumentów (8.4.2). Czasami trzeba przesłać zbiór argumentów do innej funkcji, nic w nich nie zmieniając (aby osiągnąć „doskonałe przekazywanie”):

```
template<typename T, typename... Args>
unique_ptr<T> make_unique(Args&&... args)
{
    return unique_ptr<T>{new T{std::forward<Args>(args)...}}; // przekazuje każdy argument
}
```

Funkcja `forward()` z biblioteki standardowej różni się od prostszej funkcji `std::move()` tym, że prawidłowo radzi sobie z zawiłymi dotychczasymi `r`-wartościami i `l`-wartościami (6.2.2). Funkcji `std::forward()` należy używać wyłącznie do przekazywania i niczego nie powinno się przekazywać więcej niż raz. Przekazany dalej obiekt nie należy już do nas.

16.7 Manipulowanie bitami

W nagłówku `<bit>` znajdują się funkcje służące do niskopoziomowej pracy z bitami. Jest to specjalistyczny, ale często niezbędny rodzaj czynności. Podczas pracy na poziomie bliskim sprzętu zwykle mamy do czynienia z bitami, zmieniamy ich ułożenie w bajtach i słowach oraz zamieniamy surową pamięć na typizowane obiekty. Na przykład typ `bit_cast` umożliwia konwersję wartości jednego typu na inny typ o takim samym rozmiarze:

```
double val = 7.2;
auto x = bit_cast<uint64_t>(val); // pobiera bitową reprezentację 64-bitowej liczby zmiennoprzecinkowej
auto y = bit_cast<uint64_t>(&val); // pobiera bitową reprezentację 64-bitowego wskaźnika

struct Word { std::byte b[8]; };
std::byte buffer[1024];
// ...
auto p = bit_cast<Word*>(&buffer[i]); // p wskazuje na 8 bajtów
auto i = bit_cast<int64_t>(*p); // konwersja tych 8 bajtów na liczbę całkowitą
```

Typ `std::byte` (przedrostek `std::` jest konieczny) z biblioteki standardowej reprezentuje bajty, ale nie jako reprezentacje znaków lub liczb całkowitych. Ma tylko bitowe operacje logiczne, więc

nie obsługuje operacji arytmetycznych. Najlepsze typy do wykonywania operacji bitowych to w większości przypadków liczby całkowite bez znaku i `std::byte`. Pisząc „najlepsze”, mam na myśli najszybsze i powodujące najmniej niespodzianek. Na przykład:

```
void use(unsigned int ui)
{
    int x0 = bit_width(ui)           // najmniejsza liczba bitów potrzebna do reprezentacji ui
    unsigned int ui2 = rotl(ui,8)    // obrót w lewo o 8 bitów (uwaga: nie zmienia ui)
    int x1 = popcount(ui);          // liczba jedynek w ui
    // ...
}
```

Zobacz także typ `bitset` (15.3.2).

16.8 Zamykanie programu

Czasami program napotyka problemy, z którymi nie jest w stanie sobie poradzić:

- Jeśli dany rodzaj błędu występuje często i bezpośredni moduł wywołujący powinien go obsłużyć, należy zwracać jakiś kod zwrotu (4.4).
- Jeśli dany rodzaj błędu występuje rzadko lub bezpośredni moduł wywołujący nie powinien go obsłużyć, należy zgłosić wyjątek (4.4).
- Jeśli błąd jest tak poważny, że żadna zwykła część programu nie jest w stanie go obsłużyć, należy zamknąć program.

Biblioteka standardowa zawiera narzędzia potrzebne w ostatnim z wymienionych przypadków (zamykanie programu):

- `exit(x)` — wywołuje funkcje zarejestrowane w funkcji `atexit()`, a następnie zamyka program z wartością zwrotną `x`. Jeśli masz taką potrzebę, poszukaj informacji o funkcji `atexit()` — jest to zasadniczo prosty mechanizm destrukcji dzielony z językiem C.
- `abort()` — zamyka program natychmiast i bezwarunkowo z wartością zwrotną oznaczającą nieudane zakończenie. Niektóre systemy operacyjne udostępniają narzędzia, które wymuszają modyfikację tego prostego wyjaśnienia.
- `quick_exit(x)` — wywołuje funkcje zarejestrowane w funkcji `at_quick_exit()`, a następnie zamyka program z wartością zwrotną `x`.
- `terminate()` — wywołuje procedurę `terminate_handler`. Domyślna procedura `terminate_handler` to `abort()`.

Te funkcje są przeznaczone do obsługi poważnych błędów. Nie wywołują destruktorów, tzn. nie wykonują zwykłych czynności porządkowych. Różne procedury obsługi pozwalają wykonywać pewne czynności przed zamknięciem programu. Muszą one być bardzo proste, ponieważ jedną z przyczyn ich wywołania jest to, że program znalazł się w nieprawidłowym stanie. Jedną z sensownych i całkiem popularnych czynności jest „restart systemu do dobrze zdefiniowanego stanu bez obecnego programu”. Inną, odrobinę bardziej ryzykowną, ale też niepozbawioną sensu czynnością jest „zapisanie wiadomości o błędzie w dzienniku i zamknięcie programu”. Problem

z zapisaniem wiadomości w dzienniku polega na tym, że system wejścia-wyjścia może nie działać z tego samego powodu, z którego funkcja zamknięcia programu została wywołana.

Obsługa błędów to jeden z najtrudniejszych rodzajów programowania. Nawet czyste zamknięcie programu może być niełatwe.

Żadna biblioteka ogólnego przeznaczenia nie powinna zamykać programu bezwarunkowo.

16.9 Porady

- [1] Biblioteka nie musi być duża ani skomplikowana, aby była przydatna (16.1).
- [2] Mierz czas wykonywania swoich programów, zanim cokolwiek stwierdzisz na temat ich wydajności (16.2.1).
- [3] Używaj `duration_cast`, aby zwracać wyniki pomiarów czasu wykonywania we właściwej jednostce (16.2.1).
- [4] Aby zaprezentować datę bezpośrednio w kodzie źródłowym, użyj notacji symbolicznej (np. `November/28/2021`) (16.2.2).
- [5] Jeśli data jest wynikiem obliczeń, sprawdź jej poprawność za pomocą funkcji `ok()` (16.2.2).
- [6] Do pracy z wartościami czasowymi w różnych miejscach używaj `zoned_time` (16.2.3).
- [7] Do wyrażania drobnych zmian w konwencjach wywoływania używaj `lambda` (16.3.1).
- [8] Używaj funkcji `mem_fn()` lub `lambd` do tworzenia obiektów funkcyjnych mogących wywoływać funkcję składową po wywołaniu przy użyciu tradycyjnej notacji wywoływania funkcji (16.3.1, 16.3.2).
- [9] Używaj `function`, gdy chcesz zapisać coś, co można wywołać (16.3.3).
- [10] Używaj koncepcji zamiast bezpośrednio używać predykatów typów (16.4.1).
- [11] Możesz pisać kod wprost zależny od właściwości typów (16.4.1, 16.4.2).
- [12] Preferuj koncepcje zamiast cech i `enable_if` zawsze, gdy to możliwe (16.4.3).
- [13] Używaj klasy `source_location` do wstawiania informacji o miejscu w kodzie źródłowym do komunikatów o błędach i dziennikowych (16.5).
- [14] Unikaj używania wprost funkcji `std::move()` (16.6) [CG: ES.56].
- [15] Funkcji `std::forward()` używaj wyłącznie do przekazywania (16.6).
- [16] Nigdy nie odczytuj obiektu po wywołaniu na nim funkcji `std::move()` lub `std::forward()` (16.6).
- [17] Używaj typu `std::byte` do reprezentowania danych, które (jeszcze) nie mają określonego typu (16.7).
- [18] Do pracy z bitami używaj liczb całkowitych bez znaku lub obiektów typu `bitset` (16.7).
- [19] Zwracaj kod błędu z funkcji, jeśli jej bezpośredni wywołujący powinien zająć się problemem (16.8).
- [20] Zgłaszaj wyjątek przez funkcję, jeśli jej bezpośredni wywołujący nie powinien zająć się problemem (16.8).
- [21] Wywołuj funkcje `exit()`, `quick_exit()` lub `terminate()`, aby zamknąć program po wystąpieniu problemu, którego nie da się rozwiązać (16.8).
- [22] Żadna biblioteka ogólnego przeznaczenia nie powinna bezwarunkowo zamykać programu (16.8).

S

Skorowidz

*Są dwa rodzaje wiedzy:
kiedy posiadamy wiedzę w jakimś przedmiocie
lub kiedy wiemy,
gdzie znaleźć potrzebne informacje.
— Samuel Johnson*

A

adaptery zakresów, range adaptors, 207
algorytmy, 202
 numeryczne, 252
 numeryczne równoległe, 253
 równoległe, 203
aliasy, 116
 typów, 257
alokator, 187
 monotonic_polymorphic_resource, 188
 unsynchronized_polymorphic_resource, 188
argumenty
 funkcji, 50
 ograniczone, 106
 szablonów z ograniczeniami, 105
 wartościowe szablonów, 106
arytmetyka wektorowa, 256
asercje, 63
 oczekiwań, 65
 statyczne, 64

B

biblioteka
 algorithm, 202
 filesystem, 171
 regex, 148

standardowa
 komponenty, 137
 nagłówki, 140
 przestrzenie nazw, 138
STL, 223
blok, 21
 try, 61
blokady, 264

C

C++
 elementy
 C++11, 288
 C++14, 289
 C++17, 290
 C++20, 290
 usunięte i wycofywane, 292
 komponenty biblioteki standardowej
 C++11, 291
 C++14, 291
 C++17, 292
 C++20, 292
 model, 288
 oś czasu, 280
 problemy ze zgodnością, 295
 standardy ISO, 284

styl programowania, 286
 zgodność C i C++, 293
 CTAD, class template argument deduction, 109

D

debugowanie, 64
 definiowanie
 funkcji, 43
 koncepcji, 124
 modułu, 47, 48
 operatorów, 96
 szablonów, 115
 zmiennej, 27
 deklarowanie, 18, 42
 destruktory, 71, 83, 87
 domyślny inicjalizator składowej, 89
 dyrektywa
 #include, 47
 import, 15
 using, 50
 działania arytmetyczne, 19
 dziedziczenie, 76
 implementacji, 81
 interfejsu, 80

E

enumeratory, 37

F

fabryki, 208
 formatowanie strumieni, 163
 funkcja, 16
 abort(), 248
 accumulate(), 252
 async(), 271
 at(), 148, 181
 atexit(), 248
 c_str(), 145
 capacity(), 180
 cat(), 147
 compose(), 147
 current_exception(), 269
 data(), 145
 exit(x), 248
 finally(), 114
 find(), 272, 273
 find_any(), 272

forward(), 246
 get(), 269
 join(), 262
 lock(), 268
 main(), 15
 make_shared(), 220
 make_unique(), 220
 mem_fn(), 239
 operator(), 110
 printf(), 164, 170
 push_back(), 73, 180
 quick_exit(x), 248
 rand(), 259
 replace(), 144
 reserve(), 180
 scanf(), 170
 sort(), 194
 std::move(), 246
 substr(), 144
 terminate(), 248
 visit(), 232
 wait(), 268
 funkcje, 16
 argumenty, 50
 dedukcja typu zwrotnego, 54
 przeciążanie, 17
 przekazywanie argumentów, 51
 przyrostkowy typ zwrotny, 54
 wiązanie strukturalne, 55
 zwracane wartości, 50, 52
 czyste, 23
 matematyczne, 251
 specjalne, 252
 standardowe, 251
 składowe, 17
 typów, 240
 generatory typów, 243
 predykaty typów, 241
 typy powiązane, 244
 właściwości warunkowe, 243
 wirtualne, 75, 77

G

generatory, 208
 liczb losowych, 254, 255
 typów, 243–245
 globalna przestrzeń nazw, 22
 granice numeryczne, 257

H

hierarchie klas, 78

I

implementacja typu string, 145

import jednostki nagłówkowej, 305

inicjalizacja, 20, 31

kontenerów, 73

inicjalizatory składowych, 89

instancja, 105

instrukcja, *Patrz także* słowo kluczowe

#include, 47

if, 27, 28

if constexpr, 117

if działająca w czasie kompilacji, 117

switch, 27, 28

inteligentne wskaźniki, 100

interfejs, 42

klasy, 36

modułu, 48

iteratory, 99, 154, 183, 195

strumieni, 199

J

jednostka

nagłówkowa, 305

translacji, 45

język SFINAE, 244

K

kalendarze, 237

klasa, 33, 35, 67

complex, 254

condition_variable, 267

Container, 75

list, 183

overloaded, 232

path, 171

scoped_lock, 265

source_location, 245

thread, 277

Vector, 72

klasy

abstrakcyjne, 75

bazowe, 76

hierarchie, 78

interfejs, 36

konkretne, 67, 68

konstruktor domyślny, 70

nawigacja po hierarchii, 82

niezmienniki, 59

pochodne, 76

składowe, 36

zalety hierarchii, 80

znaków, 152

klauzula case, 28

kolejność zależności, 45

kompilacja, 45

komponenty biblioteki standardowej, 137

koncepcje, 120, 127, 210

definicja, 124

iteratorów, 212

jednoargumentowe, 128

obiektów, 212

porównywania, 211

stosowanie, 129

techniki przeciążania, 122

używanie, 121

wywoływalne, 212

zakresów, 214

konstruktory, 37, 87

domyślne, 70

kopiujące, 90, 92

z listą inicjalizacyjną, 73

kontener, 71, 177, 223

array, 224

basic_string, 224

bitset, 224, 226

lista, 183

pair, 224, 227

słownik, 184

tuple, 224, 229

valarray, 224

vector<bool>, 224

wektor, 178

kontenery

heterogeniczne, 224

homogeniczne, 224

inicjalizacja, 73

kopiowanie, 90

operacje, 99, 190

przenoszenie, 92

standardowe, 189

zastosowania iteratorów, 195

konwersje typów, 74, 88
 arytmetyczne zwykle, 20
 zawężające, 20
 kopiowanie, 89
 kontenerów, 90
 krotka, tuple, 229

L

lambdy
 generyczne, 113
 jako adaptery, 239
 liczby, 250
 losowe, 254
 zespolone, 254
 lifting, 131
 lista
 dwukierunkowa, 183
 iterator, 183
 jednokierunkowa forward_list, 184
 przechwytyjąca, capture list, 111
 literały zdefiniowane przez użytkownika, 100

Ł

łańcuchy, 144
 formatowania, 165
 w stylu języka C, 26

M

makro assert(), 64
 manipulatory, 163
 mapowanie sprzętowe, 29
 mechanizm static_assert, 65
 metaprogramowanie, 241
 model
 iteracyjny, 99
 kompilacji szablonów, 134
 moduł std, 303
 moduły, 42, 43, 46, 140
 definiowanie, 46, 48
 symulacja, 45
 muteksy, 264

N

nagłówki, 140
 nawiasy klamrowe, 14

nazwa
 globalna, 22
 lokalna, 21
 składowa, 21
 składowa przestrzeni nazw, 21
 niespójności, 45
 niezmienniki, 59

O

obiekt, 18
 any, 233
 span, 222
 string_view, 147
 typu Vector, 36
 variant, 231
 obiekty
 funkcyjne, 110, 261
 future, 261
 kopiowanie, 87
 polityki, policy object, 111
 przenoszenie, 87
 obietnice, 269
 obsługa błędów, 57
 RAII, 59
 sposoby, 61
 operacja
 hash<>, 100
 swap(), 100
 operacje
 klasy path, 173
 kontenerów, 99
 kontenerów standardowych, 190
 kopiowania, 87
 na systemie plików, 174
 parametryzowane, 109
 podstawowe, 86, 87
 porównywania, 97
 przenoszenia, 87
 standardowe, 97
 wejścia i wyjścia, 100
 operator
 [], 144
 [](), 36, 59
 +(), 53
 <=>, 98
 =delete, 88
 ->, 35, 243
 aplikacji, 110

operator
 delete, 34, 72, 187
 delete[], 72
 dopełnienia, ~, 72
 new, 34, 187
 static_cast, 74
 wejścia >>, 27, 159
 wyjścia <<, 15, 27, 158

operatory
 definiowanie, 96
 deklaracji, 25
 literałów, 101
 przeciążanie, 95
 relacyjne, 97

opuszczanie kopiowania, copy elision, 53

P

pakiet parametrów, parameter pack, 132
 pamięć wolna, 34
 pętla
 for, 28
 while, 27
 planer, 275
 pliki
 .cpp, 45
 nagłówkowe, 43, 44
 źródłowe, 14
 pole typu, 40
 polimorfizm, 275
 potoki, 209
 predykaty, 111, 201
 typów, type predicate, 241, 242

programowanie
 generyczne, 119, 129
 abstrakcje, 129
 stosowanie koncepcji, 129
 obiektowe i generyczne, 68
 proceduralne, 13

programy, 14
 przeciążanie
 funkcji, 17
 oparte na koncepcjach, 122
 operatorów, operator overloading, 95

przedrostek
 std::, 15
 template<typename T>, 104

przejściowość, 45
 przekazywanie argumentów, 51, 133, 263

przenoszenie, 89
 kontenerów, 92

przestrzeń nazw, 49, 138
 ranges, 140

przypisanie, 29
 kopiujące, 90, 92
 przenoszące, 93

przyrostek sv, string view, 147

R

RAII, Resource Acquisition Is Initialization, 59, 73,
 95, 219, 265

referencje, 24
 do r-wartości, 93

reguła zera, 88

rzutowanie, 74

S

sekwencja specjalna \n, 15

sekwencje, 99, 120, 194

składniki
 biblioteki standardowej, 14
 rdzenne, 14

składowe
 prywatne, private, 36
 publiczne, public, 36

słownik, map, 184
 klucz, 185
 nieuporządkowany, 185
 wartość, 185

słowo kluczowe
 auto, 21, 127
 import
 case, 28
 class, 36–39
 const, 22, 70
 consteval, 23
 constexpr, 22
 enum, 37
 explicit, 89
 import, 15, 47
 module, 46
 noexcept, 62, 65
 nullptr, 26
 override, 75, 76
 requires, 122
 struct, 34

- throw, 58
- using, 49
- virtual, 75
- span, 221
- specjalizacja, 105
- specyfikator const, 22, 70
- stałe, 22
 - matematyczne, 258
- stan wejścia–wyjścia, 161
- sterta, 34
- strefy czasowe, 238
- struktury danych, 34
- strumienie
 - iteratory, 199
 - łańcuchowe, 168
 - pamięci, 169
 - plikowe, 168
 - stan, 161
 - standardowe, 167
 - zsynchronizowane, 170
- strumień
 - istream, 161
 - istream, 158, 159
 - ostream, 157, 158
- system plików, 171
- szablon packaged_task, 271
- szablony, 103
 - argumenty
 - domyślne, 125
 - wartościowe, 106
 - z ograniczeniami, 105
 - dedukcja argumentów, 107
 - model kompilacji, 134
 - funkcji, 109
 - ograniczone, 106
 - zmienne, variadic template, 115, 131

Ś

- ścieżki, 171

T

- tablica, array, 24, 224
 - asocjacyjna, 185
 - funkcji wirtualnych, 77
- testy, 27
- typ, 18
 - bit_cast, 247

- bitset, 224, 226
- char, 146
- complex, 69
- container, 75
- function, 239
- future, 269
- istream, 158
- optional, 232
- ostream, 157, 158
- packaged_task, 270
- pair, 224, 227
- promise, 269
- regular, 212
- span, 222
- std::byte, 247
- string, 144, 145
- string_view, 143, 146, 147
- union, 230
- unique_ptr, 83
- variant, 40
- vector, 34, 178
- void, 16
- typy
 - abstrakcyjne, 74
 - iteratorów, 198
 - koncepcji, 210
 - konkretne, 68
 - parametryzowane, 104
 - plików, 175
 - podstawowe, 18
 - polimorficzne, 75
 - powiązane, 244
 - wbudowane, 33
 - zdefiniowane przez użytkownika, 33, 37

U

- uchwyt do zasobów, 90
- UDL, user-defined literal, 101
- unie, 39

W

- wartość, 18
 - nullptr, 26
- wątki, 261
 - funkcji accumulate(), 270
 - oczekiwanie na zdarzenia, 267
 - zatrzymywanie, 272

wejście i wyjście, 159
 dla własnych typów danych, 162
 w stylu języka C, 170

wektor
 elementy, 180
 składowe, 180
 sprawdzanie zakresu, 181

widoki, 206
 łańcuchów, 146

wielodziedziczenie, 283

wielozadaniowość kooperacyjna, cooperative
 multitasking, 275

wskaźnik, 24, 217
 shared_ptr, 218, 219
 unique_ptr, 218, 219

wskaźniki
 będące właścicielem, 217
 do obiektów, 83
 niebędące właścicielem, 217
 puste, 25, 26
 sposobu wykonywania, 203

współbieżność, 260

współprocedury, 273

wycieki, 83

wyjątek, 58
 bad_any_access, 233
 out_of_range, 58, 181

wyjście, 158
 formatowanie, 163

wykonywanie
 równoległe, 203
 wektoryzowane, 203

wyliczenia, 33, 37

wyrażenia
 lambda, 111
 regularne, 148
 biblioteka regex, 148
 grupowanie, 154
 iteratory, 154
 klasy znaków, 152

notacja, 150
 reguła maksymalnego kęsa, 151
 wyszukiwanie, 149
 znaki specjalne, 150
 zwijania, 132

wyrażenie stałe, 23

wyróżnik, 40

Z

zadania, 261
 komunikacja, 268
 packaged_task, 261
 przekazywanie argumentów, 263
 zwracanie wyników, 263

zakres, range, 131, 205
 klasowy, 21
 lokalny, 21
 przestrzeni nazw, 21

zamykanie programu, 248

zarządzanie zasobami, 94

zasoby niepamięciowe, 95

zbiór bitów, bitset, 224, 226

zdarzenia, 267

zegary, 236

ziarno, 256

zmienne, 18
 atomowe, 266
 condition_variable, 267
 definiowanie, 34

znacznik, 40

znaki
 kropka, 35
 podwójny ukośnik, //, 14
 specjalne, 15, 150
 wsteczny ukośnik, \, 15

zwijanie
 lewostronne, left fold, 133
 prawostronne, right fold, 133

PROGRAM PARTNERSKI

— GRUPY HELION —

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

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

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

GRUPA
Helion 



Oto nowoczesny C++: poznaj jego ukryte możliwości!

C++ zmienił się nie do poznania: jest nowocześniejszy, bardziej precyzyjny, pozwala też na pisanie zwięzłego i efektywnego kodu. Programista ma dokładniejszą kontrolę nad działaniem programu, co przekłada się na szybkość pracy i lepsze wykorzystywanie zasobów sprzętowych. Dodatkowo ekosystem C++ oferuje mnóstwo bibliotek, narzędzi czy środowisk programistycznych. Aby jednak pisać w nim dobry kod, trzeba sprawnie poruszać się po świecie C++.

To drugie wydanie zwięzłego przewodnika po C++ dla doświadczonych programistów, zaktualizowane do standardu C++20. Dzięki niemu zaznajomisz się z najważniejszymi elementami języka i biblioteki standardowej, koniecznymi do efektywnej pracy w takich stylach programowania jak programowanie zorientowane obiektowo czy generyczne. W tym wydaniu opisano wiele nowych elementów dodanych w C++20, w tym moduły, koncepcje, współprocedury i zakresy. Omówiono też wybrane komponenty biblioteki, które pojawiają się dopiero w standardzie C++23.

Jeśli jesteś programistą C lub C++ i zależy Ci, by lepiej poznać najnowsze możliwości języka C++, albo biegle posługujesz się innym językiem programowania i chcesz ogólnie zaznajomić się z zaletami nowoczesnego C++ — nie znajdziesz bardziej zwięzłego i prostszego przewodnika niż ten.

W książce między innymi:

- nowe możliwości języka w standardzie C++20
- moduły, klasy i obsługa błędów
- operacje, zarządzanie zasobami i wejście-wyjście
- generatory, potoki, kontenery
- współbieżność i wielozadaniowość



Dr Bjarne Stroustrup jest duńskim informatykiem, twórcą języka C++ i autorem wielu książek. Obecnie pełni funkcję dyrektora działu technologicznego banku Morgan Stanley w Nowym Jorku, jest też profesorem wizytującym na Columbia University. Laureat wielu nagród, członek National Academy of Engineering, IEEE Fellow, CHM Fellow i Churchill College Cambridge Fellow.

Helion
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!



ISBN 978-83-8322-525-8



Cena: 79,00 zł

Pearson
Addison-Wesley