

C++11

JĘZYK

C++

KOMPENDIUM WIEDZY

WYDANIE IV

BJARNE STROUSTRUP

TWÓRCA JĘZYKA C++

Helion

Tytuł oryginału: The C++ Programming Language, 4th Edition

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-8329-6

Authorized translation from the English language edition, entitled:
THE C++ PROGRAMMING LANGUAGE, FOURTH EDITION;
ISBN 0321563840; by Bjarne Stroustrup; published by Pearson Education, Inc, publishing as Addison
Wesley.

Copyright © 2013 by Pearson Education.

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 © 2014, 2021.

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.

Helion S.A.

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/jcppkv>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

Spis treści

Przedmowa	23
Przedmowa do wydania trzeciego	27
Przedmowa do wydania drugiego	29
Przedmowa do wydania pierwszego	31
CZĘŚĆ I. WPROWADZENIE	33
Rozdział 1. Uwagi do czytelnika	35
1.1. Struktura książki	35
1.1.1. Wprowadzenie	36
1.1.2. Podstawowe narzędzia	36
1.1.3. Techniki abstrakcji	37
1.1.4. Biblioteka standardowa	39
1.1.5. Przykłady i odwołania	40
1.2. Projekt języka C++	41
1.2.1. Styl programowania	43
1.2.2. Kontrola typów	46
1.2.3. Zgodność z językiem C	47
1.2.4. Język, biblioteki i systemy	48
1.3. Nauka języka C++	50
1.3.1. Programowanie w języku C++	52
1.3.2. Rady dla programistów C++	53
1.3.3. Rady dla programistów C	53
1.3.4. Rady dla programistów języka Java	54
1.4. Historia	55
1.4.1. Oś czasu	56
1.4.2. Pierwsze lata	57
1.4.3. Standard z 1998 r.	59
1.4.4. Standard z 2011 r.	62
1.4.5. Do czego jest używany język C++	65
1.5. Rady	67
1.6. Literatura	68

Rozdział 2. Kurs języka C++ . Podstawy	73
2.1. Wprowadzenie	73
2.2. Podstawy	74
2.2.1. Witaj, świecie!	75
2.2.2. Typy, zmienne i arytmetyka	76
2.2.3. Stałe	78
2.2.4. Testy i pętle	79
2.2.5. Wskaźniki, tablice i pętle	80
2.3. Typy zdefiniowane przez użytkownika	82
2.3.1. Struktury	83
2.3.2. Klasy	84
2.3.3. Wyliczenia	86
2.4. Modułowość	87
2.4.1. Osobna kompilacja	88
2.4.2. Przestrzeń nazw	89
2.4.3. Obsługa błędów	90
2.5. Posłowie	93
2.6. Rady	93
Rozdział 3. Kurs języka C++ . Techniki abstrakcji	95
3.1. Wprowadzenie	95
3.2. Klasy	96
3.2.1. Typy konkretne	96
3.2.2. Typy abstrakcyjne	101
3.2.3. Funkcje wirtualne	103
3.2.4. Hierarchie klas	104
3.3. Kopiowanie i przenoszenie	108
3.3.1. Kopiowanie kontenerów	108
3.3.2. Przenoszenie kontenerów	110
3.3.3. Zarządzanie zasobami	112
3.3.4. Tłumienie operacji	113
3.4. Szablony	113
3.4.1. Typy parametryzowane	114
3.4.2. Szablony funkcji	115
3.4.3. Obiekty funkcyjne	116
3.4.4. Zmienne szablony	118
3.4.5. Aliasy	119
3.5. Rady	120
Rozdział 4. Kurs języka C++ . Kontenery i algorytmy	121
4.1. Biblioteki	121
4.1.1. Przegląd biblioteki standardowej	122
4.1.2. Nagłówki i przestrzeń nazw biblioteki standardowej	123
4.2. Łańcuchy	124
4.3. Strumienie wejścia i wyjścia	126
4.3.1. Wyjście	126
4.3.2. Wejście	127
4.3.3. Wejście i wyjście typów zdefiniowanych przez użytkownika	128

4.4. Kontenery	129
4.4.1. vector	130
4.4.2. list	133
4.4.3. map	134
4.4.4. unordered_map	135
4.4.5. Przegląd kontenerów	135
4.5. Algorytmy	137
4.5.1. Używanie iteratorów	138
4.5.2. Typy iteratorów	140
4.5.3. Iteratory strumieni	140
4.5.4. Predykaty	142
4.3.3. Przegląd algorytmów	143
4.5.6. Algorytmy kontenerowe	143
4.6. Rady	144
Rozdział 5. Kurs języka C++ . Współbieżność i narzędzia	145
5.1. Wprowadzenie	145
5.2. Zarządzanie zasobami	146
5.2.1. unique_ptr i shared_ptr	146
5.3. Współbieżność	148
5.3.1. Zadania i wątki	149
5.3.2. Przekazywanie argumentów	150
5.3.3. Zwracanie wyników	150
5.3.4. Wspólne używanie danych	151
5.3.5. Komunikacja między zadaniami	154
5.4. Drobne, ale przydatne narzędzia	157
5.4.1. Czas	157
5.4.2. Funkcje typowe	158
5.4.3. pair i tuple	160
5.5. Wyrażenia regularne	161
5.6. Matematyka	162
5.6.1. Funkcje i algorytmy matematyczne	162
5.6.2. Liczby zespolone	163
5.6.3. Liczby losowe	163
5.6.4. Arytmetyka wektorów	165
5.6.5. Limity liczbowe	165
5.7. Rady	166
CZĘŚĆ II. PODSTAWOWE NARZĘDZIA	167
Rozdział 6. Typy i deklaracje	169
6.1. Standard ISO języka C++	169
6.1.1. Implementacje	171
6.1.2. Podstawowy źródłowy zestaw znaków	171
6.2. Typy	172
6.2.1. Typy podstawowe	172
6.2.2. Typ logiczny	173
6.2.3. Typy znakowe	174
6.2.4. Typy całkowitoliczbowe	179
6.2.5. Typy zmiennoprzecinkowe	181

6.2.6. Przedrostki i przyrostki	182
6.2.7. void	183
6.2.8. Rozmiary	183
6.2.9. Wyrównanie	185
6.3. Deklaracje	186
6.3.1. Struktura deklaracji	188
6.3.2. Deklarowanie po kilka nazw	189
6.3.3. Nazwy	189
6.3.4. Zakres dostępności	191
6.3.5. Inicjacja	194
6.3.6. Dedukowanie typu: auto i decltype()	197
6.4. Obiekty i wartości	200
6.4.1. Wartości lewo- i prawostronne	200
6.4.2. Cykl istnienia obiektów	201
6.5. Aliasy typów	202
6.6. Rady	203
Rozdział 7. Wskaźniki, tablice i referencje	205
7.1. Wprowadzenie	205
7.2. Wskaźniki	205
7.2.1. void*	206
7.2.2. nullptr	207
7.3. Tablice	208
7.3.1. Inicjatory tablic	209
7.3.2. Literały łańcuchowe	210
7.4. Wskaźniki do tablic	213
7.4.1. Przeglądanie tablic	214
7.4.2. Tablice wielowymiarowe	217
7.4.3. Przekazywanie tablic	217
7.5. Wskaźniki i const	220
7.6. Wskaźniki i własność	221
7.7. Referencje	222
7.7.1. Referencje lewostronne	224
7.7.2. Referencje prawostronne	227
7.7.3. Referencje do referencji	229
7.7.4. Wskaźniki i referencje	230
7.8. Rady	232
Rozdział 8. Struktury, unie i wyliczenia	233
8.1. Wprowadzenie	233
8.2. Struktury	234
8.2.1. Układ struktur	235
8.2.2. Nazwy struktur	236
8.2.3. Struktury a klasy	237
8.2.4. Struktury a tablice	239
8.2.5. Ekwiwalencja typów	241
8.2.6. Stare zwykłe dane	241
8.2.7. Pola	244

8.3. Unie	244
8.3.1. Unie a klasy	246
8.3.2. Anonimowe unie	247
8.4. Wyliczenia	249
8.4.1. Klasy wyliczeniowe	250
8.4.2. Zwyczajne wyliczenia	253
8.4.3. Wyliczenia anonimowe	254
8.5. Rady	255
Rozdział 9. Instrukcje	257
9.1. Wprowadzenie	257
9.2. Zestawienie instrukcji	258
9.3. Deklaracje jako instrukcje	259
9.4. Instrukcje wyboru	260
9.4.1. Instrukcje if	260
9.4.2. Instrukcje switch	261
9.4.3. Deklaracje w warunkach	264
9.5. Instrukcje iteracyjne	264
9.5.1. Zakresowe instrukcje for	265
9.5.2. Instrukcje for	266
9.5.3. Instrukcje while	267
9.5.4. Instrukcje do	267
9.5.5. Kończenie pętli	268
9.6. Instrukcje goto	269
9.7. Komentarze i wcięcia	269
9.8. Rady	271
Rozdział 10. Wyrażenia	273
10.1. Wprowadzenie	273
10.2. Kalkulator	273
10.2.1. Parser	274
10.2.2. Wejście	278
10.2.3. Wejście niskopoziomowe	282
10.2.4. Obsługa błędów	283
10.2.5. Sterownik	284
10.2.6. Nagłówki	284
10.2.7. Argumenty wiersza poleceń	285
10.2.8. Uwaga na temat stylu	286
10.3. Zestawienie operatorów	287
10.3.1. Wyniki	291
10.3.2. Kolejność wykonywania działań	292
10.3.3. Priorytety operatorów	292
10.3.4. Obiekty tymczasowe	293
10.4. Wyrażenia stałe	295
10.4.1. Stałe symboliczne	297
10.4.2. const w wyrażeniach stałych	297
10.4.3. Typy literalowe	297
10.4.4. Argumenty referencyjne	298
10.4.5. Wyrażenia stałe adresowe	299

10.5. Niejawna konwersja typów	299
10.5.1. Promocje	300
10.5.2. Konwersje	300
10.5.3. Typowe konwersje arytmetyczne	303
10.6. Rady	304
Rozdział 11. Operacje wyboru	305
11.1. Różne operatory	305
11.1.1. Operatory logiczne	305
11.1.2. Bitowe operatory logiczne	306
11.1.3. Wyrażenia warunkowe	307
11.1.4. Inkrementacja i dekrementacja	307
11.2. Pamięć wolna	309
11.2.1. Zarządzanie pamięcią	311
11.2.2. Tablice	313
11.2.3. Sprawdzanie dostępności miejsca w pamięci	314
11.2.4. Przeciążanie operatora new	315
11.3. Listy	318
11.3.1. Model implementacji	318
11.3.2. Listy kwalifikowane	319
11.3.3. Listy niekwalifikowane	320
11.4. Wyrażenia lambda	322
11.4.1. Model implementacji	322
11.4.2. Alternatywy dla lambda	323
11.4.3. Lista zmiennych	325
11.4.4. Wywoływanie i zwracanie wartości	329
11.4.5. Typ lambda	329
11.5. Jawna konwersja typów	330
11.5.1. Konstrukcja	331
11.5.2. Rzutowania nazwane	333
11.5.3. Rzutowanie w stylu języka C	334
11.5.4. Rzutowanie w stylu funkcyjnym	335
11.6. Rady	335
Rozdział 12. Funkcje	337
12.1. Deklarowanie funkcji	337
12.1.1. Dlaczego funkcje	338
12.1.2. Składniki deklaracji funkcji	338
12.1.3. Definiowanie funkcji	339
12.1.4. Zwracanie wartości	340
12.1.5. Funkcje inline	342
12.1.6. Funkcje constexpr	343
12.1.7. Funkcje [[noreturn]]	346
12.1.8. Zmienne lokalne	346
12.2. Przekazywanie argumentów	347
12.2.1. Argumenty referencyjne	348
12.2.2. Argumenty tablicowe	350
12.2.4. Nieokreślona liczba argumentów	353
12.2.5. Argumenty domyślne	356

12.3. Przeciążanie funkcji	358
12.3.1. Automagiczne wybieranie przeciążonych funkcji	358
12.3.2. Przeciążanie a typ zwrotny	360
12.3.3. Przeciążanie a zakres	360
12.3.4. Wybieranie przeciążonych funkcji z wieloma argumentami	361
12.3.5. Ręczne wybieranie przeciążonej funkcji	362
12.4. Warunki wstępne i końcowe	362
12.5. Wskaźnik do funkcji	364
12.6. Makra	368
12.6.1. Kompilacja warunkowa	370
12.6.2. Makra predefiniowane	371
12.6.3. Pragmy	372
12.7. Rady	372
Rozdział 13. Obsługa wyjątków	373
13.1. Obsługa błędów	373
13.1.1. Wyjątki	374
13.1.2. Tradycyjna obsługa błędów	376
13.1.3. Niedbała obsługa błędów	377
13.1.4. Alternatywne spojrzenie na wyjątki	378
13.1.5. Kiedy nie można używać wyjątków	379
13.1.6. Hierarchiczna obsługa błędów	380
13.1.7. Wyjątki a wydajność	381
13.2. Gwarancje wyjątków	383
13.3. Zarządzanie zasobami	385
13.3.1. Finalizacja	388
13.4. Egzekwowanie przestrzegania niezmienników	389
13.5. Zgłaszanie i przechwytywanie wyjątków	394
13.5.1. Zgłaszanie wyjątków	394
13.5.2. Przechwytywanie wyjątków	397
13.5.3. Wyjątki a wątki	404
13.6. Implementacja wektora	405
13.6.1. Prosty wektor	405
13.6.2. Jawna reprezentacja pamięci	409
13.6.3. Przypisywanie	411
13.6.4. Zmienianie rozmiaru	414
13.7. Rady	416
Rozdział 14. Przestrzenie nazw	419
14.1. Kwestie dotyczące kompozycji	419
14.2. Przestrzenie nazw	420
14.2.1. Bezpośrednia kwalifikacja	422
14.2.2. Deklaracje using	423
14.2.3. Dyrektywy using	424
14.2.4. Wyszukiwanie wg argumentów	425
14.2.5. Przestrzenie nazw są otwarte	427
14.3. Modularyzacja i interfejsy	428
14.3.1. Przestrzenie nazw i moduły	430
14.3.2. Implementacje	431
14.3.3. Interfejsy i implementacje	433

14.4. Składanie przy użyciu przestrzeni nazw	435
14.4.1. Wygoda a bezpieczeństwo	435
14.4.2. Aliasy przestrzeni nazw	436
14.4.3. Składanie przestrzeni nazw	436
14.4.4. Składanie i wybieranie	438
14.4.5. Przestrzenie nazw a przeciążanie	439
14.4.6. Wersjonowanie	441
14.4.7. Zagnieżdżanie przestrzeni nazw	443
14.4.8. Anonimowe przestrzenie nazw	444
14.4.9. Nagłówki języka C	444
14.5. Rady	445
Rozdział 15. Pliki źródłowe i programy	447
15.1. Rozdzielna kompilacja	447
15.2. Konsolidacja	448
15.2.1. Nazwy lokalne w plikach	451
15.2.2. Pliki nagłówkowe	451
15.2.3. Reguła jednej definicji	453
15.2.4. Nagłówki z biblioteki standardowej	455
15.2.5. Konsolidacja z kodem w innym języku	456
15.2.6. Konsolidacja a wskaźniki do funkcji	458
15.3. Używanie plików nagłówkowych	459
15.3.1. Organizacja z jednym nagłówkiem	459
15.3.2. Organizacja z wieloma nagłówkami	463
15.3.3. Strażnicy dołączania	467
15.4. Programy	468
15.4.1. Inicjacja zmiennych nielokalnych	469
15.4.2. Inicjacja i współbieżność	470
15.4.3. Zamykanie programu	470
15.5. Rady	472
CZĘŚĆ III. TECHNIKI ABSTRAKCJI	473
Rozdział 16. Klasy	475
16.1. Wprowadzenie	475
16.2. Podstawowe wiadomości o klasach	476
16.2.1. Funkcje składowe	477
16.2.2. Kopiowanie domyślne	478
16.2.3. Kontrola dostępu	479
16.2.4. Klasy i struktury	480
16.2.5. Konstruktory	481
16.2.6. Konstruktory explicit	483
16.2.7. Inicjatory wewnątrzklasowe	485
16.2.8. Wewnątrzklasowe definicje funkcji	486
16.2.9. Zmienność	487
16.2.10. Słowo kluczowe this	490
16.2.11. Dostęp do składowych	491
16.2.12. Składowe statyczne	492
16.2.13. Typy składowe	494

16.3. Klasy konkretne	495
16.3.1. Funkcje składowe	498
16.3.2. Funkcje pomocnicze	500
16.3.3. Przeciążanie operatorów	502
16.3.4. Znaczenie klas konkretnych	503
16.4. Rady	504
Rozdział 17. Tworzenie, kasowanie, kopiowanie i przenoszenie	505
17.1. Wprowadzenie	505
17.2. Konstruktory i destruktory	507
17.2.1. Konstruktory i niezmienniki	508
17.2.2. Destruktry i zasoby	509
17.2.3. Destruktry klas bazowych i składowych klas	510
17.2.4. Wywoływanie konstruktorów i destruktorów	511
17.2.5. Destruktry wirtualne	512
17.3. Inicjacja obiektów klas	513
17.3.1. Inicjacja bez konstruktorów	513
17.3.2. Inicjacja przy użyciu konstruktorów	515
17.3.3. Konstruktory domyślne	517
17.3.4. Konstruktory z listą inicjacyjną	519
17.4. Inicjacja składowych i bazy	524
17.4.1. Inicjacja składowych	524
17.4.2. Inicjatory bazy	525
17.4.3. Delegowanie konstruktorów	526
17.4.4. Inicjatory wewnątrzklasowe	527
17.4.5. Inicjacja składowych statycznych	529
17.5. Kopiowanie i przenoszenie	530
17.5.1. Kopiowanie	530
17.5.2. Przenoszenie	537
17.6. Generowanie domyślnych operacji	541
17.6.1. Jawne operacje domyślne	541
17.6.2. Operacje domyślne	542
17.6.3. Używanie operacji domyślnych	543
17.6.4. Usuwanie funkcji	547
17.7. Rady	548
Rozdział 18. Przeciążanie operatorów	551
18.1. Wprowadzenie	551
18.2. Funkcje operatorowe	553
18.2.1. Operatory dwu- i jednoargumentowe	554
18.2.2. Predefiniowane znaczenie operatorów	555
18.2.3. Operatory i typy zdefiniowane przez użytkownika	555
18.2.4. Przekazywanie obiektów	556
18.2.5. Operatory w przestrzeniach nazw	557
18.3. Typ reprezentujący liczby zespolone	559
18.3.1. Operatory składowe i zewnętrzne	559
18.3.2. Arytmetyka mieszana	560
18.3.3. Konwersje	561
18.3.4. Literały	564
18.3.5. Funkcje dostępowe	565

18.3.6. Funkcje pomocnicze	565
18.4. Konwersja typów	567
18.4.1. Operatory konwersji	567
18.4.2. Operatory konwersji explicit	569
18.4.3. Niejednoznaczności	569
18.5. Rady	571
Rozdział 19. Operatory specjalne	573
19.1. Wprowadzenie	573
19.2. Operatory specjalne	573
19.2.1. Indeksowanie	573
19.2.2. Wywoływanie funkcji	574
19.2.3. Dereferencja	576
19.2.4. Inkrementacja i dekrementacja	578
19.2.5. Alokacja i dezalokacja	580
19.2.6. Literały zdefiniowane przez użytkownika	581
19.3. Klasa String	584
19.3.1. Podstawowe operacje	585
19.3.2. Dostęp do znaków	585
19.3.3. Reprezentacja	586
19.3.4. Funkcje składowe	589
19.3.5. Funkcje pomocnicze	591
19.3.6. Sposoby użycia	593
19.4. Przyjaciele	594
19.4.1. Znajdowanie przyjaciół	596
19.4.2. Przyjaciele i składowe	597
19.5. Rady	598
Rozdział 20. Derywacja klas	599
20.1. Wprowadzenie	599
20.2. Klasy pochodne	600
20.2.1. Funkcje składowe	602
20.2.2. Konstruktory i destruktory	604
20.3. Hierarchie klas	604
20.3.1. Pola typów	605
20.3.2. Funkcje wirtualne	607
20.3.3. Bezpośrednia kwalifikacja	610
20.3.4. Kontrola przesłaniania	610
20.3.5. Używanie składowych klasy bazowej	614
20.3.6. Rozluźnienie zasady dotyczącej typów zwrotnych	617
20.4. Klasy abstrakcyjne	619
20.5. Kontrola dostępu	621
20.5.1. Składowe chronione	624
20.5.2. Dostęp do klas bazowych	625
20.5.3. Deklaracje using i kontrola dostępu	627
20.6. Wskaźniki do składowych	627
20.6.1. Wskaźniki do funkcji składowych	628
20.6.2. Wskaźniki do danych składowych	630
20.6.3. Składowe bazy i klasy pochodnej	631
20.7. Rady	631

Rozdział 21. Hierarchie klas	633
21.1. Wprowadzenie	633
21.2. Projektowanie hierarchii klas	633
21.2.1. Dziedziczenie implementacji	634
21.2.2. Dziedziczenie interfejsu	637
21.2.3. Alternatywne implementacje	639
21.2.4. Lokalizowanie tworzenia obiektu	642
21.3. Wielodziedziczenie	644
21.3.1. Wiele interfejsów	644
21.3.2. Wiele klas implementacyjnych	644
21.3.3. Rozstrzyganie niejednoznaczności	646
21.3.4. Wielokrotne użycie klasy bazowej	649
21.3.5. Wirtualne klasy bazowe	651
21.3.6. Bazy wirtualne a replikowane	655
21.4. Rady	658
Rozdział 22. Informacje o typach w czasie działania programu	659
22.1. Wprowadzenie	659
22.2. Poruszanie się w obrębie hierarchii klas	660
22.2.1. Rzutowanie dynamiczne	661
22.2.2. Wielodziedziczenie	664
22.2.3. Rzutowanie statyczne i dynamiczne	665
22.2.4. Odzyskiwanie interfejsu	667
22.3. Podwójny polimorfizm i wizytatorzy	670
22.3.1. Podwójny polimorfizm	671
22.3.2. Wizytatorzy	673
22.4. Konstrukcja i destrukcja	675
22.5. Identyfikacja typów	675
22.5.1. Rozszerzone informacje o typie	677
22.6. Poprawne i niepoprawne używanie RTTI	678
22.7. Rady	680
Rozdział 23. Szablony	681
23.1. Wprowadzenie i przegląd	681
23.2. Prosty szablon łańcucha	684
23.2.1. Definiowanie szablonu	685
23.2.2. Konkretyzacja szablonu	687
23.3. Kontrola typów	688
23.3.1. Ekwiwalencja typów	689
23.3.2. Wykrywanie błędów	690
23.4. Składowe szablonu klasy	691
23.4.1. Dane składowe	691
23.4.2. Funkcje składowe	692
23.4.3. Aliasy typów składowych	692
23.4.4. Składowe statyczne	692
23.4.5. Typy składowe	693
23.4.6. Szablony składowe	694
23.4.7. Przyjaciele	698

23.5. Szablony funkcji	699
23.5.1. Argumenty szablonu funkcji	701
23.5.2. Dedukcja argumentów szablonu funkcji	702
23.5.3. Przeciążanie szablonów funkcji	704
23.6. Aliasy szablonów	708
23.7. Organizacja kodu źródłowego	709
23.7.1. Konsolidacja	711
23.8. Rady	712
Rozdział 24. Programowanie ogólne	713
24.1. Wprowadzenie	713
24.2. Algorytmy i uogólnianie	714
24.3. Koncepcje	718
24.3.1. Odkrywanie koncepcji	718
24.3.2. Koncepcje i ograniczenia	722
24.4. Konkretyzacja koncepcji	724
24.4.1. Aksjomaty	727
24.4.2. Koncepcje wieloargumentowe	728
24.4.3. Koncepcje wartości	729
24.4.4. Sprawdzanie ograniczeń	730
24.4.5. Sprawdzanie definicji szablonu	731
24.5. Rady	733
Rozdział 25. Specjalizacja	735
25.1. Wprowadzenie	735
25.2. Argumenty i parametry szablonu	736
25.2.1. Typy jako argumenty	736
25.2.2. Wartości jako argumenty	738
25.2.3. Operacje jako argumenty	739
25.2.4. Szablony jako argumenty	742
25.2.5. Domyślne argumenty szablonów	742
25.3. Specjalizacja	744
25.3.1. Specjalizacja interfejsu	747
25.3.2. Szablon podstawowy	748
25.3.3. Porządek specjalizacji	750
25.3.4. Specjalizacja szablonu funkcji	750
25.4. Rady	753
Rozdział 26. Konkretyzacja	755
26.1. Wprowadzenie	755
26.2. Konkretyzacja szablonu	756
26.2.1. Kiedy konkretyzacja jest potrzebna	757
26.2.2. Ręczne sterowanie konkretyzacją	758
26.3. Wiązanie nazw	759
26.3.1. Nazwy zależne	761
26.3.2. Wiązanie w miejscu definicji	762
26.3.3. Wiązanie w miejscu konkretyzacji	763
26.3.4. Wiele miejsc konkretyzacji	766

26.3.5. Szablony i przestrzenie nazw	767
26.3.6. Nadmiernie agresywne wyszukiwanie wg argumentów	768
26.3.7. Nazwy z klas bazowych	770
26.4. Rady	772
Rozdział 27. Hierarchie szablonów	773
27.1. Wprowadzenie	773
27.2. Parametryzacja i hierarchia	774
27.2.1. Typy generowane	776
27.2.2. Konwersje szablonów	778
27.3. Hierarchie szablonów klas	779
27.3.1. Szablony jako interfejsy	780
27.4. Parametry szablonowe jako klasy bazowe	781
27.4.1. Składanie struktur danych	781
27.4.2. Linearyzacja hierarchii klas	785
27.5. Rady	790
Rozdział 28. Metaprogramowanie	791
28.1. Wprowadzenie	791
28.2. Funkcje typowe	794
28.2.1. Aliasy typów	796
28.2.2. Predykaty typów	798
28.2.3. Wybieranie funkcji	799
28.2.4. Cechy	800
28.3. Struktury sterujące	802
28.3.1. Wybieranie	802
28.3.2. Iteracja i rekurencja	805
28.3.3. Kiedy stosować metaprogramowanie	806
28.4. Definicja warunkowa	807
28.4.1. Używanie Enable_if	809
28.4.2. Implementacja Enable_if	811
28.4.3. Enable_if i koncepcje	811
28.4.4. Dodatkowe przykłady użycia Enable_if	812
28.5. Lista czasu kompilacji	814
28.5.1. Prosta funkcja wyjściowa	816
28.5.2. Dostęp do elementów	818
28.5.3. make_tuple	820
28.6. Szablony zmienne	821
28.6.1. Bezpieczna typowo funkcja printf()	821
28.6.2. Szczegóły techniczne	824
28.6.3. Przekazywanie	825
28.6.4. Typ tuple z biblioteki standardowej	827
28.7. Przykład z jednostkami układu SI	830
28.7.1. Jednostki	830
28.7.2. Wielkości	831
28.7.3. Literały jednostkowe	833
28.7.4. Funkcje pomocnicze	834
28.8. Rady	836

Rozdział 29. Projekt macierzy	837
29.1. Wprowadzenie	837
29.1.1. Podstawowe zastosowania macierzy	838
29.1.2. Wymagania dotyczące macierzy	840
29.2. Szablon macierzy	841
29.2.1. Konstrukcja i przypisywanie	842
29.2.2. Indeksowanie i cięcie	843
29.3. Operacje arytmetyczne na macierzach	845
29.3.1. Operacje skalarne	846
29.3.2. Dodawanie	847
29.3.3. Mnożenie	848
29.4. Implementacja macierzy	850
29.4.1. Wycinki	850
29.4.2. Wycinki macierzy	850
29.4.3. Matrix_ref	852
29.4.4. Inicjacja listy macierzy	853
29.4.5. Dostęp do macierzy	855
29.4.6. Macierz zerowymiarowa	857
29.5. Rozwiązywanie równań liniowych	858
29.5.1. Klasyczna eliminacja Gaussa	859
29.5.2. Znajdowanie elementu centralnego	860
29.5.3. Testowanie	861
29.5.4. Połączone operacje	862
29.6. Rady	864
CZĘŚĆ IV. BIBLIOTEKA STANDARDOWA	865
Rozdział 30. Przegląd zawartości biblioteki standardowej	867
30.1. Wprowadzenie	867
30.1.1. Narzędzia biblioteki standardowej	868
30.1.2. Kryteria projektowe	869
30.1.3. Styl opisu	870
30.2. Nagłówki	871
30.3. Wsparcie dla języka	875
30.3.1. Wsparcie dla list inicjacyjnych	876
30.3.2. Wsparcie dla zakresowej pętli for	876
30.4. Obsługa błędów	877
30.4.1. Wyjątki	877
30.4.2. Asercje	882
30.4.3. system_error	882
30.5. Rady	892
Rozdział 31. Kontenery STL	893
31.1. Wprowadzenie	893
31.2. Przegląd kontenerów	893
31.2.1. Reprezentacja kontenera	896
31.2.2. Wymagania dotyczące elementów	898

31.3. Przegląd operacji	901
31.3.1. Typy składowe	904
31.3.2. Konstruktory, destruktory i przypisania	904
31.3.3. Rozmiar i pojemność	906
31.3.4. Iteratory	907
31.3.5. Dostęp do elementów	908
31.3.6. Operacje stosowe	908
31.3.7. Operacje listowe	909
31.3.8. Inne operacje	910
31.4. Kontenery	910
31.4.1. vector	911
31.4.2. Listy	915
31.4.3. Kontenery asocjacyjne	917
31.5. Adaptacje kontenerów	929
31.5.1. Stos	929
31.5.2. Kolejka	931
31.5.3. Kolejka priorytetowa	931
31.6. Rady	932
Rozdział 32. Algorytmy STL	935
32.1. Wprowadzenie	935
32.2. Algorytmy	935
32.2.1. Sekwencje	936
32.3. Argumenty zasad	938
32.3.1. Złożoność	939
32.4. Algorytmy nie modyfikujące sekwencji	940
32.4.1. for_each()	940
32.4.2. Predykaty sekwencji	940
32.4.3. count()	940
32.4.4. find()	941
32.4.5. equal() i mismatch()	942
32.4.6. search()	942
32.5. Algorytmy modyfikujące sekwencje	943
32.5.1. copy()	944
32.5.2. unique()	945
32.5.3. remove() i replace()	946
32.5.4. rotate(), random_shuffle() oraz partition()	947
32.5.5. Permutacje	948
32.5.6. fill()	948
32.5.7. swap()	949
32.6. Sortowanie i wyszukiwanie	950
32.6.1. Wyszukiwanie binarne	952
32.6.2. merge()	954
32.6.3. Algorytmy działające na zbiorach	954
32.6.4. Sterty	955
32.6.5. lexicographical_compare()	956
32.7. Element minimalny i maksymalny	957
32.8. Rady	958

Rozdział 33. Iteratory STL	959
33.1. Wprowadzenie	959
33.1.1. Model iteratorów	959
33.1.2. Kategorie iteratorów	961
33.1.3. Cechy iteratorów	962
33.1.4. Operacje iteratorów	964
33.2. Adaptacje iteratorów	965
33.2.1. Iterator odwrotny	966
33.2.2. Iteratory wstawiające	968
33.2.3. Iteratory przenoszące	969
33.3. Zakresowe funkcje dostępne	970
33.4. Obiekty funkcyjne	971
Adaptacje funkcji	972
33.5.1. bind()	972
33.5.2. mem_fn()	974
33.5.3. function	974
33.6. Rady	976
Rozdział 34. Pamięć i zasoby	977
34.1. Wprowadzenie	977
34.2. „Prawie kontenery”	977
34.2.1. array	978
34.2.2. bitset	981
34.2.3. vector<bool>	985
34.2.4. Krotki	986
34.3. Wskaźniki do zarządzania pamięcią	990
34.3.1. unique_ptr	990
34.3.2. shared_ptr	993
34.3.3. weak_ptr	996
34.4. Alokatory	998
34.4.1. Alokator domyślny	1000
34.4.2. Cechy alokatorów	1001
34.4.3. Cechy wskaźników	1002
34.4.4. Alokatory zakresowe	1003
34.5. Interfejs odśmiecacza	1004
34.6. Pamięć niezainicjowana	1007
34.6.1. Bufory tymczasowe	1007
34.6.2. raw_storage_iterator	1008
34.7. Rady	1009
Rozdział 35. Narzędzia pomocnicze	1011
35.1. Wprowadzenie	1011
35.2. Czas	1011
35.2.1. duration	1012
35.2.2. time_point	1015
35.2.3. Zegary	1017
35.2.4. Cechy czasu	1018
35.3. Działania arytmetyczne na liczbach wymiernych w czasie kompilacji	1019

35.4. Funkcje typowe	1020
35.4.1. Cechy typów	1020
35.4.2. Generatory typów	1025
35.5. Drobne narzędzia	1030
35.5.1. <code>move()</code> i <code>forward()</code>	1030
35.5.2. <code>swap()</code>	1031
35.5.3. Operatory relacyjne	1031
35.5.4. Porównywanie i mieszanie <code>type_info</code>	1032
35.6. Rady	1033
Rozdział 36. Łańcuchy	1035
36.1. Wprowadzenie	1035
36.2. Klasyfikacja znaków	1035
36.2.1. Funkcje klasyfikacji	1035
36.2.2. Cechy znaków	1036
36.3. Łańcuchy	1038
36.3.1. Typ string a łańcuchy w stylu C	1039
36.3.2. Konstruktory	1040
36.3.3. Operacje podstawowe	1042
36.3.4. Łańcuchowe wejście i wyjście	1044
36.3.5. Konwersje numeryczne	1044
36.3.6. Operacje w stylu biblioteki STL	1046
36.3.7. Rodzina funkcji <code>find</code>	1048
36.3.8. Podłańcuchy	1049
36.4. Rady	1050
Rozdział 37. Wyrażenia regularne	1053
37.1. Wyrażenia regularne	1053
37.1.1. Notacja wyrażeń regularnych	1054
37.2. <code>regex</code>	1059
37.2.1. Wyniki dopasowywania	1061
37.2.2. Formatowanie	1063
37.3. Funkcje wyrażeń regularnych	1064
37.3.1. <code>regex_match()</code>	1064
37.3.2. <code>regex_search()</code>	1066
37.3.3. <code>regex_replace()</code>	1067
37.4. Iteratory wyrażeń regularnych	1068
37.4.1. <code>regex_iterator</code>	1068
37.4.2. <code>regex_token_iterator</code>	1070
37.5. <code>regex_traits</code>	1072
37.6. Rady	1073
Rozdział 38. Strumienie wejścia i wyjścia	1075
38.1. Wprowadzenie	1075
38.2. Hierarchia strumieni wejścia i wyjścia	1077
38.2.1. Strumienie plikowe	1078
38.2.2. Strumienie łańcuchowe	1079
38.3. Obsługa błędów	1081

38.4. Operacje wejścia i wyjścia	1082
38.4.1. Operacje wejściowe	1083
38.4.2. Operacje wyjściowe	1086
38.4.3. Manipulatory	1088
38.4.4. Stan strumienia	1089
38.4.5. Formatowanie	1094
38.5. Iteratory strumieniowe	1101
38.6. Buforowanie	1102
38.6.1. Strumień wyjściowy i bufor	1105
38.6.2. Strumień wejściowy i bufor	1106
38.6.3. Iteratory buforów	1107
38.7. Rady	1109
Rozdział 39. Lokalizacje	1111
39.1. Różnice kulturowe	1111
39.2. Klasa locale	1114
39.2.1. Lokalizacje nazwane	1116
39.2.2. Porównywanie łańcuchów	1120
39.3. Klasa facet	1120
39.3.1. Dostęp do faset w lokalizacji	1121
39.3.2. Definiowanie prostej fasy	1122
39.3.3. Zastosowania lokalizacji i faset	1125
39.4. Standardowe fasy	1125
39.4.1. Porównywanie łańcuchów	1127
39.4.2. Formatowanie liczb	1131
39.4.3. Formatowanie kwot pieniężnych	1136
39.4.4. Formatowanie daty i godziny	1141
39.4.5. Klasyfikacja znaków	1144
39.4.6. Konwersja kodów znaków	1147
39.4.7. Wiadomości	1151
39.5. Interfejsy pomocnicze	1155
39.5.1. Klasyfikacja znaków	1155
39.5.2. Konwersje znaków	1156
39.5.3. Konwersje łańcuchów	1156
39.5.4. Buforowanie konwersji	1157
39.6. Rady	1158
Rozdział 40. Liczby	1159
40.1. Wprowadzenie	1159
40.2. Granice liczbowe	1160
40.2.1. Makra ograniczające	1162
40.3. Standardowe funkcje matematyczne	1163
40.4. Liczby zespolone	1164
40.5. Tablica numeryczna valarray	1166
40.5.1. Konstruktory i przypisania	1166
40.5.2. Indeksowanie	1168
40.5.3. Operacje	1169
40.5.4. Wycinki	1172
40.5.5. slice_array	1174
40.5.6. Uogólnione wycinki	1175

40.6. Uogólnione algorytmy numeryczne	1176
40.6.1. Algorytm accumulate()	1177
40.6.2. Algorytm inner_product()	1177
40.6.3. Algorytmy partial_sum() i adjacent_difference()	1178
40.6.4. Algorytm iota()	1179
40.7. Liczby losowe	1180
40.7.1. Mechanizmy	1182
40.7.2. Urządzenie losowe	1184
40.7.3. Rozkłady	1185
40.7.4. Losowanie liczb w stylu C	1189
40.8. Rady	1189
Rozdział 41. Współbieżność	1191
41.1. Wprowadzenie	1191
41.2. Model pamięci	1193
41.2.1. Lokalizacje pamięci	1194
41.2.2. Zmianianie kolejności instrukcji	1195
41.2.3. Porządek pamięci	1196
41.2.4. Wyścigi do danych	1197
41.3. Konstrukcje atomowe	1198
41.3.1. Typy atomowe	1201
41.3.2. Flagi i bariery	1205
41.4. Słowo kluczowe volatile	1207
41.5. Rady	1207
Rozdział 42. Wątki i zadania	1209
42.1. Wprowadzenie	1209
42.2. Wątki	1210
42.2.1. Tożsamość	1211
42.2.2. Konstrukcja	1212
42.2.3. Destrukcja	1213
42.2.4. Funkcja join()	1214
42.2.5. Funkcja detach()	1215
42.2.6. Przestrzeń nazw this_thread	1217
42.2.7. Likwidowanie wątku	1218
42.2.8. Dane lokalne wątku	1218
42.3. Unikanie wyścigów do danych	1220
42.3.1. Muteksy	1220
42.3.2. Wiele blokad	1228
42.3.3. Funkcja call_once()	1230
42.3.4. Zmienne warunkowe	1231
42.4. Współbieżność zadaniowa	1235
42.4.1. Typy future i promise	1236
42.4.2. Typ promise	1237
42.4.3. Typ packaged_task	1238
42.4.4. Typ future	1241
42.4.5. Typ shared_future	1244
42.4.6. Funkcja async()	1245
42.4.7. Przykład równoległej funkcji find()	1247
42.5. Rady	1251

Rozdział 43. Biblioteka standardowa C	1253
43.1. Wprowadzenie	1253
43.2. Pliki	1253
43.3. Rodzina printf()	1254
43.4. Łańcuchy w stylu C	1259
43.5. Pamięć	1260
43.6. Data i godzina	1261
43.7. Itd.	1264
43.8. Rady	1266
Rozdział 44. Zgodność	1267
44.1. Wprowadzenie	1267
44.2. Rozszerzenia C++11	1268
44.2.1. Narzędzia językowe	1268
44.2.2. Składniki biblioteki standardowej	1269
44.2.3. Elementy wycofywane	1270
44.2.4. Praca ze starszymi implementacjami C++	1271
44.3. Zgodność C i C++	1271
44.3.1. C i C++ to rodzeństwo	1271
44.3.2. „Ciche” różnice	1273
44.3.3. Kod C nie będący kodem C++	1274
44.3.4. Kod C++ nie będący kodem C	1277
44.4. Rady	1279
Skorowidz	1281

Wyrażenia

*Programowanie jest jak seks:
może dawać konkretne wyniki,
ale nie po to się to robi
— przeprosiny dla Richarda Feynmana*

- Wprowadzenie
- Kalkulator
 - Parser; Wejście; Wejście niskopoziomowe; Obsługa błędów; Sterownik; Nagłówki; Argumenty wiersza poleceń; Uwaga na temat stylu
- Zestawienie operatorów
 - Wyniki; Kolejność wykonywania działań; Kolejność wykonywania operatorów; Obiekty tymczasowe
- Wyrażenia stałe
 - Stałe symboliczne; const w wyrażeniach stałych; Typy literałów; Argumenty referencyjne; Wyrażenia stałe adresowe
- Niejawna konwersja typów
 - Promocje; Konwersje; Typowe konwersje arytmetyczne
- Rady

10.1. Wprowadzenie

W tym rozdziale znajduje się szczegółowy opis właściwości wyrażen. W języku C++ wyrażeniem jest przypisanie, wywołanie funkcji, utworzenie obiektu, jak również wiele innych operacji wykraczających daleko poza konwencjonalne obliczanie wyrażen arytmetycznych. Aby pokazać, w jaki sposób używa się wyrażen, a także przedstawić je w odpowiednim kontekście, opisałem budowę niewielkiego programu — prostego kalkulatora. Dalej przedstawiłem zestawienie operatorów oraz zwięźle opisałem sposób ich działania dla typów wbudowanych. Operatory wymagające bardziej szczegółowego omówienia są opisane w rozdziale 11.

10.2. Kalkulator

Wyobraź sobie prosty kalkulator służący do wykonywania czterech podstawowych działań arytmetycznych reprezentowanych przez operatory infiksowe działające na liczbach zmienoprzecinkowych. Na przykład dla poniższych danych:

```
r = 2.5
area = pi*r*r
```

(wartość pi jest zdefiniowana standardowo) program ten zwróci taki wynik:

```
2.5
19.635
```

2.5 to wynik dla pierwszej liniiki danych wejściowych, a 19.635 to wynik dla drugiej.

Cztery główne elementy budowy kalkulatora to: parser, funkcja przyjmowania danych, tablica symboli oraz sterownik. W istocie jest to miniaturowy kompilator, w którym parser wykonuje analizę składniową, funkcja wejściowa pobiera dane i wykonuje analizę leksykalną, tablica symboli przechowuje informacje stałe, a sterownik obsługuje inicjację, wyjście i błędy. Kalkulator ten można by było wzbogacić o wiele dodatkowych funkcji, ale jego kod i tak już jest długi, a takie dodatki w żaden sposób nie przyczyniłyby się do lepszego poznania sposobu używania języka C++.

10.2.1. Parser

Poniżej znajduje się gramatyka języka rozpoznawanego przez parser:

```
program:
    end                // end oznacza koniec danych wejściowych
    expr_list end

expr_list:
    expression print // print jest znakiem nowego wiersza lub średnikiem
    expression print expr_list

expression:
    expression + term
    expression - term
    term

term:
    term / primary
    term*primary
    primary

primar y:
    number            // number jest literalem zmiennoprzecinkowym
    name              // name jest identyfikatorem
    name = expression
    - primar y
    ( expression )
```

Innymi słowy, program jest sekwencją wyrażeń oddzielonych średnikami. Podstawowymi jednostkami wyrażenia są liczby, nazwy i operatory *, /, +, - (zarówno jedno-, jak i dwuargumentowy) oraz = (przypisanie). Nazw nie trzeba deklarować przed użyciem.

Rodzaj analizy, który stosuje, nazywa się **zstępowaniem rekurencyjnym** (ang. *recursive descent*). Jest to popularna i prosta technika przetwarzania kodu od góry do dołu. W językach programowania takich jak C++, w których wywołania funkcji są względnie mało kosztowne, metoda ta jest dodatkowo bardzo wydajna. Dla każdej produkcji w gramatyce istnieje funkcja wywołująca inne funkcje. Symbole terminalne (np. end, number, + i -) są rozpoznawane przez

analizator leksykalny, a symbole nieterminalne rozpoznają funkcję analizy leksykalnej: `expr()`, `term()` i `prim()`. Gdy oba argumenty wyrażenia lub podwyrażenia są znane, następuje obliczenie wartości. W prawdziwym kompilatorze mógłby to być moment wygenerowania kodu.

Do obsługi danych wejściowych wykorzystywana jest klasa `Token_stream` zawierająca operacje wczytywania znaków oraz składania z nich tokenów. Innymi słowy, klasa `Token_stream` zamienia strumienie znaków, takie jak `123.45`, w tokeny (`Token`). Tokeny to pary {rodzaj-tokenu, wartość}, jak np. {`number`, `123.45`}, gdzie literał `123.45` jest zamieniony na wartość zmiennoprzecinkową. Główne części parsera muszą tylko znać nazwę strumienia tokenów (`Token_stream`), `ts`, oraz móc pobrać z niego tokeny. W celu wczytania kolejnego tokenu wywoływana jest funkcja `ts.get()`. Aby pobrać ostatnio wczytany (bieżący) token, wywoływana jest funkcja `ts.current()`. Dodatkową funkcją klasy `Token_stream` jest ukrywanie prawdziwego źródła znaków. Zobaczysz, że mogą one pochodzić wprost od użytkownika wpisującego je za pomocą klawiatury do strumienia `cin`, z wiersza poleceń lub z jakiegoś innego strumienia wejściowego (10.2.7).

Definicja tokenu jest następująca:

```
enum class Kind : char {
    name, number, end,
    plus='+', minus='-', mul='*', div='/', print=';', assign='=', lp='(', rp=')'
};

struct Token {
    Kind kind;
    string string_value;
    double number_value;
};
```

Reprezentowanie każdego tokenu przez wartość całkowitoliczbową jego znaku jest wygodnym i wydajnym rozwiązaniem oraz może być pomocne dla programistów używających debugera. Metoda ta działa, pod warunkiem że żaden znak pojawiający się na wejściu nie ma wartości użytej jako enumerator — a żaden znany mi zestaw znaków nie zawiera żadnego drukowalnego znaku w postaci jednocyfrowej liczby całkowitej.

Interfejs klasy `Token_stream` wygląda tak:

```
class Token_stream {
public:
    Token get(); // wczytuje i zwraca następny token
    const Token& current(); // ostatnio wczytany token
    //...
};
```

Implementacja jest przedstawiona w sekcji 10.2.2.

Każda funkcja parsera pobiera argument typu `bool` (6.2.2), o nazwie `get`, wskazujący, czy konieczne jest wywołanie funkcji `Token_stream::get()` w celu pobrania następnego tokenu. Każda funkcja parsera oblicza wartość „swojego” wyrażenia i ją zwraca. Funkcja `expr()` obsługuje dodawanie i odejmowanie. Zawiera jedną pętlę znajdującą wyrazy do dodania lub odjęcia:

```
double expr(bool get) // dodaje i odejmuje
{
    double left = term(get);

    for (;;) { // wieczność
        switch (ts.current().kind) {
            case Kind::plus:
```

```

        left += term(true);
        break;
    case Kind::minus:
        left -= term(true);
        break;
    default:
        return left;
    }
}
}

```

Funkcja ta sama niewiele robi. Jak typowa wysokopoziomowa funkcja w każdym większym programie ogranicza się do wywoływania innych funkcji.

Instrukcja `switch` (2.2.4, 9.4.2) porównuje swój warunek, który jest wpisany w nawiasie za słowem kluczowym `switch`, ze zbiorem stałych. Instrukcje `break` służą do wychodzenia z instrukcji `switch`. Jeśli porównywana wartość nie pasuje do żadnej z klauzul `case`, wybierana jest klauzula `default`. Programista nie musi definiować tej klauzuli.

Zwróć uwagę, że wyrażenia takie jak $2-3+4$ są zgodnie z gramatyką obliczane jako $(2-3)+4$.

Ta dziwna instrukcja `for(;;)` to pętla nieskończona (9.5). Alternatywnie można też używać pętli `while(true)` w tym samym celu. Wykonanie instrukcji `switch` jest powtarzane, dopóki nie pojawi się coś innego niż `+` lub `-`, w którym to przypadku następuje przejście do klauzuli `default` i wykonanie instrukcji `return`.

Operatory `+=` i `-=` służą do obsługi dodawania i odejmowania. Zamiast nich można by było bez znaczenia dla działania programu użyć wyrażeń `left=left+term(true)` i `left=left-term(true)`. Jednak wyrażenia `left+=term(true)` i `left-=term(true)` nie dość, że są krótsze, to na dodatek bardziej bezpośrednio odzwierciedlają zamiar programisty. Każdy operator przypisania jest osobnym tokenem leksykalnym, a więc `a + = 1;` jest błędem składniowym z powodu spacji między operatorami `+` i `=`.

W języku C++ dla operatorów binarnych dostępne są operatory przypisania:

```
+ - * / % & | ^ << >>
```

Można więc używać następujących operatorów przypisania:

```
= += -= *= /= %= &= |= ^= <<= >>=
```

Operator `%` oznacza resztę z dzielenia. Operatory `&`, `|` oraz `^` to bitowe operacje logiczne i, lub oraz lub wykluczające. Operatory `<<` i `>>` to operacje przesunięcia w lewo i prawo. Zestawienie operatorów i opis ich działania znajduje się w podrozdziale 10.3. Dla binarnego operatora `@` zastosowanego do argumentów typu wbudowanego wyrażenie `x@y` oznacza `x=x@y`, z tym że wartość `x` jest obliczana tylko raz.

Funkcja `term()` obsługuje mnożenie i dzielenie w taki sam sposób jak `expr()` dodawanie i odejmowanie:

```

double term(bool get) // mnoży i dzieli
{
    double left = prim(get);

    for (;;) {
        switch (ts.current().kind) {
            case Kind::mul:
                left*= prim(true);
                break;

```

```

    case Kind::div:
        if (auto d = prim(true)) {
            left /= d;
            break;
        }
        return error("Dzielenie przez 0");
    default:
        return left;
    }
}

```

Wynik dzielenia przez zero jest niezdefiniowany i najczęściej operacja taka ma katastrofalne skutki. Dlatego przed wykonaniem dzielenia sprawdzamy, czy dzielnik nie jest zerem, i w razie potrzeby wywołujemy funkcję `error()`. Jej opis znajduje się w sekcji 10.2.4.

Zmienna `d` została wprowadzona do programu dokładnie w miejscu, w którym jest potrzebna, i od razu jest zainicjowana. Zakres dostępności nazwy wprowadzonej w warunku obejmuje instrukcję kontrolowaną przez ten warunek, a powstała wartość jest wartością tego warunku (9.4.3). W konsekwencji dzielenie i przypisanie `left/=d` są wykonywane tylko wtedy, gdy `d` nie równa się zero.

Funkcja `prim()` obsługująca *wyrażenia pierwotne* jest podobna do funkcji `expr()` i `term()`, ale w odróżnieniu od nich znajduje się na nieco niższym poziomie hierarchii wywołań i wykonuje trochę realnej pracy oraz nie zawiera pętli:

```

double prim(bool get) // obsługuje wyrażenia pierwotne
{
    if (get) ts.get(); // wczytuje następny token

    switch (ts.current().kind) {
        case Kind::number: // stała zmiennoprzecinkowa
        { double v = ts.current().number_value;
          ts.get();
          return v;
        }
        case Kind::name:
        { double& v = table[ts.current().string_value]; // znajduje odpowiednik
          if (ts.get().kind == Kind::assign) v = expr(true); // znaleziono operator =: przypisanie
          return v;
        }
        case Kind::minus: // jednoargumentowy minus
          return -prim(true);
        case Kind::lp:
        { auto e = expr(true);
          if (ts.current().kind != Kind::rp) return error("Oczekiwano ') '");
          ts.get(); // zjada ')'
          return e;
        }
        default:
          return error("Oczekiwano wyrażenia pierwotnego");
    }
}

```

Gdy zostanie znaleziony Token będący liczbą (tzn. literałem całkowitoliczbowym lub zmienno-przecinkowym), jego wartość jest umieszczana w składowej `number_value`. Analogicznie, gdy zostanie napotkany Token będący nazwą (`name`) — jakakolwiek jest jej definicja; zobacz 10.2.2 i 10.2.3 — jego wartość jest umieszczana w składowej `string_value`.

Zwróć uwagę, że funkcja `prim()` zawsze wczytuje o jeden token więcej, niż potrzebuje do analizy swojego wyrażenia pierwotnego. Powodem tego jest fakt, że *musi* to robić w niektórych przypadkach (aby np. sprawdzić, czy do nazwy jest coś przypisywane), więc dla zachowania spójności robi to zawsze. W przypadkach, w których funkcja parsera ma tylko przejść do następnego tokenu, nie musi używać wartości zwrótej funkcji `ts.get()`. Nie ma problemu, bo wynik można pobrać z `ts.current()`. Gdyby przeszkadzało mi ignorowanie wartości zwrótej funkcji `get()`, to albo dodałbym funkcję `read()`, która tylko by aktualizowała `current()` bez zwracania wartości, albo jawnie „wyrzucałbym” wynik: `void(ts.get())`.

Zanim kalkulator w jakikolwiek sposób użyje nazwy, najpierw musi sprawdzić, czy jest coś do niej przypisywane, czy też jest ona tylko odczytywana. W obu przypadkach trzeba sięgnąć do tablicy symboli. Tablica ta jest mapą (4.4.3, 31.4.3):

```
map<string,double> table;
```

To znaczy, że mapa `table` jest indeksowana typem `string`, dla którego zwracane są wartości typu `double`. Jeśli na przykład użytkownik wpisze:

```
radius = 6378.388;
```

kalkulator dojdzie do klauzuli `case Kind::name` i wykona:

```
double& v = table["radius"];
// ... expr() oblicza wartość do przypisania...
v = 6378.388;
```

Referencja `v` jest użyta jako uchwyt do wartości `double` związanej z `radius`, podczas gdy funkcja `expr()` oblicza wartość 6378.388 ze znaków wejściowych.

W rozdziałach 14. i 15. znajdują się wskazówki na temat organizacji programu jako zbioru modułów. Ale w tym kalkulatorze deklaracje można uporządkować tak, że (z jednym wyjątkiem) każda wystąpi tylko raz, przed samym jej użyciem. Wyjątkiem jest funkcja `expr()`, która wywołuje funkcję `term()`, która wywołuje funkcję `prim()`, która z kolei wywołuje funkcję `expr()`. Ten krąg wywołań trzeba jakoś rozerwać. Można na przykład umieścić deklarację:

```
double expr(bool);
```

przed definicją funkcji `prim()`.

10.2.2. Wejście

Mechanizm wczytywania danych wejściowych to często najbardziej skomplikowana część programu, bo trzeba wziąć pod uwagę humory, przyzwyczajenia i rozmaite błędy popełniane przez ludzi. Próby zmuszenia użytkownika do zachowywania się w sposób bardziej odpowiadający maszynie są zwykle (słusznie) uważane za niegrzeczne. Zadaniem niskopoziomowej procedury wejściowej jest wczytanie znaków i złożenie z nich tokenów, którymi następnie posługują się procedury działające na wyższym poziomie. W opisywanym programie niskopoziomowa procedura obsługi wejścia nazywa się `td.get()`. Pisanie takich funkcji nie jest częstym zadaniem, bo wiele systemów zawiera standardowe funkcje tego rodzaju.

Najpierw jednak musimy zobaczyć kompletny kod klasy `Token_stream`:

```

class Token_stream {
public:
    Token_stream(istream& s) : ip{&s}, owns{false} { }
    Token_stream(istream*p) : ip{p}, owns{true} { }

    ~Token_stream() { close(); }

    Token get();           // wczytuje token i go zwraca
    Token& current();     // ostatnio wczytany token

    void set_input(istream& s) { close(); ip = &s; owns=false; }
    void set_input(istream*p) { close(); ip = p; owns = true; }

private:
    void close() { if (owns) delete ip; }

    istream*ip;           // wskaźnik do strumienia wejściowego
    bool owns;           // czy Token_stream jest właścicielem strumienia istream?
    Token ct {Kind::end}; // bieżący token
};

```

Obiekt klasy `Token_stream` inicjujemy strumieniem wejściowym (4.3.2, rozdział 38.), z którego pobierane są znaki. Klasa `Token_stream` jest zaimplementowana w taki sposób, że staje się właścicielem (i ostatecznie też niszczycielem — 3.2.1.2, 11.2) strumienia `istream` przekazanego jako wskaźnik, ale nie strumienia `istream` przekazanego jako referencja. W tak prostym programie może nie trzeba stosować aż tak wyszukanego rozwiązania, ale jest to bardzo przydatna i ogólna technika wykorzystywana do budowy klas przechowujących wskaźniki do zasobów, które kiedyś trzeba usunąć.

Klasa `Token_stream` zawiera trzy wartości: wskaźnik do swojego strumienia wejściowego (`ip`), wartość logiczną (`owns`) określającą własność strumienia wejściowego oraz bieżący token (`ct`).

Zmiennej `cp` nadałem wartość domyślną, bo wydawało mi się, że niezrobienie tego byłoby nie w porządku. Wprawdzie nie powinno się wywoływać funkcji `current()` przed `get()`, ale jeśli ktoś to zrobi, to otrzyma poprawny token. Jako wartość początkową `ct` wybrałem `Kind::end`, aby w przypadku niewłaściwego użycia funkcji `current()` program nie otrzymał żadnej wartości, która nie pojawiła się w strumieniu wejściowym.

Funkcję `Token_stream::get()` przedstawię w dwóch odsłonach. Najpierw pokażę złudnie prostą wersję, która będzie mniej przyjazna dla użytkownika. A następnie zmodyfikuję ją do mniej eleganckiej, ale za to o wiele łatwiejszej w użyciu postaci. Ogólnie zadaniem funkcji `get()` jest wczytanie znaku, zdecydowanie, do jakiego rodzaju tokenu należy go użyć, oraz w razie potrzeby wczytanie większej ilości znaków i zwrócenie tokenu reprezentującego te wczytane znaki.

Początkowe instrukcje wczytują pierwszy niebiały znak z `*ip` (strumienia wejściowego wskazywanego przez `ip`) do `ch` i sprawdzają, czy operacja odczytu się powiodła:

```

Token Token_stream::get()
{
    char ch = 0;
    *ip>>ch;

    switch (ch) {
    case 0:
        return ct={Kind::end}; // przypisanie i zwrot

```

Domyślnie operator `>>` pomija białe znaki (spacje, tabulatory, znaki nowego wiersza itd.) i pozostawia wartość `ch` niezmienną, jeśli operacja przyjmowania danych się nie powiedzie. W konsekwencji `ch==0` oznacza koniec wprowadzania danych.

Przypisanie jest operatorem, a jego wynik jest wartością zmiennej, której dotyczy to przypisanie. Dzięki temu mogę przypisać wartość `Kind::end` do `curr_tok` i zwrócić ją w tej samej instrukcji. Jedna instrukcja zamiast dwóch ułatwia konserwację kodu. Gdybym rozdzielił przypisanie i zwrot, to inny programista mógłby zmienić coś w jednej części i zapomnieć odpowiednio dostosować drugą.

Zwróć też uwagę na sposób użycia notacji listowej `{}` (3.2.1.3, 11.3) po prawej stronie przypisania. Jest to wyrażenie. Powyższą instrukcję `return` można by było napisać również tak:

```
ct.kind = Kind::end; // przypisanie
return ct;          // zwrot
```

Uważam jednak, że przypisanie kompletnego obiektu `{Kind::end}` jest bardziej zrozumiałe niż posługiwanie się poszczególnymi składowymi `ct`. Zapis `{Kind::end}` jest równoważny z `{Kind::end,0,0}`. To dobrze, jeśli interesują nas dwie ostatnie składowe tokenu, i źle, jeśli zależy nam na wydajności. Nas w tym przypadku nie dotyczy ani pierwsze, ani drugie, ale ogólnie rzecz biorąc, posługiwanie się kompletnymi obiektami stwarza mniej okazji do popełnienia błędu niż grzebanie przy poszczególnych składowych. Poniżej przedstawiona jest implementacja tej drugiej strategii.

Najpierw przeanalizuj kilka z poniższych klauzul `case` osobno, a dopiero potem zastanów się nad działaniem całej funkcji. Znak oznaczający koniec wyrażenia `(;)`, nawiasy oraz operatory są obsługiwane poprzez zwykłe zwracanie ich wartości:

```
case ';': // koniec wyrażenia; drukowanie
case '*':
case '/':
case '+':
case '-':
case '(':
case ')':
case '=':
    return ct={static_cast<Kind>(ch)};
```

Operacja `static_cast` (11.5.2) jest w tym przypadku niezbędna, ponieważ nie istnieje niejawną konwersja typu `char` na `Kind` (8.4.1). Tylko niektóre znaki odpowiadają wartościom `Kind`, więc musimy zagwarantować to dla `ch`.

Liczby są obsługiwane w następujący sposób:

```
case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7':
↳ case '8': case '9': case '.':
ip->putback(ch); // wstawia pierwszą cyfrę (albo .) z powrotem do strumienia wejściowego
*ip >> ct.number_value; // wczytuje liczbę do ct
ct.kind=Kind::number;
return ct;
```

Wypisanie wszystkich klauzul `case` w jednej linii zamiast każdej w osobnym wierszu nie jest dobrym pomysłem, bo taki kod jest mniej czytelny. Z drugiej strony, wpisywanie każdej cyfry w osobnej linijce jest żmudne. Dzięki temu, że operator `>>` już obsługuje wczytywanie liczb zmiennoprzecinkowych do zmiennych typu `double`, ten kod jest bardzo prosty. Najpierw pierwszy znak (cyfra lub kropka) jest umieszczany w `cin`. Następnie wartość zmiennoprzecinkowa może zostać wczytana do `ct.number_value`.

Jeśli token nie jest końcem danych wejściowych ani operatorem, znakiem interpunkcyjnym czy też liczbą, to musi być nazwą. Nazwy są obsługiwane podobnie do liczb:

```
default: // name, name = lub błąd
  if (isalpha(ch)) {
    ip->putback(ch); // umieszcza pierwszy znak z powrotem w strumieniu wejściowym
    *ip>>ct.string_value; // wczytuje łańcuch do ct
    ct.kind=Kind::name;
    return ct;
  }
```

W końcu może też wystąpić błąd. Proste a zarazem całkiem efektywne rozwiązanie na porażenie sobie z błędami to napisanie funkcji `error()` i zwrócenie tokenu `print` w wartości zwrotnej tej funkcji:

```
error("Niepoprawny token");
return ct={Kind::print};
```

Funkcja `isalpha()` z biblioteki standardowej (36.2.1) została użyta po to, aby uniknąć konieczności wymieniania każdego znaku w osobnej klauzuli `case`. Jeśli operator `>>` zostanie zastosowany do łańcucha (w tym przypadku `string_value`), wczytuje znaki do momentu napotkania białego znaku. W związku z tym użytkownik musi po nazwie wpisać biały znak przed operatorem używającym tej nazwy jako argumentu. Jest to bardzo słabe rozwiązanie i będziemy musieli jeszcze do tego wrócić w sekcji 10.2.3.

Poniżej znajduje się ukończona funkcja wejściowa:

```
Token Token_stream::get()
{
  char ch = 0;
  *ip>>ch;

  switch (ch) {
  case 0:
    return ct={Kind::end}; // przypisanie i zwrot
  case ';': // koniec wyrażenia; drukuje
  case '*':
  case '/':
  case '+':
  case '-':
  case '(':
  case ')':
  case '=':
    return ct={static_cast<Kind>(ch)};

  case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7':
  ↪case '8': case '9': case '.':
    ip->putback(ch); // wstawia pierwszą cyfrę (albo .) z powrotem do strumienia wejściowego
    *ip >> ct.number_value; // wczytuje liczbę do ct
    ct.kind=Kind::number;
    return ct;
  default: // name, name = lub błąd
    if (isalpha(ch)) {
      ip->putback(ch); // umieszcza pierwszy znak z powrotem w strumieniu wejściowym
      *ip>>ct.string_value; // wczytuje łańcuch do ct
      ct.kind=Kind::name;
    }
```

```

        return ct;
    }

    error("Niepoprawny token");
    return ct={Kind::print};
}
}

```

Konwersja operatora na token jest bardzo prosta, bo rodzaje operatorów zostały zdefiniowane jako wartości całkowitoliczbowe (10.2.1).

10.2.3. Wejście niskopoziomowe

W kalkulatorze w obecnej postaci można znaleźć kilka niedogodności. Trzeba pamiętać o dodawaniu średnika na końcu wyrażień, aby otrzymać wynik, oraz po nazwie zawsze trzeba wpisać biały znak, co również jest uciążliwe. Na przykład $x+7$ jest tylko identyfikatorem, a nie identyfikatorem x , operatorem $=$ i liczbą 7. Aby ten zapis został zinterpretowany tak, jak byśmy tego normalnie chcieli, trzeba by było dodać biały znak za x : $x =7$. Oba problemy można rozwiązać poprzez zamianę zorientowanych na typy domyślnych operacji w funkcji `get()` na kod wczytujący pojedyncze znaki.

Najpierw zrównamy znaczenie znaku nowego wiersza ze średnikiem oznaczającym koniec wyrażenia:

```

Token Token_stream::get()
{
    char ch;

    do { // pomija białe znaki oprócz '\n'
        if (!ip->get(ch)) return ct={Kind::end};
    } while (ch!='\n' && isspace(ch));

    switch (ch) {
    case ';':
    case '\n':
        return ct={Kind::print};
    }
}

```

W kodzie tym użyłem instrukcji `do`, która różni się od pętli `while` tylko tym, że kontrolowane przez nią instrukcje są wykonywane przynajmniej raz. Wywołanie `ip->get(ch)` wczytuje jeden znak ze strumienia wejściowego `*ip` do `ch`. Domyślnie funkcja `get()` nie pomija białych znaków tak jak operator `>>`. Test `if (!ip->get(ch))` kończy się pomyślnie, gdy nie można wczytać żadnego znaku z `cin`. W tym przypadku sesję kalkulatora kończy zwrócenie wartości `Kind::end`. Operator `!` (nie) został użyty, bo funkcja `get()` zwraca `true` w przypadku powodzenia.

Do sprawdzania, czy znak jest znakiem białym, można używać funkcji `isspace()` z biblioteki standardowej (36.2.1). Wywołanie `isspace(c)` zwraca wartość niezerową, jeśli `c` jest białym znakiem, a zero w przeciwnym przypadku. Test ten jest zaimplementowany w formie przeszukiwania tablicy, dzięki czemu użycie funkcji `isspace()` jest wydajniejszym rozwiązaniem niż samodzielne sprawdzanie poszczególnych znaków. Istnieją podobne funkcje do sprawdzania, czy znak jest cyfrą (`isdigit()`), literą (`isalpha()`) oraz cyfrą lub literą (`isalnum()`).

Po pominięciu białego znaku brany jest następny znak w celu określenia, jaki pojawi się token leksykalny.

Problem powodowany przez wczytywanie przez operator `>>` znaków do łańcucha do momentu napotkania białego znaku jest rozwiązany przez wczytywanie po jednym znaku aż do znalezienia znaku nie będącego literą ani cyfrą:


```

default:          // name, name= lub błąd
  if (isalpha(ch)) {
    ct.string_value = ch;
    while (ip->get(ch))
      if (isalnum(ch))
        ct.string_value += ch;    // dołącz ch na końcu string_value
      else {
        ip->putback(ch);
        break;
      }
    ct.kind = Kind::name;
    return ct;
  }

```

Na szczęście oba udoskonalenia można zaimplementować poprzez wprowadzenie zmian w jednym lokalnym miejscu kodu. Konstruowanie programów w taki sposób, aby można je było usprawniać poprzez wprowadzanie zmian lokalnych, jest ważną częścią projektowania.

Ktoś może się obawiać, że dodawanie znaków po jednym do łańcucha to bardzo niewydajna operacja. W przypadku bardzo długich łańcuchów rzeczywiście by tak było, ale wszystkie nowoczesne implementacje typu `string` zawierają „mechanizm optymalizacji małych łańcuchów” (19.3.3). To oznacza, że obsługa łańcuchów, jakie mogą pojawić się jako nazwy w naszym kalkulatorze (a nawet w kompilatorze), nie wymaga wykonywania kosztownych operacji. W szczególności użycie krótkiego łańcucha nie wymaga korzystania z pamięci wolnej. Maksymalna liczba znaków w krótkim łańcuchu jest zależna od implementacji, ale myślę, że można powiedzieć, iż wynosi około 14.

10.2.4. Obsługa błędów

Nigdy nie należy zaniedbywać wykrywania i zgłaszania błędów, chociaż w tym programie wystarczy prosta strategia obsługi błędów. Funkcja `error()` liczy błędy, drukuje informację o błędzie i dokonuje zwrotu:

```

int no_of_errors;

double error(const string& s)
{
  no_of_errors++;
  cerr << "Błąd: " << s << '\n';
  return 1;
}

```

Strumień `cerr` to niebuforowany strumień wyjściowy służący do zgłaszania błędów (38.1).

Powodem, dla którego zwracana jest wartość, jest to, że błędy najczęściej występują w trakcie obliczania wartości wyrażeń, więc należy całkiem przerwać tę operację albo zwrócić wartość, która nie spowoduje kolejnych błędów. To drugie rozwiązanie jest właściwe w przypadku naszego kalkulatora. Gdyby funkcja `Token_stream::get()` zapamiętywała numery wierszy, funkcja `error()` mogłaby informować, gdzie w przybliżeniu wystąpił błąd. Informacja taka byłaby przydatna przy używaniu kalkulatora w nieinteraktywny sposób.

Można by było opracować bardziej elegancki i ogólny mechanizm obsługi błędów, w którym wykrywanie błędów byłoby oddzielone od procedur odzyskiwania sprawności po ich wystąpieniu. Można do tego celu użyć wyjątków (2.4.3.1, rozdział 13.), ale zastosowane obecnie rozwiązanie jest w zupełności wystarczające jak dla kalkulatora składającego się z około 180 wierszy kodu.

10.2.5. Sterownik

Mamy już praktycznie wszystkie części programu i potrzebujemy jeszcze tylko sterownika, aby go uruchomić. Zdecydowałem się użyć dwóch funkcji: `main()` do konfiguracji i zgłaszania błędów oraz `calculate()` do wykonywania samych obliczeń:

```
Token_stream ts {cin}; // użycie danych z cin

void calculate()
{
    for (;;) {
        ts.get();
        if (ts.current().kind == Kind::end) break;
        if (ts.current().kind == Kind::print) continue;
        cout << expr(false) << '\n';
    }
}

int main()
{
    table["pi"] = 3.1415926535897932385; // standardowo zdefiniowane nazwy
    table["e"] = 2.7182818284590452354;

    calculate();

    return no_of_errors;
}
```

Funkcja `main()` tradycyjnie zwraca zero, jeżeli działanie programu zakończy się w normalny sposób, a wartość różną od zera w pozostałych przypadkach (2.2.1) — dobrym pomysłem jest zwracanie liczby błędów. W tym programie jedyne inicjacje, jakie należy wykonać, to wprowadzenie standardowych nazw do tablicy symboli.

Najważniejszym zadaniem pętli głównej (w funkcji `calculate()`) jest wczytywanie wyrażeń i drukowanie wyników. Służy do tego poniższy wiersz:

```
cout << expr(false) << '\n';
```

Argument `false` informuje funkcję `expr()`, że nie trzeba wywoływać funkcji `ts.get()` w celu wczytania tokenu do obróbki.

Test dotyczący `Kind::end` gwarantuje poprawne wyjście z pętli, gdy funkcja `ts.get()` natopka błąd wejściowy albo koniec pliku. Instrukcja `break` powoduje wyjście z najbliższego otaczającego bloku `switch` lub pętli (9.5). Test dotyczący `Kind::print` (tzn. znaków `'\n'` i `';`) zwalnia funkcję `expr()` z obowiązku obsługi pustych wyrażeń. Instrukcja `continue` jest równoznaczna z przejściem na koniec pętli.

10.2.6. Nagłówki

Do budowy kalkulatora zostały użyte narzędzia z biblioteki standardowej. Aby więc działał, trzeba dołączyć odpowiednie nagłówki:

```
#include<iostream> // wejście i wyjście
#include<string>   // łańcuchy
#include<map>      // mapa
#include<cctype>   // isalpha() itp.
```

Narzędzia z tych nagłówków są dostępne w przestrzeni nazw `std`, a więc żeby posługiwać się nazwami tych narzędzi, trzeba stosować kwalifikator `std::` lub przenieść je do globalnej przestrzeni nazw:

```
using namespace std;
```

Aby nie mieszać kwestii dotyczących wyrażeń z modułowością, zdecydowałem się na drugie z wymienionych rozwiązań. W rozdziałach 14. i 15. znajduje się opis technik podziału tego programu na moduły przy użyciu przestrzeni nazw oraz jego organizacji w kilku plikach.

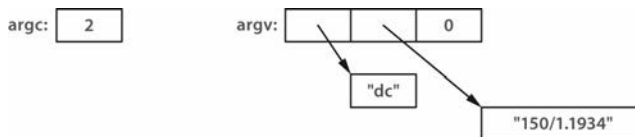
10.2.7. Argumenty wiersza poleceń

Po przetestowaniu programu zauważyłem, że jego uruchamianie, wpisywanie wyrażeń i w końcu zamykanie jest kłopotliwe. Najczęściej służył mi do sprawdzania wartości pojedynczych wyrażeń. Gdyby te wyrażenia dało się wpisywać jako argumenty wiersza poleceń, można by było zaoszczędzić trochę stukania w klawiaturę.

Program rozpoczyna działanie od wywołania funkcji `main()` (2.2.1, 15.4). Funkcji tej przekazywane są dwa argumenty, z których jeden określa liczbę argumentów (o nazwie `argc`) przekazanych przy wywołaniu, a drugi jest tablicą przechowującą te argumenty (o nazwie `argv`). Argumenty te są łańcuchami w stylu języka C (2.2.5, 7.3), a więc `argv` ma typ `char*[argc+1]`. Nazwę programu (w postaci występującej w wierszu poleceń) przekazuje się jako `argv[0]`, więc argument `argc` nigdy nie ma wartości mniejszej od zera. Lista argumentów jest zakończona zerem, tzn. `argv[argc]==0`. Na przykład dla polecenia:

```
dc 150/1.1934
```

argumenty mają następujące wartości:



Jako że metoda wywoływania funkcji `main()` jest w C++ taka sama jak w C, używane są tablice i łańcuchy w stylu języka C.

Chodzi o to, aby wczytywać dane z łańcucha wiersza poleceń w taki sam sposób jak ze strumienia wejściowego. Strumień wczytujący dane z łańcucha nazywa się `istream` (38.2.2). Aby więc obliczyć wartość wyrażeń podanych w wierszu poleceń, należy tylko sprawić, aby klasa `Token_stream` pobierała dane z odpowiedniego strumienia `istream`:

```
Token_stream ts {cin};

int main(int argc, char*argv[])
{
    switch (argc) {
        case 1: // odczyt ze standardowego strumienia wejściowego
            break;
        case 2: // odczyt z łańcucha argumentów
            ts.set_input(new istringstream{argv[1]});
            break;
        default:
            error("Zbyt wiele argumentów ");
    }
}
```

```

    return 1;
}

table["pi"] = 3.1415926535897932385; // standardowe nazwy
table["e"] = 2.7182818284590452354;

calculate();

return no_of_errors;
}

```

Aby móc korzystać ze strumienia `istringstream`, należy dołączyć do programu nagłówek `<sstream>`.

Łatwo można zmodyfikować funkcję `main()`, aby przyjmowała kilka argumentów z wiersza poleceń, ale wydaje się to niepotrzebne, zwłaszcza że przecież można przekazać kilka wyrażeń w jednym argumencie:

```
dc "rate=1.1934;150/rate;19.75/rate;217/rate"
```

Używam cudzysłowu, bo w systemach uniksowych, z których korzystam, średnik jest znakiem oddzielania poleceń. W innych systemach mogą obowiązywać inne konwencje przekazywania argumentów do uruchamianego programu.

Mimo swojej prostoty argumenty `argc` i `argv` często są źródłem drobnych, choć irytujących błędów. Aby ich uniknąć, a przede wszystkim żeby móc łatwiej posługiwać się argumentami programu, zazwyczaj używam prostej funkcji tworzącej wektor `vector<string>`:

```
vector<string> arguments(int argc, char*argv[])
{
    vector<string> res;
    for (int i = 0; i!=argc; ++i)
        res.push_back(argv[i]);
    return res;
}

```

Często spotyka się również bardziej wyszukane funkcje parsujące.

10.2.8. Uwaga na temat stylu

Dla programisty nie obeznanego z tablicami asocjacyjnymi użycie mapy ze standardowej biblioteki w roli tablicy symboli wygląda prawie jak oszustwo. Ale nim nie jest. Biblioteka standardowa i inne biblioteki są po to, by ich używać. Implementacji i projektowaniu wielu składników bibliotek poświęcono tak dużo czasu i uwagi, że żaden programista nie mógłby tak dopracować jakiegokolwiek fragmentu kodu do użytku w tylko jednym programie.

Patrząc na kod kalkulatora, zwłaszcza jego pierwszą wersję, można zauważyć, że nie ma w nim zbyt wiele tradycyjnego niskopoziomowego kodu w stylu języka C. Wiele zazwyczaj sprawiających trudności problemów rozwiązano poprzez użycie klas z biblioteki standardowej, takich jak `ostream`, `string` i `map` (4.3.1, 4.2, 4.4.3, 31.4, rozdział 36., rozdział 38.).

Zwróć uwagę na względnie niewielką ilość pętli, działań arytmetycznych i przypisań. Tak właśnie powinien wyglądać kod źródłowy programu, który nie operuje bezpośrednio na zasobach sprzętowych ani nie implementuje niskopoziomowych abstrakcji.

10.3. Zestawienie operatorów

W tym podrozdziale znajduje się zestawienie wyrażeń i trochę przykładów. Dla każdego operatora podana jest przynajmniej jedna nazwa i przykład użycia. W poniższych tabelach:

- *Nazwa* jest identyfikatorem (np. `sum`, `map`), nazwą operatora (np. `operator int`, `operator+ i operator"" km`) lub nazwą specjalizacji szablonu (np. `sort<Record> i array<int,10>`) z ewentualnym kwalifikatorem `::` (np. `std::vector` i `vector<T>::operator[]`).
- *Nazwa-klasy* to po prostu nazwa klasy (wliczając `decltype(expr)`, gdzie `expr` oznacza klasę).
- *Składowa* to nazwa składowej (wliczając nazwy destruktorów i szablony będące składowymi).
- *Obiekt* to wyrażenie zwracające obiekt klasy.
- *Wskaźnik* to wyrażenie zwracające wskaźnik (wliczając `this` i obiekt tego typu, który obsługuje tę operację wskaźnikową).
- *Wyrażenie* to wyrażenie, wliczając literał (np. `17`, `"mysz"`, `true`).
- *Lista-wyrażeń* to po prostu lista wyrażeń, która może też być pusta.
- *Lvalue* to wyrażenie oznaczające obiekt, który można modyfikować (6.4.1).
- *Typ* może być w pełni ogólną nazwą typu (z `*`, `()` itd.), tylko gdy znajduje się w nawiasie. W pozostałych przypadkach obowiązują pewne ograniczenia (iso.A).
- *Deklarator-lambda* to lista (może być pusta) parametrów oddzielanych przecinkami, po której znajduje się opcjonalny specyfikator `mutable`, po nim może znajdować się opcjonalny specyfikator `noexcept`, a po nim może się znajdować opcjonalne określenie typu zwrotnego (11.4).
- *Lista-zmiennych-lokalnych* to lista (może być pusta) określająca zależności kontekstowe (11.4).
- *Lista-instrukcji* to potencjalnie pusta lista instrukcji (2.2.4, rozdział 9.).

Składnia wyrażeń jest niezależna od typów argumentów. Sposób działania podany w tym podrozdziale dotyczy tylko sytuacji, gdy argumenty są typów wbudowanych (6.2.1). Można też definiować nowe znaczenie operatorów dla argumentów typów zdefiniowanych przez użytkownika (2.3, rozdział 18.).

W tabeli da się przedstawić gramatykę języka tylko w przybliżeniu. Szczegółowe informacje można znaleźć w iso.5 i iso.A.

Zestawienie operatorów		
Wyrażenie w nawiasie	<i>(wyrażenie)</i>	
Lambda	<i>[lista-zmiennych-lokalnych] deklarator-lambda { lista-instrukcji }</i>	11.4
Określanie zakresu	<i>nazwa-klasy :: składowa</i>	16.2.3
Określanie zakresu	<i>nazwa-przestrzeni-nazw :: składowa</i>	14.2.1
Globalny	<i>:: nazwa</i>	14.2.1
Wybór składowej	<i>obiekt . składowa</i>	16.2.3
Wybór składowej	<i>wskaźnik -> składowa</i>	16.2.3
Indeksowanie	<i>wskaźnik [wyrażenie]</i>	7.3
Wywołanie funkcji	<i>wyrażenie (lista-wyrażeń)</i>	12.2
Tworzenie wartości	<i>typ { lista-wyrażeń }</i>	11.3.2
Konwersja typów w stylu funkcyjnym	<i>typ { lista-wyrażeń }</i>	11.5.4

Zestawienie operatorów — ciąg dalszy		
Postinkrementacja	<i>lvalue++</i>	11.1.4
Postdekrementacja	<i>lvalue--</i>	11.1.4
Identyfikacja typu	<i>typeid(typ)</i>	22.5
Identyfikacja typu w czasie działania programu	<i>typeid(wyrażenie)</i>	22.5
Konwersja kontrolowana w czasie działania programu	<i>dynamic_cast< typ > (wyrażenie)</i>	22.2.1
Konwersja kontrolowana w czasie kompilacji	<i>static_cast< typ > (wyrażenie)</i>	11.5.2
Niekontrolowana konwersja	<i>reinterpret_cast< typ > (wyrażenie)</i>	11.5.2
Konwersja const	<i>const_cast < typ > (wyrażenie)</i>	11.5.2
Rozmiar obiektu	<i>sizeof wyrażenie</i>	6.2.8
Rozmiar typu	<i>sizeof (typ)</i>	6.2.8
Rozmiar pakietu parametrów	<i>sizeof... nazwa</i>	28.6.2
Układ typu	<i>alignof (typ)</i>	6.2.9
Preinkrementacja	<i>++lvalue</i>	11.1.4
Predekrementacja	<i>--lvalue</i>	11.1.4
Dopełnienie	<i>~ wyrażenie</i>	11.1.2
Nie	<i>! wyrażenie</i>	11.1.1
Jednoargumentowy minus	<i>- wyrażenie</i>	2.2.2
Jednoargumentowy plus	<i>+ wyrażenie</i>	2.2.2
Adres	<i>& lvalue</i>	7.2
Dereferencja	<i>* wyrażenie</i>	7.2
Tworzenie (alokacja)	<i>new typ</i>	11.2
Tworzenie (alokacja i inicjacja)	<i>new typ (lista-wyrażeń)</i>	11.2
Tworzenie (alokacja i inicjacja)	<i>new typ { lista-wyrażeń }</i>	11.2
Tworzenie (umieszczenie)	<i>new (lista-wyrażeń) typ</i>	11.2.4
Tworzenie (umieszczenie i inicjacja)	<i>new (lista-wyrażeń) typ (lista-wyrażeń)</i>	11.2.4
Tworzenie (umieszczenie i inicjacja)	<i>new (lista-wyrażeń) typ (lista-wyrażeń)</i>	11.2.4
Usuwanie (dealokacja)	<i>delete wskaźnik</i>	11.2
Usuwanie tablicy	<i>delete [] wskaźnik</i>	11.2.2
Czy wyrażenie może zgłosić wyjątek?	<i>noexcept (wyrażenie)</i>	13.5.1.2
Rzutowanie (konwersja typów)	<i>(typ) wyrażenie</i>	11.5.3
Wybór składowej	<i>obiekt .* wskaźnik-do-składowej</i>	20.6
Wybór składowej	<i>wskaźnik ->* wskaźnik-do-składowej</i>	20.6

Każda część tabeli zawiera operatory o tym samym priorytecie. Operatory znajdujące się wyżej mają wyższy priorytet. Na przykład $N::x.m$ oznacza $(N::x).m$, a nie $N::(x.m)$.

Na przykład przyrostkowy operator ++ ma wyższy priorytet od jednoargumentowego operatora *, więc *p++ oznacza *(p++), *nie* (*p)++.

Zestawienie operatorów (kontynuacja)		
Mnożenie	wyrażenie * wyrażenie	10.2.1
Dzielenie	wyrażenie / wyrażenie	10.2.1
Modulo (reszta z dzielenia)	wyrażenie % wyrażenie	10.2.1
Dodawanie	wyrażenie + wyrażenie	10.2.1
Odejmowanie	wyrażenie - wyrażenie	10.2.1
Przesunięcie w lewo	wyrażenie << wyrażenie	11.1.2
Przesunięcie w prawo	wyrażenie >> wyrażenie	11.1.2
Mniejszość	wyrażenie < wyrażenie	2.2.2
Mniejszy lub równy	wyrażenie <= wyrażenie	2.2.2
Większość	wyrażenie > wyrażenie	2.2.2
Większy lub równy	wyrażenie >= wyrażenie	2.2.2
Równość	wyrażenie == wyrażenie	2.2.2
Nierówny	wyrażenie != wyrażenie	2.2.2
Bitowe i	wyrażenie & wyrażenie	11.1.2
Bitowe lub wykluczające	wyrażenie ^ wyrażenie	11.1.2
Bitowe lub niewykluczające	wyrażenie wyrażenie	11.1.2
Logiczne i	wyrażenie && wyrażenie	11.1.1
Logiczne lub niewykluczające	wyrażenie wyrażenie	11.1.1
Wyrażenie warunkowe	wyrażenie ? wyrażenie : wyrażenie	11.1.3
Lista	{ lista-wyrażeń }	11.3
Zgłoszenie wyjątku	throw wyrażenie	13.5
Proste przypisanie	lvalue = wyrażenie	10.2.1
Mnożenie i przypisanie	lvalue *= wyrażenie	10.2.1
Dzielenie i przypisanie	lvalue /= wyrażenie	10.2.1
Modulo i przypisanie	lvalue %= wyrażenie	10.2.1
Dodawanie i przypisanie	lvalue += wyrażenie	10.2.1
Odejmowanie i przypisanie	lvalue -= wyrażenie	10.2.1
Przesunięcie w lewo i przypisanie	lvalue <<= wyrażenie	10.2.1
Przesunięcie w prawo i przypisanie	lvalue >>= wyrażenie	10.2.1
Bitowe i oraz przypisanie	lvalue &= wyrażenie	10.2.1
Bitowe niewykluczające lub i przypisanie	lvalue = wyrażenie	10.2.1
Bitowe lub wykluczające i przypisanie	lvalue ^= wyrażenie	10.2.1
Przecinek (do oznaczania sekwencji)	wyrażenie, wyrażenie	10.3.2

Na przykład $a+b*c$ oznacza $a+(b*c)$, a nie $(a+b)*c$, ponieważ operator $*$ ma wyższy priorytet niż $+$.

Operatory jednoargumentowe i operatory przypisania mają łączność prawostronną, a wszystkie pozostałe lewostronną. Na przykład $a=b=c$ znaczy $a=(b=c)$, podczas gdy $a+b+c$ znaczy $(a+b)+c$.

Kilku zasad gramatyki nie da się wyrazić za pomocą reguł priorytetów operatorów (zwanych także siłą wiązania) i łączności. Na przykład $a=b<c?d=e$ znaczy $a=((b<c)?(d=e):(f=g))$, ale żeby o tym wiedzieć, trzeba zapoznać się z gramatyką (iso.A).

Zanim zastosowane zostaną zasady gramatyki, najpierw ze znaków układane są tokeny leksykalne. Do utworzenia tokenu wybierana jest najdłuższa możliwa sekwencja znaków. Na przykład $\&\&$ jest pojedynczym operatorem, a nie dwoma operatorami $\&$. Podobnie $a+++1$ znaczy $(a++) + 1$. Czasami nazywa się to **regułą Maksa Muncha** (ang. *Max Munch rule*).

Zestawienie tokenów		
Klasa tokenów	Przykład	Więcej informacji
Identyfikator	vector, foo_bar, x3	6.3.3
Słowo kluczowe	int, for, virtual	6.3.3.1
Literał znakowy	'x', \n', 'U'\UFADEFADe'	6.2.3.2
Literał całkowitoliczbowy	12, 012, 0x12	6.2.4.1
Literał zmiennoprzecinkowy	1.2, 1.2e-3, 1.2L	6.2.5.1
Literał łańcuchowy	"Witaj,! ", R("świecie!")"	7.3.2
Operator	+=, %, <<	10.3
Znak interpunkcyjny	;, ,, {, }, (,)	
Notacja preprocesora	#, ##	12.6

Białe znaki (np. spacja, tabulator i nowy wiersz) mogą być separatorami tokenów (np. `int count` to słowo kluczowe i identyfikator, a nie `intcount`), ale poza tym są ignorowane.

Wpisywanie niektórych znaków z podstawowego zestawu znaków źródła (6.1.2), takich jak `|`, na niektórych klawiaturach jest uciążliwe. Ponadto dla niektórych programistów używanie symboli typu $\&\&$ czy \sim do oznaczania podstawowych operacji logicznych jest dziwne. Dlatego dostępne są też zastępcze formy w postaci słów kluczowych:

Alternatywne reprezentacje (iso.2.12)										
and	and_eq	bitand	bitor	compl	not	not_eq	or	or_eq	xor	xor_eq
$\&\&$	$\&=$	$\&$	$ $	\sim	$!$	$!=$	$ $	$ =$	\wedge	$\wedge=$

Na przykład:

```
bool b = not (x or y) and z;
int x4 = ~ (x1 bitor x2) bitand x3;
```

Powyższy kod jest równoznaczny z poniższym:

```
bool b = !(x || y) && z;
int x4 = ~(x1 | x2) & x3;
```

Zwróć uwagę, że `and=` to nie to samo co `&=`. Jeśli preferujesz słowa kluczowe, to musisz pisać `and_eq`.

10.3.1. Wyniki

Typy wyników działań arytmetycznych są określone przez zestaw reguł zwanych „typowymi konwencjami arytmetycznymi” (10.5.3). Ogólnie otrzymuje się wynik „największego” typu argumentu. Na przykład: jeżeli operatorowi binarnemu zostanie przekazany argument zmiennoprzecinkowy, to obliczenia zostaną wykonane wg zasad arytmetyki zmiennoprzecinkowej i wynik będzie typu zmiennoprzecinkowego. Analogicznie: jeżeli argument jest typu `long`, obliczenia zostaną wykonane wg zasad arytmetyki dla typu `long` i wynik będzie typu `long`. Argumenty typów mniejszych niż `int` (np. `bool` i `char`) są konwertowane na `int` przed zastosowaniem operatora.

Operatory relacyjne — `==`, `<=` itd. — zwracają wyniki logiczne. Sposób działania i typ wyniku operatorów zdefiniowanych przez użytkownika są określone przez ich deklaracje (18.2).

Gdzie jest to logicznie możliwe, wynik operatora przyjmującego jako argument wartość lewostronną jest wartością lewostronną oznaczającą ten argument lewostronny. Na przykład:

```
void f(int x, int y)
{
    int j = x = y;           // wartość x=y jest wartością x po przypisaniu
    int*p = &+x;           // p wskazuje x
    int*q = &(x++);        // błąd: x++ nie jest wartością lewostronną (nie jest wartością przechowywaną w x)
    int*p2 = &(x>y?x:y);    // adres większej wartości
    int& r = (x<y)?x:1;     // błąd: 1 nie jest wartością lewostronną
}
```

Jeśli drugi i trzeci argument operatora `?:` są wartościami lewostronnymi i mają ten sam typ, to wynik jest tego typu i jest wartością lewostronną. Taki sposób zachowywania wartości lewostronnych sprawia, że można bardziej elastycznie posługiwać się operatorami. Jest to szczególnie przydatne przy pisaniu kodu, który musi być tak samo wydajny niezależnie od tego, czy działa na typach wbudowanych, czy zdefiniowanych przez użytkownika (np. przy pisaniu szablonów albo programów generujących kod C++).

Wynik działania operatora `sizeof` jest typu całkowitoliczbowego bez znaku o nazwie `size_t` zdefiniowanego w nagłówku `<cstdlib>`. Wynik odejmowania wskaźników jest typu całkowitoliczbowego o nazwie `ptrdiff_t` zdefiniowanego w nagłówku `<cstdlib>`.

Implementacje nie muszą pilnować przepełnienia arytmetycznego i rzadko to robią. Na przykład:

```
void f()
{
    int i = 1;
    while (0 < i) ++i;
    cout << "i ma teraz wartość ujemną!" << i << '\n';
}
```

Ten kod w końcu dojdzie do momentu, w którym zwiększy wartość `i` poza zakres typu `int`. Nie jest określone, co się w takiej sytuacji zdarzy, ale najczęściej następuje „zawinięcie” wartości do liczby ujemnej (tu mnie np. jest to `-2147483648`). Także wynik dzielenia przez zero jest niezdefiniowany, ale działanie takie najczęściej powoduje nagle przerwanie pracy programu. W szczególności niedopełnienie, przepełnienie i dzielenie przez zero nie powodują zgłoszenia standardowych wyjątków (30.4.1.1).

10.3.2. Kolejność wykonywania działań

Kolejność wykonywania podwyrażeń w wyrażeniach jest niezdefiniowana. W szczególności nie można zakładać, że wyrażenie zostanie wykonane od lewej. Na przykład:

```
int x = f(2)+g(3); // nie wiadomo, czy najpierw zostanie wywołana funkcja f(), czy g()
```

Brak ograniczeń dotyczących kolejności wykonywania działań pozwala na generowanie lepszego kodu. Jednakże może on prowadzić także do powstawania niezdefiniowanych wyników. Na przykład:

```
int i = 1;
v[i] = i++; // wynik niezdefiniowany
```

Przypisanie w tym kodzie może zostać zinterpretowane jako $v[1]=1$ lub $v[2]=2$ albo spowodować jakieś dziwne operacje. Kompilator może ostrzegać o takich niejednoznacznościach. Niestety większość kompilatorów tego nie robi, więc uważaj, by nie pisać wyrażień odczytujących lub zapisujących obiekt więcej niż raz, chyba że przy użyciu jednego operatora nie pozostawiającego wątpliwości, np. $++$ albo $+=$, lub bezpośrednio określając sekwencję przy użyciu operatora $,$ (przecinek), $\&\&$ lub $||$.

Operatory $,$ (przecinek), $\&\&$ (logiczne i) i $||$ (logiczne lub) gwarantują, że najpierw zostanie obliczona wartość lewego argumentu, a potem prawego. Na przykład $b=(a=2,a+1)$ przypisuje wartość 3 do b. Przykłady użycia operatorów $\&\&$ i $||$ znajdują się w sekcji 10.3.3. Dla typów wbudowanych wartość drugiego argumentu operatora $\&\&$ jest sprawdzana tylko wtedy, gdy pierwszy argument ma wartość true. Podobnie w przypadku operatora $||$ wartość drugiego argumentu jest sprawdzana, gdy pierwszy ma wartość false. Technikę tę nazywa się czasami **skrótowym określaniem wartości** (ang. *short-circuit evaluation*). Zwróć uwagę, że operator sekwencji (przecinek) różni się pod względem logicznym od przecinka oddzielającego argumenty wywołania funkcji. Na przykład:

```
f1(v[i], i++); // dwa argumenty
f2((v[i], i++)); // jeden argument
```

Funkcja $f1()$ jest wywoływana z dwoma argumentami, $v[i]$ oraz $i++$, a kolejność wykonywania wyrażień w argumentach jest niezdefiniowana, więc należy tego unikać. Zależność od kolejności wyrażień w argumentach wywołania funkcji jest oznaką bardzo słabego stylu programowania, w którym nie da się przewidzieć zachowania programu. Wywołanie funkcji $f2()$ ma tylko jeden argument, wyrażenie z przecinkiem $(v[i], i++)$, które jest równoważne z $i++$. Jest to niejasne i tego też należy się wystrzeżać.

Do grupowania wyrażień można używać nawiasów. Na przykład $a*b/c$ oznacza $(a*b)/c$, więc jeśli chodzi nam o $a*(b/c)$, musimy użyć nawiasu. Wyrażenie $a*(b/c)$ może zostać wykonane $(a*b)/c$, tylko gdy zmiana ta nie ma znaczenia dla użytkownika. Różnice między $a*(b/c)$ i $(a*b)/c$ są szczególnie wyraźne w przypadku działań na liczbach zmiennoprzecinkowych i dlatego kompilator takie wyrażenia wykonuje dokładnie tak, jak są zapisane.

10.3.3. Priorytety operatorów

Zasady dotyczące priorytetów i łączności operatorów są zgodne z praktyką. Na przykład:

```
if (i<=0 || max<i) //...
```

znaczy „jeśli i jest mniejsze lub równe 0 lub max jest mniejsze od i”. Jest to równoważne z:

```
if ( (i<=0) || (max<i) ) //...
```

Natomiast poniższy kod jest poprawny, ale bezsensowny:

```
if (i <= (0 || max) < i) //...
```

Jeśli są jakiegokolwiek wątpliwości co do kolejności wykonywania działań, należy używać nawiasów. Im bardziej skomplikowane wyrażenie, tym więcej nawiasów można w nim znaleźć, a mimo to skomplikowane wyrażenia są częstym źródłem rozmaitych błędów. Dlatego jeżeli czujesz potrzebę użycia nawiasu, to zastanów się, czy nie lepiej podzielić wyrażenie na części, wprowadzając dodatkową zmienną.

Czasami kolejność wykonywania operatorów nie jest oczywista. Na przykład:

```
if (i&mask == 0) // ups! wyrażenie == jako argument operatora &
```

To wyrażenie nie oznacza, że maska zostanie zastosowana do *i*, a potem nastąpi sprawdzenie, czy wynik jest równy 0. Operator `==` ma wyższy priorytet od `&`, więc wyrażenie zostanie zinterpretowane jako `i&(mask==0)`. Na szczęście kompilator może łatwo wykryć takie błędy i ostrzec o nich użytkownika. W tym przypadku nawiasy są bardzo ważne:

```
if ((i&mask) == 0) //...
```

Należy zauważyć, że poniższe wyrażenie nie zostanie zinterpretowane w taki sposób, jakiego oczekiwałby matematyk:

```
if (0 <= x <= 99) //...
```

Ten poprawny kod zostanie zinterpretowany jako `(0<=x)<=99`, gdzie wynik pierwszego porównywania to `true` albo `false`. Ta logiczna wartość jest następnie niejawnie konwertowana na 1 lub 0 i porównywana z 99, wynikiem czego jest `true`. Aby sprawdzić, czy *x* mieści się w przedziale od 0 do 99, można napisać:

```
if (0<=x && x<=99) //...
```

Początkujący programiści często przypadkowo używają operatora przypisania (`=`) zamiast równości (`==`):

```
if (a = 7) // ups! przypisanie stałe w warunku
```

Jest to naturalne, bo w wielu językach znak `=` oznacza „równa się”. Dla kompilatora wykrycie takich błędów jest łatwe i wiele kompilatorów ostrzeże o ich występowaniu. Nie zalecam udziwniania stylu programowania tylko po to, by zrekompensować niedobory kompilatorów w zakresie ostrzegania. W szczególności nie należy robić czegoś takiego:

```
if (7 == a) // próba ochrony przed niepoprawnym użyciem operatora =; niezalecane
```

10.3.4. Obiekty tymczasowe

Kompilator często musi tworzyć obiekty tymczasowe do przechowywania pośrednich wyników wyrażień. Na przykład dla wyrażenia `v=x+y*z` trzeba gdzieś przechować wynik podwyrażenia `y*z` przed dodaniem go do *x*. W przypadku typów wbudowanych wszystko dzieje się w taki sposób, że **obiekt tymczasowy** jest niewidoczny dla użytkownika. Jednak w przypadku typów zdefiniowanych przez użytkownika zawierających zasoby wiedza o cyklu istnienia obiektu tymczasowego może być dla programisty ważna. Obiekt tymczasowy, jeśli nie jest związany z referencją ani wykorzystywany do inicjacji obiektu mającego nazwę, jest usuwany na końcu pełnego wyrażenia, dla którego został utworzony. **Pełne wyrażenie** to wyrażenie nie będące częścią innego wyrażenia.

Typ `string` z biblioteki standardowej zawiera składową `c_str()` (36.3) zwracającą wskaźnik w stylu języka C do zakończonej zerem tablicy znaków (2.2.5, 43.4). Ponadto operator `+` dla tego typu oznacza łączenie łańcuchów. Jeśli te bardzo przydatne do pracy z łańcuchami narzędzia zostaną użyte razem, mogą powodować różne trudne do określenia błędy. Na przykład:

```
void f(string& s1, string& s2, string& s3)
{
    const char*cs = (s1+s2).c_str();
    cout << cs;
    if (strlen(cs=(s2+s3).c_str())<8 && cs[0]=='a') {
        // użycie cs
    }
}
```

Pewnie w pierwszej chwili wykrzyknąłeś: „Ale tak się nie robi!”. Zgoda. Niemniej jednak ludzie piszą taki kod, więc warto wiedzieć, jak go interpretować.

Do przechowywania wartości wyrażenia `s1+s2` zostanie utworzony obiekt tymczasowy. Następnie z obiektu tego zostanie pobrany wskaźnik do łańcucha w stylu C. Później — na końcu wyrażenia — obiekt tymczasowy zostanie usunięty. A przecież łańcuch zwrócony przez funkcję `c_str()` został alokowany jako część obiektu tymczasowego przechowującego wynik podwyrażenia `s1+s2` i nie ma gwarancji, że ta pamięć będzie nadal istnieć po usunięciu tego obiektu tymczasowego. W konsekwencji `cs` wskazuje dealokowaną pamięć. Operacja wyjściowa `cout<<cs` może zadziałać zgodnie z oczekiwaniami, ale byłoby to czyste szczęście. Kompilator może wykryć wiele rodzajów takich problemów.

Problem z instrukcją `if` jest nieco bardziej subtelny. Warunek zadziała tak jak powinien, bo pełne wyrażenie, w którym tworzony jest obiekt tymczasowy przechowujący wynik `s2+s3`, samo jest tym warunkiem. Jednak obiekt ten zostanie usunięty przed rozpoczęciem wykonywania instrukcji kontrolowanej, przez co jakiegokolwiek posługiwanie się zmienną `cs` w tym kodzie jest ryzykowne.

Należy zauważyć, że zarówno w tym przypadku, jak i wielu innych problemy z obiektami tymczasowymi są spowodowane użyciem wysokopoziomowego typu danych w niskopoziomowy sposób. Gdyby zastosowano prostszy styl programowania, powstałby bardziej zrozumiały kod i problemu by nie było. Na przykład:

```
void f(string& s1, string& s2, string& s3)
{
    cout << s1+s2;
    string s = s2+s3;
    if (s.length()<8 && s[0]=='a') {
        // użycie s
    }
}
```

Obiektu tymczasowego można użyć jako inicjatora dla referencji stałej lub obiektu nazwanego. Na przykład:

```
void g(const string&, const string&);

void h(string& s1, string& s2)
{
    const string& s = s1+s2;
    string ss = s1+s2;
    g(s,ss); // tu można używać s i ss
}
```

Ten kod jest w porządku. Obiekt tymczasowy jest usuwany w miejscu zakończenia zakresu „jego” referencji lub obiektu nazwanego. Pamiętaj, że zwrot referencji do zmiennej lokalnej jest błędem (12.1.4) oraz że obiektu tymczasowego nie można związać z niestałą referencją lewostronną (7.7).

Obiekt tymczasowy można też utworzyć w wyrażeniu bezpośrednio poprzez wywołanie konstruktora (11.5.1). Na przykład:

```
void f(Shape& s, int n, char ch)
{
    s.move(string{n,ch}); // utworzenie łańcucha z n kopii ch do przekazania do Shape::move()
    //...
}
```

Takie obiekty tymczasowe są usuwane wg tych samych zasad co obiekty tworzone niejawnie.

10.4. Wyrażenia stałe

W języku C++ funkcjonują dwa pokrewne pojęcia „stałej”:

- `constexpr` — wartość wyznaczana w czasie kompilacji (2.2.3);
- `const` — oznacza brak możliwości modyfikacji obiektu w określonym zakresie (2.2.3, 7.5).

Najkrócej mówiąc, słowo kluczowe `constexpr` ma umożliwić gwarancję obliczania wartości podczas kompilacji, a `const` służy do zapewniania niezmienności interfejsów. Głównym tematem tego podrozdziału jest pierwsza rola: obliczanie wartości w czasie kompilacji.

Wyrażenie stałe to takie, którego wartość może obliczyć kompilator. Nie może zawierać wielkości nieznanych w czasie kompilacji ani powodujących efekty uboczne. W związku z tym na początku wyrażenia stałego musi znajdować się wartość całkowitoliczbowa (6.2.1), zmienno-przecinkowa (6.2.5) lub enumerator (8.4). Można dodatkowo używać operatorów i funkcji `constexpr`, które tworzą wartości. Ponadto w niektórych formach wyrażen stałych można używać adresów. Dla uproszczenia kwestię tę opisałem osobno w sekcji 10.4.5.

Jest wiele powodów, dla których należy użyć nazwanej stałej zamiast literału lub zmiennej:

1. Dzięki nazwanym stałym kod jest łatwiej zrozumieć i utrzymywać.
2. Zmienna może się zmienić (co zmusza programistę do bardziej ostrożnego postępowania się niż w przypadku stałej).
3. Zasady języka wymagają używania wyrażen stałych do określania rozmiarów tablic, w etykietach `case` oraz jako argumentów wartości szablonów.
4. Programiści układów wbudowanych lubią zapisywać niezmiennie dane w pamięci tylko do odczytu, bo jest ona tańsza niż pamięć dynamiczna (przy uwzględnieniu ceny i zużycia energii) i często jest jej więcej. Ponadto dane znajdujące się w pamięci tylko do odczytu są odporne na większość awarii systemowych.
5. Jeśli obiekt jest inicjowany w czasie kompilacji, to nie ma mowy o wyścigu do niego w systemach wielowątkowych.
6. Czasami wykonanie jakichś obliczeń raz w czasie kompilacji jest o wiele lepszym rozwiązaniem pod względem wydajności niż powtarzanie ich miliony razy w czasie działania programu.

Powody [1], [2], [5] oraz częściowo [4] są logiczne. Nie używa się wyrażeń stałych tylko z powodu obsesyjnej dbałości o wydajność. Często jest tak, że wyrażenie stałe po prostu bardziej bezpośrednio reprezentuje wymogi systemu.

Wyrażenie stałe użyte jako część definicji elementu danych (celowo unikam tu słowa „zmienna”) wyraża konieczność obliczenia wartości w czasie kompilacji. Jeżeli wartości inicjatora wyrażenia stałego nie można określić w czasie kompilacji, kompilator zgłosi błąd. Na przykład:

```
int x1 = 7;
constexpr int x2 = 7;

constexpr int x3 = x1;    // błąd: inicjator nie jest wyrażeniem stałym
constexpr int x4 = x2;    // OK

void f()
{
    constexpr int y3 = x1; // błąd: inicjator nie jest wyrażeniem stałym
    constexpr int y4 = x2; // OK
    //...
}
```

Sprytny kompilator powinien wydedukować, że wartość `x1` w inicjatorze `x3` wynosi 7. Jednak lepiej jest nie polegać na sprycie kompilatora. W dużych programach określenie wartości zmiennych w czasie kompilacji jest zazwyczaj bardzo trudne albo niemożliwe.

Wielką zaletą wyrażeń stałych jest ich ekspresja. Można w nich używać wartości całkowitych, zmiennoprzecinkowych oraz wyliczeniowych. Można używać wszystkich operatorów, które nie zmieniają stanu (np. `+`, `?`, `i []`, ale nie `=` ani `++`). Używając funkcji `constexpr` (12.1.6) i typów literałowych (10.4.3), można zapewnić wysoki poziom bezpieczeństwa pod względem typów. Jest wręcz nadużyciem porównywanie tych możliwości z tym, co zwykle robi się przy użyciu makr (12.6).

Do wyboru możliwości w wyrażeniach stałych służy operator wyrażenia warunkowego `?:`. Na przykład można obliczyć pierwiastek kwadratowy liczby całkowitej w czasie kompilacji:

```
constexpr int isqrt_helper(int sq, int d, int a)
{
    return sq <= a ? isqrt_helper(sq+d,d+2,a) : d;
}

constexpr int isqrt(int x)
{
    return isqrt_helper(1,3,x)/2 - 1;
}

constexpr int s1 = isqrt(9); // s1 ma wartość 3
constexpr int s2 = isqrt(1234);
```

Najpierw sprawdzana jest wartość warunku operatora `?:`, a następnie zostaje wykonane wybrane wyrażenie. Niewybrane wyrażenie nie jest wykonywane i może nawet nie być stałe. Analogicznie argumenty operatorów `&&` i `||`, które nie są obliczane, również nie muszą być wyrażeniami stałymi. Jest to najbardziej przydatne w funkcjach `constexpr`, które czasami są używane jako wyrażenia stałe, a czasami nie.

10.4.1. Stałe symboliczne

Najważniejszym zastosowaniem stałych (`constexpr` i `const`) jest tworzenie symbolicznych nazw dla wartości. Nazw takich należy używać po to, by uniknąć tzw. magicznych liczb. Literały rozsiane po całym kodzie programu są jednym z najgorszych koszmarów programisty zajmującego się konserwacją kodu. Jeśli jakaś numeryczna stała, jak np. określenie granicy tablicy, jest powtórzona w kodzie, to utrudnia poprawianie tego kodu, bo trzeba znaleźć jej wszystkie wystąpienia i je zmienić. Dzięki użyciu stałej można wszystkie informacje mieć w jednym miejscu. Stałe liczbowe często reprezentują jakieś założenia o programie. Na przykład liczba 4 może reprezentować liczbę bajtów w typie całkowitoliczbowym, 128 może być liczbą znaków mieszczących się w buforze, a 6.24 może oznaczać kurs wymiany walut między duńską koroną a amerykańskim dolarem. Jeśli takie wartości zostałyby umieszczone w kodzie, to osoba go czytająca mogłaby się nie domyślić, co oznaczają. Ponadto wiele tego typu wartości z czasem się zmienia. Łatwo je przeoczyć i problem gotowy, gdy program przeniesie się na inną platformę albo zmieni się coś, od czego one zależą. Najłatwiej tego typu trudności uniknąć poprzez reprezentowanie założeń jako stałych symbolicznych opatrzonych dobrym komentarzem.

10.4.2. `const` w wyrażeniach stałych

Słowa kluczowego `const` najczęściej używa się do budowy interfejsów (7.5), ale to nie znaczy, że nie może ono zostać użyte do wyrażania wartości stałych. Na przykład:

```
const int x = 7;
const string s = "asdf";
const int y = sqrt(x);
```

Stała `const` zainicjowana wyrażeniem stałym może być używana w wyrażeniach stałych. Różnica między `const` a `constexpr` jest taka, że stałą `const` można zainicjować czymś, co nie jest wyrażeniem stałym. W takim przypadku stałej `const` nie można używać jako wyrażenia stałego. Na przykład:

```
constexpr int xx = x;    // OK
constexpr string ss = s; // błąd: s nie jest wyrażeniem stałym
constexpr int yy = y;   // błąd: sqrt(x) nie jest wyrażeniem stałym
```

Błędy w dwóch ostatnich wierszach są spowodowane tym, że `string` nie jest typem literalowym (10.4.3), a `sqrt()` nie jest funkcją `constexpr` (12.1.6).

Często przy definiowaniu prostych stałych lepszym wyborem jest użycie `constexpr`, ale słowo to jest dostępne dopiero od C++11, więc w starszym kodzie nie można go spotkać. W wielu przypadkach dobrym zastępnikiem dla stałych `const` są enumeratory (8.4).

10.4.3. Typy literalowe

W wyrażeniu stałym można użyć prostego typu zdefiniowanego przez użytkownika. Na przykład:

```
struct Point {
    int x,y,z;
    constexpr Point up(int d) { return {x,y,z+d}; }
    constexpr Point move(int dx, int dy) { return {x+dx,y+dy}; }
    // ...
};
```

Klasa zawierająca konstruktor constexpr nazywa się **typem literałowym** (ang. *literal type*). Aby spełniać wszystkie wymagania stawiane wyrażeniom stałym, konstruktor musi być pusty, a wszystkie składowe muszą być inicjowane potencjalnie stałymi wyrażeniami. Na przykład:

```
constexpr Point origo {0,0};
constexpr int z = origo.x;

constexpr Point a[] = {
    origo, Point{1,1}, Point{2,2}, origo.move(3,3)
};
constexpr int x = a[1].x;    // x ma wartość 1

constexpr Point xy{0,sqrt(2)}; // błąd: sqrt(2) nie jest wyrażeniem stałym
```

Podkreślę, że można też tworzyć tablice constexpr oraz uzyskiwać dostęp do elementów tablic i składowych obiektów.

Oczywiście można definiować funkcje constexpr pobierające argumenty typów literałowych. Na przykład:

```
constexpr int square(int x)
{
    return x*x;
}

constexpr int radial_distance(Point p)
{
    return isqrt(square(p.x)+square(p.y)+square(p.z));
}

constexpr Point p1 {10,20,30}; // domyślny konstruktor jest constexpr
constexpr p2 {p1.up(20)};     // Point::up() jest constexpr
constexpr int dist = radial_distance(p2);
```

Użyłem typu int zamiast double, bo nie miałem pod ręką funkcji constexpr do obliczania pierwiastka kwadratowego z liczby zmiennoprzecinkowej.

Dzięki temu, że funkcja składowa zdefiniowana jako constexpr implikuje const, nie musiałem pisać czegoś takiego:

```
constexpr Point move(int dx, int dy) const { return {x+dx,y+dy}; }
```

10.4.4. Argumenty referencyjne

Przy używaniu stałych constexpr należy pamiętać, że w wyrażeniach tych chodzi głównie o wartości. Nie ma tu obiektów, które mogą zmienić wartość albo mieć efekty uboczne: constexpr to coś w rodzaju miniaturowego funkcyjnego języka programowania interpretowanego w czasie kompilacji. Pewnie zgadujesz, że w wyrażeniach stałych nie można używać referencji. To prawda, ale tylko do pewnego stopnia, bo referencje const odnoszą się do wartości i mogą być używane. Weźmy na przykład specjalizację ogólnego typu `complex<T>` do `complex<double>` z biblioteki standardowej:

```
template<> class complex<double> {
public:
    constexpr complex(double re = 0.0, double im = 0.0);
    constexpr complex(const complex<float>&);
```



```

explicit constexpr complex(const complex<long double>&);

constexpr double real(); // odczytuje część rzeczywistą
void real(double);      // ustawia część rzeczywistą
constexpr double imag(); // odczytuje część urojoną
void imag(double);      // ustawia część urojoną

complex<double>& operator= (double);
complex<double>& operator+=(double);
// ...
};

```

Oczywiście operacje modyfikujące obiekty, takie jak `=` i `+=`, nie mogą być wyrażeniami stałymi. Natomiast operacje tylko odczytujące obiekty, takie jak `real()` i `imag()`, mogą być `constexpr` i mogą być obliczane w czasie kompilacji, jeśli jest podane wyrażenie stałe. Interesującą składową jest konstruktor szablonu z innego typu `complex`:

```

constexpr complex<float> z1 {1,2}; // uwaga: <float> nie <double>
constexpr double re = z1.real();
constexpr double im = z1.imag();
constexpr complex<double> z2 {re,im}; // z2 jest kopią z1
constexpr complex<double> z3 {z1}; // z3 jest kopią z1

```

Ten konstruktor kopiujący działa, bo kompilator rozpoznaje, że referencja (`const complex<float>&`) odnosi się do wartości stałej, i używamy po prostu tej wartości (zamiast próbować wymyślać jakieś skomplikowane lub niemądre rozwiązania oparte na referencjach i wskaźnikach).

Dzięki typom literałowym możliwe jest używanie typów do wykonywania obliczeń w czasie kompilacji. Kiedyś obliczenia czasu kompilacji w języku C++ były ograniczone do używania wartości całkowitoliczbowych (i bez funkcji). Przymus kodowania wszystkich informacji w liczbach całkowitych sprawiał, że trzeba było pisać nadmiernie skomplikowany kod, w którym łatwo było się pomylić. Przykładami tego są niektóre zastosowania metaprogramowania szablonów (rozdział 28). Niektórzy programiści po prostu woleli korzystać z obliczeń w czasie działania programu, zamiast walczyć z ograniczeniami języka.

10.4.5. Wyrażenia stałe adresowe

Adres statycznie alokowanego obiektu (6.4.2), np. zmiennej globalnej, jest stałą. Jednakże jego wartość jest przypisywana przez konsolidator, a nie kompilator, więc kompilator nie może znać wartości takiej stałej adresowej. To ogranicza wachlarz zastosowań wyrażen stałych typów wskaźnikowych i referencyjnych. Na przykład:

```

constexpr const char*p1 = "asdf";
constexpr const char*p2 = p1; // OK
constexpr const char*p2 = p1+2; // błąd: kompilator nie zna wartości p1
constexpr char c = p1[2]; // OK, c='d'. Kompilator zna wartość wskazywaną przez p1

```

10.5. Niejawna konwersja typów

Typy całkowitoliczbowe i zmiennoprzecinkowe (6.2.1) można dowolnie mieszać w przypisaniach i wyrażeniach. Gdy to możliwe, ich wartości są konwertowane tak, aby nie nastąpiła utrata informacji. Niestety także pewne zawężające (powodujące utratę części informacji)

konwersje są wykonywane w sposób niejawni. Konwersja nie powoduje utraty informacji, jeżeli wartość przekonwertuje się na inny typ, a potem z powrotem na poprzedni i w wyniku tych działań otrzyma się identyczną wartość jak na początku. Jeśli te warunki nie są spełnione, to mamy do czynienia z **konwersją zawężającą** (10.5.2.6). W tej sekcji znajduje się opis zasad konwersji oraz związanych z nimi problemów i metod ich rozwiązywania.

10.5.1. Promocje

Niejawne konwersje zachowujące wartości nazywają się **promocjami**. Zanim zostanie wykonane działanie arytmetyczne, stosowana jest **promocja całkowitoliczbowa** w celu zamiany na int krótszych typów całkowitych. Analogicznie typ float jest zamieniany na double w wyniku **promocji zmiennoprzecinkowej**. Należy podkreślić, że w wyniku tych promocji *nie* następuje zamiana na typ long (chyba że argument jest typu char16_t, char32_t, wchar_t lub jest wyliczeniem przekraczającym rozmiar typu int) ani long double. Stanowi to odzwierciedlenie pierwotnego przeznaczenia promocji jeszcze w języku C, w którym miały za zadanie sprowadzać argumenty do „naturalnego” rozmiaru dla działań arytmetycznych.

Promocje całkowitoliczbowe to:

- Typy char, signed char, unsigned char, short int oraz unsigned short int są zamieniane na int, jeśli typ ten może zostać użyty do reprezentowania wszystkich wartości typu źródłowego. W przeciwnym razie następuje konwersja na typ unsigned int.
- Typy char16_t, char32_t, wchar_t (6.2.3) oraz zwykle wyliczenia (8.4.2) są konwertowane na pierwszy z następujących typów, który może zostać użyty do reprezentowania ich wartości: int, unsigned int, long, unsigned long, unsigned long long.
- Pola bitowe (8.2.7) są konwertowane na typ int, jeżeli typ ten może reprezentować wszystkie wartości tego pola. W przeciwnym razie następuje konwersja na typ unsigned int, jeśli jest on odpowiedni. Jeżeli nie, to promocja całkowitoliczbowa nie jest wykonywana.
- Typ bool jest konwertowany na int. Wartość false zamienia się na 0, a true na 1.

Promocje są używane w ramach zwykłych konwersji arytmetycznych (10.5.3).

10.5.2. Konwersje

Typy podstawowe mogą być konwertowane niejawnie na wiele sposobów (iso.4). Uważam, że zezwala się na zbyt wiele konwersji. Na przykład:

```
void f(double d)
{
    char c = d; // uwaga: konwersja double na char
}
```

Pisząc kod, należy unikać wszelkich niezdefiniowanych zachowań i konwersji powodujących niepostrzeżoną utratę części informacji (zawężających).

Na szczęście o wielu budzących wątpliwości konwersjach może nas ostrzec kompilator. Konwersje zawężające eliminuje składnia inicjacji z użyciem {} (6.3.5). Na przykład:

```
void f(double d)
{
    char c {d}; // błąd: konwersja double na char
}
```

Jeśli nie da się uniknąć potencjalnie zawężającej konwersji, należy rozważyć możliwość użycia konwersji kontrolowanej w czasie działania programu, np. `narrow_cast<>` (11.5).

10.5.2.1. Konwersje całkowitoliczbowe

Liczba typu całkowitego może zostać przekonwertowana na inny typ całkowity. Także wartość wyliczeniową (zwykle wyliczenie) można przekonwertować na typ całkowitoliczbowy (8.4.2).

Jeżeli typ docelowy jest bez znaku (`unsigned`), to wynik konwersji zawiera tyle bitów z wartości źródłowej, ile się zmieści w wartości docelowej (w razie potrzeby usuwane są bardziej znaczące bity). Mówiąc dokładniej, wynik jest najmniejszą liczbą całkowitą bez znaku przystającą do źródłowej liczby całkowitej modulo 2 do n , gdzie n jest liczbą bitów użytych do reprezentacji tego typu bez znaku. Na przykład:

```
unsigned char uc = 1023; // binarna liczba 111111111: uc ma wartość 11111111, czyli 255
```

Jeśli typ docelowy ma znak, to wartość pozostaje bez zmian, jeżeli może być reprezentowana przez ten typ. W przeciwnym przypadku wszystko zależy od implementacji:

```
signed char sc = 1023; // zależy od implementacji
```

Możliwe wyniki to 127 i -1 (6.2.3).

Wartość logiczna i wyliczeniowa (zwykle wyliczenie) może być niejawnie przekonwertowana na swój odpowiednik całkowitoliczbowy (6.2.2, 8.4).

10.5.2.2. Konwersje zmiennoprzecinkowe

Wartość zmiennoprzecinkową można przekonwertować na inny typ zmiennoprzecinkowy. Jeżeli wartość źródłową można dokładnie przedstawić w typie docelowym, to wynikiem jest oryginalna wartość liczbowa. Jeśli wartość źródłowa mieści się między dwiema przystającymi wartościami docelowymi, to wynik jest jedną z nich. W pozostałych przypadkach wynik jest niezdefiniowany. Na przykład:

```
float f = FLT_MAX; // największa wartość typu float
double d = f;      // OK: d == f

double d2 = DBL_MAX; // największa wartość typu double
float f2 = d2;       // niezdefiniowane, jeśli FLT_MAX < DBL_MAX

long double ld = d2; // OK: ld = d2
long double ld2 = numeric_limits<long double>::max();
double d3 = ld2;     // niezdefiniowane, jeśli sizeof(long double) > sizeof(double)
```

Definicje `DBL_MAX` i `FLT_MAX` znajdują się w nagłówku `<climits>`. Definicja `numeric_limits` znajduje się w nagłówku `<limits>` (40.2).

10.5.2.3. Konwersje wskaźników i referencji

Każdy wskaźnik na typ obiektowy może zostać niejawnie przekonwertowany na `void*` (7.2.1). Wskaźnik (referencja) do klasy pochodnej może być niejawnie przekonwertowany na wskaźnik (referencję) do dostępnej i jednoznacznej klasy bazowej (20.2). Wskaźnik do funkcji lub składowej nie może zostać niejawnie przekonwertowany na `void*`.

Wyrażenie stałe (10.4) o wartości 0 może zostać niejawnie przekonwertowane na wskaźnik pustego dowolnego typu wskaźnikowego. Podobnie wyrażenie stałe o wartości 0 może zostać niejawnie przekonwertowane na typ wskaźnika do składowej (20.6). Na przykład:

```
int*p = (1+2)*(2*(1-1)); // w porządku, chociaż dziwne
```

Lepiej użyć `nullptr` (7.2.2).

Wskaźnik `T*` można niejawnie przekonwertować na `const T*` (7.5). Podobnie referencję `T&` można niejawnie przekonwertować na `const T&`.

10.5.2.4. Konwersje wskaźników do składowych

Wskaźniki i referencje do składowych mogą być niejawnie konwertowane, a opis tych konwersji znajduje się w sekcji 20.6.3.

10.5.2.5. Konwersje wartości logicznych

Wartości wskaźnikowe, całkowitoliczbowe i zmiennoprzecinkowe mogą być niejawnie konwertowane na typ `bool` (6.2.2). Wartości niezerowe są zamieniane na `true`, a zero na `false`. Na przykład:

```
void f(int*p, int i)
{
    bool is_not_zero = p; // true, jeśli p!=0
    bool b2 = i;         // true, jeśli i!=0
    //...
}
```

Zamiana wskaźników na wartości logiczne jest przydatna w warunkach, ale w innych miejscach może powodować nieporozumienia:

```
void fi(int);
void fb(bool);

void ff(int*p, int*q)
{
    if (p) do_something(*p); // OK
    if (q!=nullptr) do_something(*q); // OK, ale rozwlekle
    //...
    fi(p); // błąd: nie można konwertować wskaźnika na int
    fb(p); // OK: konwersja wskaźnika na bool (niespodzianka!?)
}
```

Trzeba mieć nadzieję, że w przypadku `fb(p)` kompilator wyświetli ostrzeżenie.

10.5.2.6. Konwersje między typami zmiennoprzecinkowymi i całkowitymi

Przy konwersji wartości zmiennoprzecinkowej na całkowitą następuje utrata części ułamkowej. Innymi słowy, konwersja typu zmiennoprzecinkowego na całkowitoliczbowy powoduje skrócenie wartości. Na przykład wartość wyrażenia `int(1.6)` to 1. Jeśli skróconej wartości nie można reprezentować przy użyciu typu docelowego, to wynik takiej operacji jest niezdefiniowany. Na przykład:

```
int i = 2.7; // i ma wartość 2
char b = 2000.7; // niezdefiniowane dla 8-bitowych znaków: 2000 nie można reprezentować w 8-bitowej
                // zmiennej typu char
```

Konwersje typów zmiennoprzecinkowych na całkowite są na tyle dokładne matematycznie, na ile pozwala sprzęt. Gdy wartość całkowita nie może być reprezentowana dokładnie jako wartość zmiennoprzecinkowa, następuje utrata precyzji. Na przykład:

```
int i = float(1234567890);
```

W komputerze, w którym typy `int` i `float` mają 32 bity, wartość `i` wyniesie 1234567936.

Oczywiście najlepiej jest unikać niejawnych konwersji mogących spowodować utratę części informacji. Niektóre oczywiste niebezpieczeństwa tego typu, takie jak konwersja liczby zmiennoprzecinkowej na całkowitą albo `long int` na `char`, mogą być wykrywane przez kompilatory. Mimo to należy uważać, bo poleganie na kompilatorze w tej kwestii jest niepraktyczne. Jeśli sama ostrożność to za mało, programista może zastosować jawną konwersję kontrolowaną. Na przykład:

```
char checked_cast(int i)
{
    char c = i; // ostrzeżenie: nieprzełożenie (10.5.2.1)
    if (i != c) throw std::runtime_error{"Nieudana konwersja int na char"};
    return c;
}

void my_code(int i)
{
    char c = checked_cast(i);
    // ...
}
```

Bardziej ogólne podejście do wyrażania konwersji kontrolowanych jest opisane w sekcji 25.2.5.1.

Aby skrócenie wartości nie niweczyło przełożności, należy użyć `numeric_limits` (40.2). W inicjacjach skrócenia można wyeliminować, stosując notację `{}` (6.3.5).

10.5.3. Typowe konwersje arytmetyczne

Opisane poniżej konwersje są wykonywane na argumentach operatorów binarnych w celu sprowadzenia ich do wspólnego typu, który następnie jest używany jako typ wyniku działania:

- Jeśli jeden z argumentów jest typu `long double`, drugi również jest konwertowany na `long double`.
 - W przeciwnym przypadku, jeśli jeden argument jest typu `double`, drugi również jest konwertowany na `double`.
 - W przeciwnym przypadku, jeśli jeden argument jest typu `float`, drugi również jest konwertowany na `float`.
 - W przeciwnym przypadku na obu argumentach jest wykonywana operacja promocji całkowitoliczbowej (10.5.1).
- Jeżeli jeden z argumentów jest typu `unsigned long long`, to drugi zostaje również przekonwertowany na typ `unsigned long long`.
 - W przeciwnym przypadku, jeśli jeden argument jest typu `long long int`, a drugi `unsigned long int`, to jeżeli typ `long long int` może reprezentować wszystkie wartości typu `unsigned long int`, typ `unsigned long int` jest konwertowany na `long long int`. W przeciwnym przypadku oba argumenty są konwertowane na typ `unsigned long long int`. W przeciwnym przypadku, jeżeli jeden z argumentów jest typu `unsigned long long`, drugi również zostaje przekonwertowany na `unsigned long long`.

- Jeżeli jeden z argumentów jest typu `long int`, drugi jest `unsigned int`, a typ `long int` może reprezentować wszystkie wartości typu `unsigned int`, to wartość typu `unsigned int` jest konwertowana na `long int`. W przeciwnym przypadku oba argumenty są konwertowane na typ `unsigned long int`.
- Jeśli jeden z argumentów jest typu `long`, drugi również jest konwertowany na `long`.
- Jeśli jeden z argumentów jest typu `unsigned`, drugi również jest konwertowany na `unsigned`.
- W pozostałych przypadkach oba argumenty są typu `int`.

Z tych zasad wynika, że wynik konwersji wartości całkowitej bez znaku na wartość ze znakiem ewentualnie większego rozmiaru jest zależny od implementacji. Jest to kolejny powód, aby nie mieszać liczb całkowitych ze znakiem z liczbami całkowitymi bez znaku.

10.6. Rady

1. Wybieraj narzędzia z biblioteki standardowej zamiast innych bibliotek i własnego kodu — 10.2.8.
2. Pobieraj znaki na wejściu tylko wtedy, gdy nie możesz tego zrobić inaczej — 10.2.3.
3. Przy wczytywaniu danych zawsze zabezpiecz się przed możliwością pojawienia się niepoprawnych danych — 10.2.3.
4. Stosuj odpowiednie abstrakcje (klasy, algorytmy itd.) zamiast bezpośredniego używania narzędzi języka (np. `int` i instrukcji) — 10.2.8.
5. Unikaj używania skomplikowanych wyrażeń — 10.3.3.
6. W razie wątpliwości co do kolejności wykonywania działań używaj nawiasów — 10.3.3.
7. Unikaj wyrażeń z nieokreśloną kolejnością wykonywania działań — 10.3.2.
8. Unikaj konwersji zawężających — 10.5.2.
9. Definiuj stałe symboliczne, aby wyeliminować „stałe magiczne” — 10.4.1.
10. Unikaj konwersji zawężających — 10.5.2.

Skorowidz

*Wyróżnia się dwa rodzaje wiedzy. Można wiedzieć coś na dany temat
albo można wiedzieć, gdzie szukać o tym informacji
— Samuel Johnson*

A

- ABI, Application Binary Interface, 556
- abstrakcja danych, 31, 44
- abstrakcyjny typ danych, 101
- adaptacje
 - funkcji, 972
 - iteratorów, 965
 - kontenerów, 929
 - mechanizmu liczb losowych, 1180, 1184
- adaptry kontenerów, 894, 895
- ADL, argument-dependent lookup, 768
- aksjomaty, 727
- algorytm, 137, 716
 - accumulate(), 1177
 - adjacent_difference(), 1178
 - binary_search(), 952
 - copy(), 944
 - count(), 940
 - equal(), 942
 - fill(), 948
 - find(), 941
 - for_each(), 940
 - inner_product(), 1177
 - iota(), 1179
 - lexicographical_compare(), 956
 - max, 957
 - merge(), 954
 - min, 957
 - mismatch(), 942
 - nth_element(), 952
 - partial_sum(), 1178
 - partition(), 947
 - random_shuffle(), 947
 - remove(), 946
 - replace(), 946
 - rotate(), 947
 - search(), 942
 - sort, 950
 - swap(), 949
 - transform(), 943
 - try_lock(), 1229
 - unique(), 945
- algorytmny, 714, 872
 - biblioteki standardowej, 143
 - działające na zbiorach, 954
 - kontenerowe, 143
 - matematyczne, 162
 - modyfikujące sekwencje, 943
 - nie modyfikujące sekwencji, 940
 - numeryczne, 1176
 - permutacyjne, 948
 - sprawiedliwe, 1221
 - stertowe, 955
 - STL, 935
- aliasy, 119
 - przestrzeni nazw, 436
 - szablonów, 708
 - typów, 202, 796
 - typów całkowitoliczbowych, 1265
 - typów składowych, 692
- alokacja, 580
- alokator, 895, 998
 - domyślny, 1000
 - zakresowy, 1003
- alokowanie pamięci, 315
- anonimowe przestrzenie nazw, 444
- archetyp, 732
- argument
 - Facet*, 1119
 - wartościowy, value
 - argument, 738
- argumenty
 - algorytmów STL, 937
 - domyślne, 356
 - domyślne szablonów, 742
 - domyślne szablonów funkcji, 744
 - formalne, 347
 - listowe, 351
 - referencyjne, 298, 348
 - szablonu, 736
 - szablonu funkcji, 701
 - tablicowe, 350
 - wskaźnikowe, 348
 - zasad, 938
- arytmetyka
 - mieszana, 560
 - wektorów, 165
- ASCII, 171
- asercja, 93, 391, 882
- atomowe
 - typy całkowitoliczbowe, 1203
 - wskaźniki, 1205

B

bariera pamięci, 1206
 bazy
 chronione, 626
 klasy pochodnej, 525
 prywatne, 625
 wirtualne, 653, 655
 bezpiecznie derywowane
 wskaźniki, 1005
 bezpośrednia kwalifikacja, 422
 białe znaki, 290
 biblioteka, 121
 standardowa, 39, 61, 64, 122, 865, 867
 nagłówki, 124
 przeczeń nazw, 123
 standardowa C, 1253
 data i godzina, 1261
 łańcuchy, 1259
 pamięć, 1260
 pliki, 1253
 rodzina printf(), 1254
 stdio, 1257
 STL, 716, 893
 wątków, 1192
 wspomagająca zadań, 1192
 binarne drzewo zrównoważone, 723
 blok konsolidacji, 457
 blokada, 1192, 1197, 1221
 dla muteksu, 1220, 1225, 1227
 z podwójnym
 zatwierdzeniem, 1204
 blok try, 401
 błąd, 90, 283, 373, 690, 861, 885
 dotyczący typu, 690
 Heisenbug, 1216
 szukania nazwy, 690
 błędy
 future, 1242
 muteksów, 1224
 składniowe, 690
 systemowe, 883
 bufor, 1105, 1106
 strumieniowy, 1102
 tymczasowy, 1007
 buforowanie, 1102
 buforowanie konwersji, 1157

C

cechy, trait, 800
 alokatorów, 1001
 czasu, 1018
 iteratorów, 962
 typów, 1020
 wskaźników, 1002
 znaków, 1036
 chronione
 pobieranie, 1104
 wstawianie, 1104
 ciągi siN, 1014
 cięcie obiektów, 536
 CRTP, 785
 cykl istnienia obiektów, 201, 327
 czas, 1011
 kompilacji, 803
 wykonywania, 803

D

dane, 241
 lokalne wątku, 1218
 składowe, 691
 data i godzina, 1261
 dedukcja
 argumentów, 702
 referencji, 703
 typu, 197
 definicja, 187
 definicja warunkowa, 807
 definiowanie
 fasety, 1122
 funkcji, 339
 predykatów, 799
 przez implementację, 170
 szablonu, 685
 deklaracja, 76, 87, 186, 259
 funkcji, 45, 337
 szablonu, 684
 using, 423, 627
 w klauzulach case, 263
 w warunkach, 264
 dekrementacja, 307, 578
 delegowanie konstruktorów, 526
 demony, 1215
 dereferencja, 576
 derywacja, 708
 klas, 599, 600
 publiczna, 625

derywowane interfejsy, 642
 destrukcja, 675
 destruktor, 99, 340, 506, 604, 905
 klas bazowych, 510
 składowych klas, 510
 typu thread, 1213
 wirtualny, 512, 639
 dezalokacja, 580
 diagnostyka, 872
 dołączanie wielokrotnie
 nagłówków, 467
 domieszka, 658
 dopasowania klasy regex, 1062
 dopasowywanie, 1061
 leniwe, 1058
 wyrażeń regularnych, 1065
 dopełnienie, 1256
 dostęp do
 elementów, 818, 908
 faset, 1121
 klas, 45
 klas bazowych, 625
 macierzy, 855
 Matrix, 843
 reprezentacji, 598
 składowych, 84, 491, 565, 772
 składowych chronionych, 624
 struktury danych, 480
 znaków, 585
 drzewo, 723, 782
 dynamiczne rzutowanie
 referencji, 663
 dyrektywa using, 424, 433
 działania
 arytmetyczne ratio, 1019
 na liczbach wymiernych, 1019
 dziedziczenie, 102, 601
 chronione, 639
 implementacji, 106, 600, 620, 634, 642
 interfejsu, 106, 600, 620, 637, 642
 konstruktorów, 616
 dzielenie na tokeny, 1071

E

ekwiwalencja typów, 241, 533, 689
 element
 centralny, pivot, 860
 maksymalny, 957
 minimalny, 957

elementy

- języka, 58, 60, 62
- kontenera, 898
- pominięte, 1277
- wycofywane, 1270

eliminacja Gaussa, 858, 859

entropia, 1185

epoka, 1015

F

fabryka, 643

faseta, 1114

- codecvt, 1148

- collate, 1128, 1130

- cctype, 1145

- messages, 1151

- money_get, 1140

- money_put, 1139

- money_punct, 1137

- num_get, 1134

- num_put, 1132

- num_punct, 1131

- String_numput, 1133

- time_get, 1142, 1143

- time_put, 1141

- wbuffer_convert, 1158

- wstring_convert, 1156

fasety standardowe, 1125, 1133

finalizacja, 388

flagi atomowe, 1206

formatowanie, 1255

- daty i godziny, 1141, 1263

- kwot pieniężnych, 1136

- liczb, 1131

- regex, 1063

funkcja, 45, 337, 338

- accept(), 674

- accumulate(), 155

- async(), 154, 156, 1245, 1250

- begin(), 136, 970

- bind(), 164, 972

- call_once(), 1230

- compare(), 1129

- composer(), 1223

- constexpr, 344

- decltype(), 813

- declval(), 1029

- default_date(), 498

- detach(), 1215

- duration_cast, 157

- end(), 136, 970

- entropy(), 1185

- find(), 1048, 1247

- find_all(), 937

- find_all_rec(), 1249

- fopen(), 1254

- forward(), 1030

- frac_digits(), 1139

- get(), 1233

- get_obj(), 667

- gets(), 1258

- imbue(), 1132

- in(), 1149

- join(), 1214

- lock(), 1229

- main(), 462

- make_tuple(), 820

- malloc(), 1260

- mem_fn(), 974

- move(), 539, 1030

- negative_sign(), 1138

- operator new(), 548

- operator*, 595

- operator[](), 819

- operator->(), 809

- pfind(), 1247

- pfind_any(), 1249

- pop(), 930

- prim(), 277

- printf(), 821, 1254, 1257

- push_back(), 136, 415

- put(), 1088, 1134, 1232

- realloc(), 1260

- ref(), 974

- regex_match(), 1064

- regex_replace(), 1063, 1067

- regex_search(), 1066

- reserve(), 414

- resize(), 415

- scanf(), 1257

- seekp(), 1106

- set_exception(), 155

- set_value(), 155

- size(), 136

- sort(), 158

- strftime (), 1255

- suspend(), 629

- swap(), 410, 412, 930, 1031

- tail(), 828

- terminate(), 881

- tie(), 1093

- tolower(), 1146

- transform(), 1128

- typeid(), 676

funkcje

- [[noreturn]], 346

- constexpr, 343

- czysto wirtualne, 619

- daty i czasu, 1261

- dostępowe, 565

- get, 565

- globalne, 597

- inline, 342

- klasy locale, 1121

- klasy String, 589, 591

- klasyfikacji, 1035

- matematyczne, 162, 1163

- dodatkowe, 1164

- specjalne, 1164

- standardowe, 1163

- mieszające, 923, 925

- noexcept, 396

- operatorowe, 553

- pobierania, 1105

- pomocnicze, 500, 565

- pomocnicze shared_ptr, 996

- porównujące, 925

- rekurencyjne, 341

- set, 565

- składowe, 477, 498, 602, 692

- szablonowe, 686, 699

- typowe, 158, 794, 1020

- wirtualne, 101, 103, 607, 657

- wstawiania, 1105

- wyjściowe, 1088

- wyrażeń regularnych, 1064

- zaprzyjaźnione, 596

- znakowe, 1258

G

generator liczb losowych, 1180

- rozkład, 163

- równomierny, 1182

- silnik, 163

generowanie

- operacji, 541

- specjalizacji, 757

- typów, 776, 1025

granice liczbowe, 1160

grupowanie, 1058

gwarancja

- bezpieczeństwa

- wyjatkowego, 373

- niezgłaszania wyjątku, 384

- podstawowa, 384

- wyjatków, 383

H

hierarchia

- dziedziczenia, 786
 - implementacji, 640
 - Ival_box, 642
 - klas, 104, 604, 633, 659, 672
 - strumieni, 1077
 - szablonów, 773
 - szablonów klas, 779
 - wyjątków, 878
- hiperwęteł, 1193
- hiperwętełowość, 1211

I

- identyfikacja typów, 675
- identyfikator, 189
- identyfikator wątku, 1211
- implementacja, 431, 433
- Enable_if, 811
 - hostowana, 171
 - klasy Ival_box, 636
 - klasy locale, 1113
 - macierzy, 850
 - samodzielna, 171
 - szablonów, 795
 - wektora, 405
- indeksowanie, 573
- krotki, 819
 - valarray, 1168
- informacje o typach, 617, 659
- inicjacja, 194
- bazy, 525
 - bezpośrednia, 484, 522
 - kopitująca, 484, 522
 - listy, 194
 - listy macierzy, 853
 - obiektów
 - bez konstruktorów, 513
 - przez konstruktory, 516
 - składowych, 524, 525
 - składowych statycznych, 529
 - uniwersalna, universal
 - initialization, 516
 - wyrażeniem stałym, 470
 - zmiennych nielokalnych, 469
- inicjatory wewnętrzklasowe, 485, 527
- inicjowanie kontenerów, 100
- inkrementacja, 307, 578

- instrukcja, 48, 257
- do, 267
 - for, 265, 266
 - goto, 269
 - if, 260
 - switch, 261
 - while, 267
- instrukcje
- iteracyjne, 264
 - wyboru, 260
- interfejs, 59, 428, 433, 644, 657
- do obiektów, 45
 - kubeków, 928
 - odśmiecacza, 1004
- interfejsy pomocnicze, 1155
- internacjonalizacja, 1112
- interpunkcja
- liczb, 1131
 - w kwotach pieniężnych, 1137
- iteracja, 805
- iterador, 138, 732, 907
- istreambuf_iterator, 1107
 - ostreambuf_iterator, 1108
 - raw_storage_iterator, 1008
 - regex_iterator, 1068
 - regex_token_iterator, 1070
 - Slice_iter, 1174
- iteratory
- buforów, 1107
 - dwukierunkowe, 961, 969
 - jednokierunkowe, 158, 732, 961
 - losowego dostępu, 158
 - łańcuchowe klasy
 - basic_string, 1046
 - o dostępie swobodnym, 961
 - odwrotne, 966
 - przenoszące, 969
 - STL, 959
 - strumieni, 140
 - strumieniowe, 1101
 - wejściowe, 961
 - wstawiające, 968
 - wyjściowe, 961
 - wyrażeń regularnych, 1068

J

- jawna
- konwersja typów, 330
 - reprezentacja pamięci, 409

jawne

- operacje domyślne, 541
 - żądanie konkretyzacji, 758
- jawny konstruktor domyślny, 695
- jednostka translacji, 448
- jednostki układu SI, 830, 1014
- język, 41
- C, 53
 - C++, 1271
 - Java, 54

K

- kacze typowanie, 714
- kalkulator, 465
- argumenty, 285
 - moduły, 465
 - nagłówki, 284
 - obsługa błędów, 283
 - parser, 274
 - sterownik, 284
 - styl, 286
 - wejście, 278
 - wejście niskopoziomowe, 282
- kategorie
- błędów, 885
 - iteratorów, 961
- klasa, 31, 45, 84, 96
- allocator, 1004
 - atomic, 1205
 - basic_ios, 1090, 1091
 - basic_istream, 1076
 - basic_regex
 - konstruktory, 1060
 - operacje, 1060
 - stałe składowe, 1059
 - basic_streambuf, 1102, 1105
 - basic_string
 - iteratory łańcuchowe, 1046
 - konstruktory, 1040
 - operacje dostępowe, 1043
 - operacje porównywania, 1042
 - operacje usuwania, 1047
 - operacje wejścia, 1044
 - operacje wstawiania, 1047
 - operacje wyjścia, 1044
 - operacje zamiany, 1047
 - podłańcuchy, 1049

- pojemność, 1042
- przypisania, 1046
- rozmiar, 1042
- znajdowanie elementów, 1048
- char_traits, 1037
- collate, 1129
- collate_byname, 1130
- complex, 559, 565, 1165
- Date, 496, 497
- facet, 1120
- future, 154, 1241
- ios_base, 1090, 1093, 1094
- Ival_box, 634, 636, 637, 640, 642
- Ival_slider, 656
- locale, 1111, 1114
- lock_guard, 1226, 1227
- macierzowa, 862
- Matrix, 852
- Matrix_ref, 852
- mutex, 1222
- nested_exception, 880
- Node, 674, 787
- packaged_task, 154, 1238
- pair, 986
- pochodna, 45, 638, 639
- promise, 154, 1238
- recursive_mutex, 1222
- regex, 1059, 1062
 - dopasowania, 1062, 1063
 - formatowanie, 1063
 - opcje dopasowywania, 1064
 - opcje formatowania, 1064
 - poddopasowania, 1062, 1063
- regex_traits, 1072
- Shape, 671
- shared_future, 1244
- składowa, 494
- String, 584
 - dostęp do znaków, 585
 - działanie operatorów, 593
 - funkcje pomocnicze, 588, 591
 - funkcje składowe, 589
 - operacje podstawowe, 585
 - reprezentacja, 586
- sub_match, 1061
- system_error, 886
- szablonowa, 686, 691
- aliasy typów składowych, 692
- dane składowe, 691
- funkcje składowe, 692
- przyjaciele, 698
- składowe statyczne, 692
- szablony składowe, 694
- typy składowe, 693
- tuple, 988
- unique_lock, 1227
- Vector, 130
- vector_base, 409
- wstring_convert, 1156
- wyliczeniowa, enum class, 233
- klasy, 475
 - abstrakcyjne, 101, 619
 - bazowe, 45, 102, 601, 649
 - dostęp do składowych, 491
 - funkcje pomocnicze, 500
 - funkcje składowe, 477, 498
 - implementacyjne, 644
 - inicjacja bazy, 524
 - inicjacja obiektów, 513
 - inicjacja składowych, 524
 - inicjatory wewnątrzklasowe, 485, 527
 - inicjowanie obiektów, 481
 - interfejsowe, 656
 - konkretne, 96, 495
 - kontrola dostępu, 479
 - kopiowanie obiektu, 479
 - muteksów, 1220
 - pochodne, 102, 600
 - destrukторы, 604
 - funkcje składowe, 602
 - konstrukторы, 604
 - pośrednie, 649
 - składowa modyfikowalna, 488
 - składowe statyczne, 492
 - stałe funkcje składowe, 487
 - tablic numerycznych, 1166
 - w hierarchiach klas, 104
 - wirtualne, 651, 654
 - wyliczeniowe, 250
 - zaprzyżnione, 594
 - zmiennosc, 487
 - znaków, 1056
- klasyfikacja znaków, 1035, 1144, 1155
- klauzula case, 263
- kod zawierający szablony, 709
- kody błędów, 883
 - errc, 885, 889
 - future_errc, 891
 - io_errc, 891
 - przenośne, 887
- kolejka, queue, 136, 931
- komunikatów, 1240
- priorytetowa, 931
- Sync_queue, 1234
- kolejność
 - instrukcji, 1195
 - wykonywania działań, 292
- komentarze, 269
- kompilacja
 - rozdzielna, 447
 - warunkowa, 370
- kompozycje, 419
- kompozycyjne obiekty
 - zamknięć, 864
- kompozytor, 864
- komunikacja między zadaniami, 154
- koncepcja, 688, 718, 722, 811
- koncepcje
 - ad hoc, 723
 - wartości, 729
 - wieloargumentowe, 728
- konflikt nazw, 420
- konkretyzacja, 755
 - koncepcji, 724
 - szablonu, 687, 756
- konserwatywne odświeżanie, 1006
- konsolidacja, 448, 711
 - a wskaźniki do funkcji, 458
 - z obcym kodem, 456
- konstrukcja, 331, 675
 - atomowe, 1198
 - pomocnicze pair, 987
- konstruktor, 85, 340, 481, 507, 604, 905
 - bitset, 982
 - delegujący, 526
 - domyślny, 97
 - explicit, 483
 - inicjujący, 506
 - kopiujący, 109, 506
 - map, 919
 - przekazujący, 526
 - przenoszący, 111, 411, 506
 - typu thread, 1212
 - unordered_map, 924, 925
 - wirtualny, 618, 643
 - z listą inicjacyjną, 100, 519

- konstruktory
 - domyślne, 517, 532, 543
 - klasy `basic_regex`, 1060
 - klasy `basic_string`, 1040, 1041
 - klasy `locale`, 1119
 - klasy `valarray`, 1166
 - konstruowanie
 - baz wirtualnych, 653
 - słowników
 - nieuporządkowanych, 923
 - kontekstowe słowo kluczowe, 612
 - kontener, 98, 129, 871
 - array, 978, 980
 - `basic_string`, 978
 - bitset, 978, 981
 - deque, 136
 - `forward_list`, 136, 916
 - list, 133, 916
 - map, 134
 - multimap, 136
 - set, 136
 - tablica, 914
 - `unordered_map`, 135
 - `unordered_multiset`, 136
 - `unordered_set`, 136
 - vector, 130, 165, 911, 930
 - `vector<bool>`, 978, 985
 - kontenery
 - adaptacje, 929
 - destruktory, 904
 - iteratory, 907
 - konstruktory, 904
 - operacje, 901
 - listowe, 909
 - stosowe, 908
 - pojemność, 906
 - porównywania, 910
 - przypisania, 905
 - rozmiar, 906
 - typy składowe, 904
 - zamienianie, 910
 - kontenery asocjacyjne, 893, 917
 - heterogeniczne, 978
 - homogeniczne, 978
 - nieuporządkowane, 135, 895, 917, 922
 - sekwencyjne, 893, 894
 - STL, 893
 - uporządkowane, 895, 917
 - kontrawariacja, 631
 - kontrola
 - dostępu, 479, 621, 626
 - przesłania, 610
 - typów, 46, 51, 680, 688
 - wyjątków, 1082
 - kontrolka, 660
 - konwersja, 300, 561
 - argumentów, 563
 - arytmetyczna, 303
 - całkowitoliczbowa, 301
 - kodów znaków, 1147, 1150
 - łańcuchów, 1156
 - między typami, 302
 - niejawna, 299
 - numeryczna, 1044, 1259
 - obiektu `ios`, 1093
 - szablonów, 778
 - typów, 567
 - typów jawna, 330
 - wartości, 1105
 - wartości logicznych, 302
 - wskaźników do składowych, 302
 - wskaźników i referencji, 301
 - zawężająca, 300, 694
 - zmiennoprzecinkowa, 301
 - znaków, 1156
 - kopiowanie, 108, 506, 530, 538
 - baz, 532
 - domyślne, 478
 - głębokie, 534
 - kontenerów, 108
 - plytkie, 534
 - przy zapisie, 535
 - kwowariantne typy zwrotne, 617
 - krotka, tuple, 815, 824, 981, 986
 - krotki stałe, 819
 - kryteria projektowe, 869
 - kryterium porządkowania, 899
 - kształty, 671
 - kubelki, 927
 - kwalifikator `Lexer::`, 432
- ## L

M
- lambda, 322, 325–328
 - liczba argumentów, 353
 - liczby, 1159
 - całkowite, 1162
 - losowe, 163, 1180
 - wymierne, 1019
 - zespolone, 163, 559, 1164
 - likwidowanie wątku, 1218
 - limity liczbowe, 165
 - linearyzacja hierarchii klas, 785
 - lista, 136, 318, 915
 - czasu kompilacji, 814
 - dwukierunkowa, 915
 - `forward_list`, 897
 - inicjacyjna, 196, 521
 - inicjacyjna składowych, 524
 - jednokierunkowa, 915
 - list, 133, 897, 915
 - kwalifikowana, 319
 - nieintruzyjna, 621
 - niekwalifikowana, 320
 - przechwytywania, 117
 - wiązana dwustronnie, 133
 - literal, 564
 - całkowitoliczbowy, 179, 582
 - jednostkowy, 833
 - łańcuchowy, 582
 - zdefiniowany przez użytkownika, 581
 - zmiennoprzecinkowy, 181, 582
 - znakowy, 177, 582
 - lokalizacja, 873, 1111
 - nazwana, 1116
 - pamięci, 1194
 - strumienia, 1096
 - lokalizowanie tworzenia obiektu, 642
 - losowanie liczb, 1189
- ## Ł
- łańcuch, 82, 124, 873, 897, 1035, 1038–1040
 - łańcuchowe wejście, 1044
 - łańcuchowe wyjście, 1044
 - łańcuchy w stylu C, 1259
 - łączenie dziedziczenia z parametryzacją, 790

- makra, 368
 - granic liczb
 - całkowitych, 1162
 - zmiennoprzecinkowych, 1163
 - predefiniowane, 371
 - makro `__cplusplus`, 1278
 - manipulatory, 1088
 - istream, 1098
 - standardowe, 1096
 - wejścia i wyjścia, 1096–1098
 - zdefiniowane przez użytkownika, 1099
 - map, 134
 - mapa `io_map`, 668
 - mapy, 135
 - maska, 1144
 - mechanizm liczb losowych, 1180, 1183
 - metaprogram, 793
 - metaprogramowanie, 158, 791, 806
 - metaprogramowanie szablonowe, 158, 714, 791
 - metoda, 607
 - miejsce konkretyzacji, 691, 763, 766
 - mieszanie, 925, 928
 - mieszanie obiektów, 1032
 - model
 - implementacji, 318, 322
 - iteratorów, 959
 - pamięci, 1192, 1193
 - modernizacja programów, 1277
 - modularność, 419
 - modularyzacja, 428
 - modułowość, 87
 - moduły, 430
 - modyfikacja
 - const, 1025
 - referencji, 1025
 - tablicy, 1026
 - volatile, 1025
 - wskaźników, 1026
 - znaku, 1026
 - modyfikatory formatu daty i czasu, 1264
 - modyfikowalność przez pośredniość, 489
 - mutex, 1210, 1220
 - recursive_timed_mutex, 1224
 - timed_mutex, 1224
- N**
- nadklasa, 102, 601
 - nagłówki
 - implementacji samodzielnej, 171
 - języka C, 444
 - z biblioteki standardowej, 455
 - narzędzia, 36, 157, 167, 872
 - atomowe, 1202
 - biblioteki standardowej, 868
 - C++, 1277
 - drobne, 1030
 - językowe, 1268
 - pomocnicze, 1011
 - pomocnicze tuple, 989
 - nazwa szablonu klasy, 684
 - nazwy, 189
 - lokalizacji, 1116, 1118
 - lokalne, 770
 - lokalne w plikach, 451
 - niezależne, 760
 - przestrzeni nazw, 327
 - struktur, 236
 - z klas bazowych, 770
 - zależne, 760, 761
 - niejawna konwersja typów, 299
 - niejednoznaczności, 569, 646
 - nieokreślona liczba argumentów, 353
 - niezależność, 533
 - nieziemiennik, 91, 373, 389, 508, 543
 - klasy, 508
 - określone częściowo, 545
 - zasobów, 544
 - niszczenie wątku, 1213
 - notacja wyrażeń regularnych, 1054
- O**
- obiekt, 76
 - blokowany, 1227
 - duration, 1012
 - error_category, 884
 - future, 1242
 - Ival_box, 634
 - POD, 242
 - shared_ptr, 993
 - slice_array, 1174
 - streambuf, 1103
 - obszar pobierania, 1103
 - obszar wstawiania, 1103
 - thread, 1210
 - zasad, policy object, 117
 - obiekty, 200
 - automatyczne, 201
 - duration, 1014
 - dynamiczne, 196
 - funkcyjne, 116, 340, 575, 851, 971
 - lokalne w wątkach, 202
 - spłątane, 534
 - styczne, 196, 201
 - tymczasowe, 202, 293
 - obliczenia liczbowe, 874
 - obsługa
 - błędów, 90, 283, 373, 877, 1081, 1123
 - hierarchiczna, 380
 - niedbała, 377
 - tradycyjna, 376
 - wyjątków, 373, 381
 - pamięci, 1260

- odczyt strumienia, 1107
 - odkrywanie koncepcji, 718
 - odśmiecacz, 1004, 1006
 - odzyskiwanie interfejsu, 667
 - ograniczenia, constraint, 722
 - opcje formatowania
 - regex, 1064
 - strftime, 1255
 - operacja
 - _cast, 333
 - dynamic_cast, 333
 - reinterpret_cast, 333
 - shared_ptr, 995
 - static_cast, 333
 - operacje
 - arytmetyczne, 972
 - atomowe, 1196, 1198
 - bitset, 983, 984
 - domyślne, 541, 542
 - dostępowe klasy basic_string, 1043
 - formatowania fmtflags, 1095
 - iteratorów, 964, 965
 - iteratorów strumieniowych, 1101
 - jako argumenty, 739
 - klasy
 - basic_ios, 1091
 - basic_regex, 1060
 - basic_streambuf, 1103
 - future, 1241
 - ios_base, 1090, 1093
 - mutex, 1222
 - packaged_task, 1238
 - regex_traits, 1072
 - shared_future, 1244
 - sub_match, 1061
 - konsumpcji, 1199
 - kontenera forward_list, 917
 - kontenera list, 916
 - kontenerowe, 901, 902
 - kontenerów asocjacyjnych, 919, 920
 - listowe, 909
 - na łańcuchach, 1259
 - pobierania, 1104, 1199
 - porównywania i zamiany, 1203
 - porównywania klasy
 - basic_string, 1042
 - składowe valarray, 1170
 - stertowe, 956
 - stosowe, 908
 - strumieni, 1076
 - strumienia fstream, 1078
 - strumienia stringstream, 1080
 - usuwania klasy basic_string, 1047
 - wejściowe, 1083
 - wstawiania, 1104
 - wstawiania klasy
 - basic_string, 1047
 - wyjściowe, 1086, 1087
 - zamiany klasy basic_string, 1047
 - zwolnienia, 1199
 - operator
 - delete, 314
 - new, 314, 540
 - dynamic_cast, 666
 - noexcept, 396
 - zakresu ::, 610
 - operatory, 48, 287
 - aplikacji, 575
 - deklaracyjne, 82, 188
 - dwuargumentowe, 554
 - jednoargumentowe, 554
 - klasy complex, 1165
 - klasy String, 593
 - konwersji, 567
 - konwersji explicit, 569
 - literałów, 581
 - literałowe szablonowe, 583
 - logiczne, 305
 - logiczne bitowe, 306
 - porównywania, 930
 - relacyjne, 291, 900, 1031
 - składowe, 559
 - specjalne, 573
 - w przestrzeniach nazw, 557
 - wywołania, 575
 - zewnątrzne, 559
 - optymalizacja, 816
 - autoprzypisania, 412
 - krótkich łańcuchów, 584, 587
 - pustej bazy, 784, 816
 - węzła, 788
 - optymalizator, 863
 - opuszczanie inicjatora, 195
 - organizacja
 - końca źródłowego, 709
 - z jednym nagłówkiem, 459
 - z wieloma nagłówkami, 463
 - osobna kompilacja, 88
 - otwieranie plików, 1253
- ## P
- paczka parametrów, parameter pack, 823, 824
 - pamięć, 977, 1260
 - dynamiczna, 83, 99
 - niezainicjowana, 1007
 - wolna, 83, 202, 309
 - parametr
 - T, 623
 - typowy, 736
 - wartościowy, 738
 - szablonu, 736, 781
 - parser, 274
 - permutacje, 948
 - pętla, 79, 80, 268
 - plik
 - parser.cpp, 461, 463
 - parser.h, 463
 - pliki
 - .cpp, 712
 - nagłówkowe, 88, 451
 - źródłowe, 74, 447, 448
 - pobieranie kwot pieniężnych, 1140
 - POD, plain old data, 798
 - poddopasowania klasy regex, 1062
 - podklasa, 102, 600
 - podłańcuchy, 1049
 - podstawianie argumentu, 706
 - podzbiór elementów tablicy, 1168
 - pojemność basic_string, 1042
 - pola, 244
 - bitowe, 244
 - typów, 605
 - polimorfizm, 609
 - czasu działania, 644, 773
 - czasu kompilacji, 735, 773
 - dwukierunkowy, 1088
 - parametryczny, 773
 - podwójny, 670, 671
 - połączenia pętli, loop fusion, 862
 - pomiary złożoności, 903
 - ponawianie zgłoszenia wyjątku, 399
 - poprawa
 - bezpieczeństwa typowego, 792
 - wydajności wykonywania, 792
 - porównywanie, 741, 925, 1042, 1050
 - leksykograficzne, 956
 - łańcuchów, 1120, 1127
 - obiektów, 1032

- porządek
 - pamięci, 1196
 - specjalizacji, 750
 - totalnym, total order, 900
 - pośredniość, indirection, 577
 - powiązane przestrzenie nazw, 427
 - powiązania między węzłami, 788
 - powiększanie wektora, 911
 - poziom zabezpieczenia, 927
 - pozycja bitu, 982
 - pragmy, 372
 - prawie kontenery, 894, 896, 977
 - predykat, 116, 142, 971
 - isspace(), 941
 - std::is, 798
 - predykaty
 - sekwencji, 940
 - typów, 160, 798
 - prymarne, 1020
 - złożone, 1021
 - właściwości typów, 1022–1024
 - prezenter lambdy, lambda
 - introducer, 325
 - priorytety operatorów, 292
 - problem
 - ABA, 1203
 - z wersjami, 639
 - procedura obsługi
 - wyjątków, 400
 - zamknięcia programu, 403
 - proces, 1192, 1193
 - programowanie, 43
 - bez użycia blokad, 1192
 - dwupoziomowe, 791
 - funkcyjne, 824
 - obiektowe, 44, 600
 - ogólne, 44, 713
 - proceduralne, 44, 73
 - systemów, 42
 - w języku C++, 52
 - wartościowe, 503
 - wielopoziomowe, 791
 - programy, 468
 - obiektowe, 31
 - zmienne nielokalne, 469
 - projekt
 - języka C++, 41
 - macierzy, 837
 - projektowanie hierarchii klas, 633
 - promocja
 - całkowitoliczbowa, 300
 - zmiennoprzecinkowa, 300
 - propagacja wyjątków, 879
 - próbka, 1182
 - prymarne predykaty typów, 1020
 - przechodniość, 899
 - przechodniość ekwiwalencji, 899
 - przechwytywanie wyjątków, 397, 399
 - przeciążanie, 439, 708, 750
 - a typ zwrotny, 360
 - a zakres, 360
 - funkcji, 358
 - operatora new, 315
 - operatorów, 502, 551
 - szablonów funkcji, 704
 - przeciwsymetria, 899
 - przeciwwrotność, 899
 - przedrostki, 182
 - przekazywanie
 - argumentów, 150
 - przez referencję, 348, 349, 556
 - przez wartość, 348, 556
 - szablonów, 688
 - obiektów, 556
 - przenoszenie, 108, 506, 530, 537
 - elementów, 720
 - kontenerów, 110
 - przenośność, 200
 - przesłanianie, 610, 611, 1104
 - funkcji, 608, 674
 - funkcji wirtualnych, 657
 - przestrzeganie niezmienników, 389
 - przestrzenie nazw, 89, 419
 - a przeciążanie, 439
 - aliasy, 436
 - anonimowe, 444
 - otwarte, 427
 - składanie, 436
 - wybieranie, 438
 - zagnieżdżanie, 443
 - przestrzeń nazw
 - std, 871
 - this_thread, 1217
 - przesyłanie znacznika, 159
 - przyjaciele, 594, 597, 698
 - przypisanie, 411, 905
 - basic_string, 1046
 - kopiujące, 109, 506, 538
 - przenoszące, 111, 506
 - valarray, 1167
 - przyrostki, 182
 - punkt dostosowywania, customization point, 751
- ## R
- rady dla programistów, 53
 - RAII, 100, 510
 - raportowanie błędów, 882
 - rdzenne elementy języka, 74
 - rdzeń, 1193
 - redukcja mapy, 1250
 - redundancje, 467
 - referencje, 344
 - regularność, 720
 - reguła
 - jednej definicji, 453
 - Maksa Muncha, 290
 - rekurencja, 805, 806
 - relacje między typami, 1024
 - replikacja klasy, 656
 - reprezentacja
 - klasy String, 586
 - kontenera, 896
 - pamięci, 409
 - rodzaje składowych, 691
 - rozkład liczb losowych, 1180, 1185
 - rozkłady
 - Bernoulliego, 1186
 - normalne, 1187, 1188
 - Poissona, 1187
 - próbkowe, 1188
 - równomierne, 1181, 1186
 - rozmiar basic_string, 1042
 - rozmiary obiektów, 184
 - rozstrzyganie
 - dwuznaczności, 648
 - niejednoznaczności, 646, 705
 - przeciążeń, 708
 - rozwiązywanie równań liniowych, 858
 - rozwijanie, inlining, 45
 - równania liniowe, 858
 - równoległe wyszukiwanie, 1247
 - różnice między językami, 1274
 - RTTI, 670, 678, 679
 - rzutowanie, 655, 1015
 - dynamiczne, 661, 663, 665
 - nazwane, 333
 - referencji, 663
 - statyczne, 665
 - w dół, 661
 - w górę, 661
 - w stylu C, 334
 - w stylu funkcyjnym, 334

S

- sekwencja, 936, 959
 - sekwencyjne przeszukiwanie, 953
 - semantyka własności na
 - wylącznieść, 1221
 - silna gwarancja, 384
 - składanie
 - kodu, 435
 - przestrzeni nazw, 436
 - struktur danych, 781
 - składnia lokacyjna, 316
 - składnik typu zwykłego, 533
 - składniki biblioteki, 58
 - składniki biblioteki
 - standardowej, 74, 1269
 - składowe, 84, 598
 - bazy, 631
 - chronione, 624, 625
 - klasy atomic, 1205
 - klasy bazowej, 614
 - klasy char_traits, 1037
 - klasy collate, 1129
 - klasy locale, 1114
 - klasy ostream, 1088
 - klasy tuple, 988
 - przestrzeni nazw, 422
 - statyczne, 492, 692
 - szablonu klasy, 691
 - zegara, 1017
 - skrótowe określanie wartości, 292
 - skrótowy klas znaków, 1056
 - słaba kontrola typów, 689
 - słaby licznik, 997
 - słowa kluczowe, 191, 1275
 - słownik, 897, 917
 - słownik nieuporządkowany, 897, 923
 - słowo kluczowe
 - const, 78, 297, 348
 - constexpr, 78
 - final, 612
 - friend, 594
 - mutable, 488
 - this, 328, 490
 - virtual, 607
 - volatile, 1207
 - sortowanie, 700, 950
 - łańcucha, 952
 - obiektów, 677
 - specjalizacja, 687, 735
 - atomic, 1202
 - częściowa, 746
 - implementacji, 748
 - interfejsu, 747
 - jawna, 756
 - kompletna, 745
 - nie będąca przeciążaniem, 752
 - szablonu funkcji, 750
 - użytkownika, 744, 756
 - wygenerowana, 756
 - specyfikacje wyjątków, 60, 397
 - specyfikator
 - decltype(), 199
 - listy {}, 199
 - override, 611
 - typu auto, 197
 - volatile, 1207
 - sprawdzanie
 - definicji szablonu, 731
 - ograniczeń szablonu, 726, 730
 - typów, 1257
 - stałe, 77, 78
 - formatowania fmtflags, 1094
 - funkcje składowe, 487
 - składowe iostate, 1090
 - składowe klasy basic_regex, 1059
 - składowe openmode, 1091
 - składowe seekdir, 1091
 - symboliczne, 297
 - stałość
 - fizyczna, 488
 - logiczna, 488
 - stały czas wykonywania, 902
 - stan strumienia, 1089
 - stan współdzielony, 1236
 - standard
 - C++11, 62, 1268
 - ISO, 169
 - standardowe
 - hierarchia wyjątków, 878
 - funkcje matematyczne, 1163
 - stany strumieni, 1081
 - starter wątków, 1245
 - statyczna
 - asercja, 92
 - kontrola typów, 680
 - sterowanie konkretyzacją, 758
 - sterta, 83, 99, 955
 - stos, 929
 - stosowanie języka C++, 65
 - strażnik
 - dla muteksu, 1220, 1225
 - dla obiektu, 1227
 - dołączania, include guard, 468
 - string, 914
 - strona kodowa, 1117
 - struktura, struct, 83, 234, 480
 - a klasa, 237
 - a tablica, 239
 - array, 898
 - cechująca, 800
 - deklaracji, 188
 - forward_list, 915
 - nazwy, 236
 - składowa, 235
 - sterująca, 802
 - Tuple, 815, 816
- strumienie
 - łańcuchowe, 1079
 - plikowe, 1078
 - wejścia i wyjścia, 1075, 1077
 - formatowanie, 1094
 - strumień
 - fstream, 1078
 - iostream, 1076, 1084
 - ostream, 1075, 1105
 - stringstream, 1080
 - wejścia, 127, 1075, 1106
 - wyjścia, 126, 1075, 1105
 - strzałka, 576
 - styl programowania, 43
 - symbole zastępcze, 973
 - synchronizacja, 1207
 - system, 660
 - szablon
 - atomic, 1201
 - conditional, 795
 - funkcji, 716
 - Io, 669
 - iterator, 964
 - klasy basic_string, 685
 - klasy, class template, 686
 - łańcucha, 684
 - macierzy, 841
 - cięcie, 843
 - indeksowanie, 843
 - konstrukcja, 842
 - przypisywanie, 842
 - numeric_limits, 1160
 - podstawowy, 748
 - szablonowy operator literałowy, 583
 - szablony, 61, 113, 681
 - argumenty, 736
 - konkretyzacja, 756
 - konwersje, 778

parametry, 736
 przekazywanie, 825
 przestrzenie nazw, 767
 szablony funkcji, 115, 685, 699
 argumenty, 701
 dedukcja argumentów, 702
 dedukcja referencji, 703
 przeciążanie, 704
 rozstrzygnięcie
 niejednoznaczności, 705
 szablony
 jako argumenty, 742
 jako interfejsy, 780
 składowe, 694
 wyrażenia, 864
 zmienne, 118, 821
 sztuczka Bartona-Nackmana, 785
 szukanie sekwencji, 942

Ś

ściśle uporządkowanie słabe, 899

T

tablica, 80, 313, 914, 978
 funkcji wirtualnych, 103, 609
 symboli, 462
 valarray, 1166
 technika RTTI, 670
 techniki
 abstrakcji, 37, 473
 wykorzystania znaczników, 963
 testowanie
 implementacji funkcji, 733
 kodu, 861
 testy ograniczeń, 726
 tłumienie operacji, 113
 tokeny, 290, 1071
 tożsamość, 200
 tryb otwierania strumieni, 1091
 tryby
 plikowe, 1254
 strumieniowe, 1079
 tworzenie
 interfejsów, 657
 iteratorów przenoszących, 969
 nowych lokalizacji, 1118
 obiektów, 506
 wątku, 1212
 wstawiaczy, 968

typ, 48, 76, 172
 arytmetyczny, 97
 C, 720
 całkowitoliczbowy
 ze znakiem, 722
 duration, 1012, 1013
 faset, 1120
 function, 974
 future, 1236, 1241
 gslice, 1175
 gslice_array, 1176
 hash, 923
 lambda, 329
 logiczny, 173
 mapowany, 134
 packaged_task, 155, 1238
 pair, 161
 polimorficzny, 609
 promise, 1236, 1237
 regularny, 720
 Semiregular, 725
 shared_future, 1244
 string, 1035, 1039, 1040
 T, 331, 688, 725
 time_point, 1015, 1016
 tuple, 161, 827
 valarray, 1166
 vector, 165, 911
 void, 183
 wspólny, 1028
 wyliczenia, underlying type,
 250
 zamknięciowy, 329
 typy
 abstrakcyjne, 101
 atomowe, 1201
 błędów systemowych, 883
 całkowitoliczbowe, 179
 daty i czasu, 1261
 generowane, 776
 iteratorów, 140
 jako argumenty, 736
 konkretne, 96, 495, 503
 literałów, 297
 literałów
 całkowitoliczbowych, 180
 parametryzowane, 45, 114
 podstawowe, 172
 polimorficzne, 102
 powiązane, 692
 składowe, 494, 693, 904

użytkownika, 82, 173, 555
 wartościowe, 503
 wbudowane, 82, 173, 540
 wygenerowane, 689
 zegarów, 1017
 zmiennoprzecinkowe, 181
 znakowe, 174
 znakowe bez znaku, 176
 zwrotne, 617

U

uchwyt, 312
 do danych, 100
 do zasobów, 108
 układ tablicy w pamięci, 777
 unia, union, 233, 244
 a klasa, 246
 anonimowa, 247, 588
 znacznikowa, 248
 unikanie wyścigów do danych,
 1220
 uogólnianie, 714, 717, 718
 uogólnione algorytmy
 numeryczne, 1176
 uporządkowanie sekwencyjnie
 spójne, 1196
 urządzenie losowe, 1184
 usuwacz, 993
 usuwacz zmiennej lokalnej, 990
 usuwanie funkcji, 547
 usuwanie obiektu, 639
 użycie
 aliasów, 797
 alokatorów, 697
 dynamic_cast, 679
 Enable_if, 809, 812
 hierarchii klas, 774
 klasy bazowej, 649
 list inicjacyjnych, 521
 metaprogramowania, 806
 muteksów, 1222
 nagłówków, 466
 operacji domyślnych, 543
 plików nagłówkowych, 459
 RTTI, 678
 składowych chronionych, 625
 składowych klasy bazowej, 614
 szablonów klas, 774
 wiadomości, 1154
 wyjątków, 379

V

vector, 130

W

wada mechanizmu szablonów, 688

wartości, 76, 200

jako argumenty, 738

lewostronne, 45, 200

prawostronne, 45, 200

wartość

jednostki, 831

result, 1149

warunki końcowe, 362

warunki wstępne, 362

warunkowe obliczanie wartości, 345

wątek, thread, 149, 404, 1191, 1209

dane lokalne, 1218

kończenie działania, 1231

likwidowanie, 1218

niszczenie, 1213

starter, 1245

systemowy, 1211

tworzenie, 1212

uruchamianie, 1245

uśpiony, 1210, 1225

zablokowany, 1210

zagliodzenie, 1221

zmienne warunkowe, 1231

wciążenia, 269

wczytywanie danych, 278

wejścia klasy basic_string, 1044

wejście, 127, 873, 1075, 1106

języka C, 1257

niesformatowane, 1084, 1086

niskopoziomowe, 282

numeryczne, 1134

sformatowane, 1083

wektor, 405, 897, 910

wersjonowanie, 441

wewnątrzklasowe definicje

funkcji, 486

węzły, 781, 788

wiadomości, 1151

wiązanie

dynamiczne, 670

nazw, 756, 759

typów, 800

w miejscu definicji, 762

w miejscu konkretyzacji, 763

wewnętrzne, 449

widziet, 660

wielodziedziczenie, 59, 626, 638,

644, 664

wielozbiór, 136

wirtualne

funkcje wyjściowe, 1088

klasy bazowe, 651

wirtualność, 695

wizytatorzy, 670, 673

wskaźnik, 80, 990

shared_ptr, 146, 993, 994

unique_ptr, 146, 990

void*, 1260

weak_ptr, 996

wskaźniki

do funkcji, 364, 458

do składowych, 627–630

wykrywalne, 1006

wsparcie

dla języka, 874

dla list inicjacyjnych, 876

dla pętli for, 876

wspólne

implementacje, 644

interfejsy, 644

używanie danych, 151

współbieżność, 148, 470, 874,

1191, 1209

współbieżność zadaniowa, 1235

współczynnikapełnienia, 928

wstawiacz, inserter, 968

wybiezanie

funkcji, 799

funkcji przeciążonych, 358,

361

typu, 802, 804

wyciek pamięci, 850, 994, 1172

wycinki

macierzy, 850

tablic, 1172

uogólnione, 1175

wydajność, 381, 696

wyjątek, 90, 317, 374, 381, 877

bad_typeid, 677

system_error, 886

wyjątki

biblioteki standardowej, 877

nie będące błędami, 378

wyjścia klasy basic_string, 1044

wyjście, 126, 128, 873, 1075, 1105

języka C, 1257

numeryczne, 1132

wykrywanie

błędów, 690

wyścigów, 1197

wyliczenia, 86, 233, 249

anonimowe, 254

zwykle, 253

wymagania, 721

wymazywanie typów, type

erasure, 747

wyniki dopasowywania, 1061

wyrażenia, 273

lambda, 117, 322, 340

regularne, 161, 1053

stałe, 79, 295, 529

stałe adresowe, 299

warunkowe, 307

wyrównanie, 185

wyrównanie w pamięci, 1027

wysyłanie kwot pieniężnych, 1139

wyszukiwanie, 950

binarne, 952

wg argumentów, 425, 768

wyrażień regularnych, 1066

wyścig do danych, 1197, 1219

wywoływanie, 329

destruktorów, 511

funkcji, 574

konstruktorów, 511

składowej, 654

wzorzec, 1055

wzorzec wizytatora, 675, 787

Z

zadanie, task, 1191, 1209

zagliodzenie wątku, 1221

zagnieżdżanie, 695, 912

przestrzeni nazw, 443

typów, 697

zajęcie muteksu, 1221

zakleszczenie, 1228

zakres

funkcji, 192

globalny, 192

instrukcji, 192

klasowy, 192

lokalny, 191

przestrzeni nazw, 192

- zalety derywacji, 637
- zamienianie
 - kodów błędów, 887
 - wyrażeń regularnych, 1067
- zamknięcie, closure, 863
- zamortyzowany koszt liniowy, 903
- zamykanie
 - plików, 1253
 - programu, 402, 470, 881
- zapełnienie, 927, 928
- zapis strumienia, 1108
- zapytania o właściwości typów, 1024
- zarządzanie
 - pamięcią, 311, 990
 - zasobami, 112, 146, 385, 510
- zasady
 - konsolidacji, 711
 - mieszania, 928
- zasoby, 509, 977
- zastosowania
 - faset, 1125
 - lambda, 325
 - lokalizacji, 1125
 - macierzy, 838
- zaśmiecanie przestrzeni nazw, 253
- zbiór, 136, 917
- zdarzenie, 153
- zdarzenie asynchroniczne, 378
- zegar, 1013, 1017
- zerowy narzut, zero overhead, 43
- zestaw znaków, 171
- zestawienie
 - instrukcji, 258
 - operacji, 901
 - operatorów, 287–289
 - standardowych kontenerów, 136
 - tokenów, 290
 - znaków, 178
- zgłaszanie wyjątków, 317, 394
- zgodność C i C++, 875, 1271
- złożone
 - operatory przypisania, 560
 - predykaty typów, 1021
- złożoność
 - algorytmów, 939
 - logarytmiczna, 903
- zmienianie
 - kolejności instrukcji, 1195
 - rozmiaru, 413
- zmienna condition_variable, 1235
- zmiennne, 76
 - lambda, 328
 - lokalne, 346
 - stałe, const, 487
 - warunkowe, 1220, 1231
- znaczenie
 - klas konkretnych, 503
 - kopiowania, 533
 - operatorów, 555
- znajdowanie
 - elementów, 1049
 - elementu centralnego, 860
 - przyjaciół, 596
- znaki, 177, 873
- znaki specjalne, 1054, 1057
- zstępowanie rekurencyjne, 274
- zwolnienie muteksu, 1221
- zwracanie wartości, 150, 329, 340

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

POZNAJ NOWOŚCI C++ 11!

C++ dzielnie broni swojej pozycji na rynku języków programowania. Pomimo silnego naporu języka Java oraz platformy .NET wciąż jest niezastąpiony w wielu dziedzinach. Jeżeli tylko wymagana jest najwyższa wydajność, dostęp do sprzętu oraz przewidywalny czas wykonania, programiści najczęściej wybierają właśnie język C++. Ostatnia wersja standardu — oznaczona numerem 11 — pozwala na jeszcze łatwiejsze pisanie kodu oraz tworzenie szybszych i wydajniejszych programów.

Najnowsze wydanie tej cenionej książki zostało poprawione i uzupełnione o nowości z tej właśnie wersji standardu języka C++. Dowiesz się, jak korzystać ze wskaźników, liczb losowych oraz udoskonalonych kontenerów. Ponadto poznasz najlepsze zastosowanie wyrażań lambda czy szablonów. Oprócz omówienia nowości znajdziesz tu również szczegółowy przegląd klasycznych elementów języka C++. Pętle, zmienne, tablice, instrukcje warunkowe — to tylko niektóre z omawianych zagadnień. Książka stanowi doskonały podręcznik dla początkujących programistów i świetne rozwinięcie dla programujących w języku C++ na co dzień. Przekonaj się, jak łatwo i przyjemnie możesz opanować ten popularny język oprogramowania.

Dzięki tej książce:

- poznasz nowości wprowadzone w C++ 11
- zaznajomisz się z elementami biblioteki standardowej
- opanujesz podstawowy model pamięci języka C++
- zrozumiesz model pamięci języka C++

 Helion	<i>Sprawdź nasze szkolenia!</i> SZKOLENIA  AKADEMIA IT & BUSINESS HELIONSZKOLENIA.PL	KOD KORZYŚCI Sięgnij po więcej! ▶  ISBN 978-83-283-8329-6  9 788328 383296
 helion.pl		
 HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl		
INFORMATYKA W NAJLEPSZYM WYDANIU		Cena: 199,00 zł

 **Addison-Wesley**
Pearson Education