

BJARNE STROUSTRUP

Zaczerpnij wiedzę o C++ od samego twórcy tego języka!



Programowanie

*Teoria i praktyka
z wykorzystaniem C++*

Wydanie II poprawione


Addison
Wesley

Jak zacząć pracę w zintegrowanym środowisku programistycznym?
Jak profesjonalnie tworzyć programy użytkowe?
Jak korzystać z biblioteki graficznego interfejsu użytkownika?

Helion


Tytuł oryginału: Programming: Principles and Practice Using C++, First Edition

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-246-7720-7

Authorized translation from the English language edition, entitled: Programming: Principles and Practice Using C++, First Edition, ISBN 0321543726, by Bjarne Stroustrup, published by Pearson Education, Inc, publishing as Addison Wesley, Copyright © 2009 by Pearson Education, Inc.

Polish language edition published by Helion S.A.
Copyright © 2013.

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.

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

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

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

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

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/pcppt2>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przyklady/pcppt2.zip>

Printed in Poland.

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

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

Spis treści

Wstęp	19
Słowo do studentów	21
Słowo do nauczycieli	22
Pomoc	23
Podziękowania	23
Uwagi do czytelnika	25
0.1. Struktura książki	26
0.1.1. Informacje ogólne	27
0.1.2. Ćwiczenia, praca domowa itp.	28
0.1.3. Po przeczytaniu tej książki	29
0.2. Filozofia nauczania i uczenia się	29
0.2.1. Kolejność tematów	32
0.2.2. Programowanie a język programowania	34
0.2.3. Przenośność	34
0.3. Programowanie a informatyka	35
0.4. Kreatywność i rozwiązywanie problemów	35
0.5. Uwagi dla autorów	35
0.6. Bibliografia	36
0.7. Noty biograficzne	37
Bjarne Stroustrup	37
Lawrence „Pete” Petersen	38

Rozdział 1. Komputery, ludzie i programowanie	39
1.1. Wstęp	40
1.2. Oprogramowanie	40
1.3. Ludzie	42
1.4. Informatyka	45
1.5. Komputery są wszędzie	46
1.5.1. Komputery z ekranem i bez	46
1.5.2. Transport	47
1.5.3. Telekomunikacja	48
1.5.4. Medycyna	50
1.5.5. Informacja	51
1.5.6. Sięgamy w kosmos	52
1.5.7. I co z tego	53
1.6. Ideały dla programistów	54
Część I Podstawy	61
Rozdział 2. Witaj, świecie!	63
2.1. Programy	64
2.2. Klasyczny pierwszy program	64
2.3. Kompilacja	67
2.4. Łączenie	69
2.5. Środowiska programistyczne	70
Rozdział 3. Obiekty, typy i wartości	77
3.1. Dane wejściowe	78
3.2. Zmienne	80
3.3. Typy danych wejściowych	81
3.4. Operacje i operatory	82
3.5. Przypisanie i inicjacja	85
3.5.1. Przykład wykrywania powtarzających się słów	87
3.6. Złożone operatory przypisania	89
3.6.1. Przykład zliczania powtarzających się słów	89
3.7. Nazwy	90
3.8. Typy i obiekty	92
3.9. Kontrola typów	94
3.9.1. Konwersje bezpieczne dla typów	95
3.9.2. Konwersje niebezpieczne dla typów	96

Rozdział 4. Wykonywanie obliczeń	103
4.1. Wykonywanie obliczeń	104
4.2. Cele i narzędzia	105
4.3. Wyrażenia	107
4.3.1. Wyrażenia stałe	108
4.3.2. Operatory	109
4.3.3. Konwersje	111
4.4. Instrukcje	112
4.4.1. Selekcja	113
4.4.2. Iteracja	119
4.5. Funkcje	122
4.5.1. Po co zaprzętać sobie głowę funkcjami	124
4.5.2. Deklarowanie funkcji	125
4.6. Wektor	126
4.6.1. Powiększanie wektora	127
4.6.2. Przykład wczytywania liczb do programu	128
4.6.3. Przykład z użyciem tekstu	130
4.7. Właściwości języka	132

Rozdział 5. Błędy	139
5.1. Wstęp	140
5.2. Źródła błędów	141
5.3. Błędy kompilacji	142
5.3.1. Błędy składni	142
5.3.2. Błędy typów	143
5.3.3. Nie błędy	144
5.4. Błędy konsolidacji	145
5.5. Błędy czasu wykonania	146
5.5.1. Rozwiązywanie problemu przez wywołującego	147
5.5.2. Rozwiązywanie problemu przez wywołowanego	148
5.5.3. Raportowanie błędów	149
5.6. Wyjątki	151
5.6.1. Nieprawidłowe argumenty	151
5.6.2. Błędy zakresu	152
5.6.3. Nieprawidłowe dane wejściowe	154
5.6.4. Błędy zawężania zakresu	156
5.7. Błędy logiczne	157
5.8. Szacowanie	159
5.9. Debugowanie	161
5.9.1. Praktyczna rada dotycząca debugowania	162
5.10. Warunki wstępne i końcowe	165
5.10.1. Warunki końcowe	167
5.11. Testowanie	168

Rozdział 6. Pisanie programu	175
6.1. Problem	176
6.2. Przemyślenie problemu	176
6.2.1. Etapy rozwoju oprogramowania	177
6.2.2. Strategia	177
6.3. Wracając do kalkulatora	179
6.3.1. Pierwsza próba	180
6.3.2. Tokeny	182
6.3.3. Implementowanie tokenów	183
6.3.4. Używanie tokenów	185
6.3.5. Powrót do tablicy	186
6.4. Gramatyki	188
6.4.1. Dygresja — gramatyka języka angielskiego	192
6.4.2. Pisanie gramatyki	193
6.5. Zamiana gramatyki w kod	194
6.5.1. Implementowanie zasad gramatyki	194
6.5.2. Wyrażenia	195
6.5.3. Składniki	199
6.5.4. Podstawowe elementy wyrażeń	200
6.6. Wypróbowywanie pierwszej wersji	201
6.7. Wypróbowywanie drugiej wersji	205
6.8. Strumienie tokenów	206
6.8.1. Implementacja typu <code>Token_stream</code>	207
6.8.2. Wczytywanie tokenów	209
6.8.3. Wczytywanie liczb	210
6.9. Struktura programu	211
Rozdział 7. Kończenie programu	217
7.1. Wprowadzenie	218
7.2. Wejście i wyjście	218
7.3. Obsługa błędów	220
7.4. Liczby ujemne	224
7.5. Reszta z dzielenia	225
7.6. Oczyszczanie kodu	227
7.6.1. Stałe symboliczne	227
7.6.2. Użycie funkcji	229
7.6.3. Układ kodu	230
7.6.4. Komentarze	231
7.7. Odzyskiwanie sprawności po wystąpieniu błędu	233
7.8. Zmienne	236
7.8.1. Zmienne i definicje	236
7.8.2. Wprowadzanie nazw	240
7.8.3. Nazwy predefiniowane	242
7.8.4. Czy to już koniec?	243

Rozdział 8. Szczegóły techniczne — funkcje itp.	247
8.1. Szczegóły techniczne	248
8.2. Deklaracje i definicje	249
8.2.1. Rodzaje deklaracji	252
8.2.2. Deklaracje stałych i zmiennych	252
8.2.3. Domyślna inicjacja	254
8.3. Pliki nagłówkowe	254
8.4. Zakres	256
8.5. Wywoływanie i wartość zwrotna funkcji	261
8.5.1. Deklarowanie argumentów i typu zwrotnego	261
8.5.2. Zwracanie wartości	263
8.5.3. Przekazywanie przez wartość	264
8.5.4. Przekazywanie argumentów przez stałą referencję	265
8.5.5. Przekazywanie przez referencję	267
8.5.6. Przekazywanie przez wartość a przez referencję	269
8.5.7. Sprawdzanie argumentów i konwersja	271
8.5.8. Implementacja wywołań funkcji	272
8.6. Porządek wykonywania instrukcji	276
8.6.1. Wartościowanie wyrażeń	277
8.6.2. Globalna inicjacja	277
8.7. Przestrzenie nazw	279
8.7.1. Dyrektywy i deklaracje using	280
Rozdział 9. Szczegóły techniczne — klasy itp.	287
9.1. Typy zdefiniowane przez użytkownika	288
9.2. Klasy i składowe klas	289
9.3. Interfejs i implementacja	289
9.4. Tworzenie klas	291
9.4.1. Struktury i funkcje	291
9.4.2. Funkcje składowe i konstruktory	293
9.4.3. Ukrywanie szczegółów	294
9.4.4. Definiowanie funkcji składowych	296
9.4.5. Odwoływanie się do bieżącego obiektu	298
9.4.6. Raportowanie błędów	299
9.5. Wyliczenia	300
9.6. Przeciążanie operatorów	302
9.7. Interfejsy klas	303
9.7.1. Typy argumentów	304
9.7.2. Kopiowanie	306
9.7.3. Konstruktory domyślne	306
9.7.4. Stałe funkcje składowe	309
9.7.5. Składowe i funkcje pomocnicze	310
9.8. Klasa Date	312

Część II Wejście i wyjście **319**

Rozdział 10. Strumienie wejścia i wyjścia **321**

10.1. Wejście i wyjście	322
10.2. Model strumieni wejścia i wyjścia	323
10.3. Pliki	325
10.4. Otwieranie pliku	326
10.5. Odczytywanie i zapisywanie plików	328
10.6. Obsługa błędów wejścia i wyjścia	330
10.7. Wczytywanie pojedynczej wartości	332
10.7.1. Rozłożenie problemu na mniejsze części	334
10.7.2. Oddzielenie warstwy komunikacyjnej od funkcji	337
10.8. Definiowanie operatorów wyjściowych	338
10.9. Definiowanie operatorów wejściowych	339
10.10. Standardowa pętla wejściowa	340
10.11. Wczytywanie pliku strukturalnego	341
10.11.1. Reprezentacja danych w pamięci	342
10.11.2. Odczytywanie struktur wartości	343
10.11.3. Zmienianie reprezentacji	347

Rozdział 11. Indywidualizacja operacji wejścia i wyjścia **353**

11.1. Regularność i nieregularność	354
11.2. Formatowanie danych wyjściowych	354
11.2.1. Wysyłanie na wyjście liczb całkowitych	355
11.2.2. Przyjmowanie na wejściu liczb całkowitych	356
11.2.3. Wysyłanie na wyjście liczb zmiennoprzecinkowych	357
11.2.4. Precyzja	358
11.2.5. Pola	360
11.3. Otwieranie plików i pozycjonowanie	361
11.3.1. Tryby otwierania plików	361
11.3.2. Pliki binarne	362
11.3.3. Pozycjonowanie w plikach	365
11.4. Strumienie łańcuchowe	365
11.5. Wprowadzanie danych wierszami	367
11.6. Klasyfikowanie znaków	368
11.7. Stosowanie niestandardowych separatorów	370
11.8. Zostało jeszcze tyle do poznania	376

Rozdział 12. Model graficzny **381**

12.1. Czemu grafika?	382
12.2. Model graficzny	383
12.3. Pierwszy przykład	384

12.4. Biblioteka GUI	387
12.5. Współrzędne	388
12.6. Figury geometryczne	388
12.7. Używanie klas figur geometrycznych	389
12.7.1. Nagłówki graficzne i funkcja main	390
12.7.2. Prawie puste okno	390
12.7.3. Klasa Axis	392
12.7.4. Rysowanie wykresu funkcji	394
12.7.5. Wielokąty	394
12.7.6. Prostokąty	395
12.7.7. Wypełnianie kolorem	397
12.7.8. Tekst	398
12.7.9. Obrazy	399
12.7.10. Jeszcze więcej grafik	400
12.8. Uruchamianie programu	401
12.8.1. Pliki źródłowe	402

Rozdział 13. Klasy graficzne **407**

13.1. Przegląd klas graficznych	408
13.2. Klasy Point i Line	410
13.3. Klasa Lines	412
13.4. Klasa Color	414
13.5. Typ Line_style	416
13.6. Typ Open_polyline	418
13.7. Typ Closed_polyline	419
13.8. Typ Polygon	420
13.9. Typ Rectangle	422
13.10. Wykorzystywanie obiektów bez nazw	426
13.11. Typ Text	428
13.12. Typ Circle	430
13.13. Typ Ellipse	431
13.14. Typ Marked_polyline	433
13.15. Typ Marks	434
13.16. Typ Mark	435
13.17. Typ Image	436

Rozdział 14. Projektowanie klas graficznych **443**

14.1. Zasady projektowania	444
14.1.1. Typy	444
14.1.2. Operacje	445
14.1.3. Nazewnictwo	446
14.1.4. Zmienność	448

14.2. Klasa Shape	448
14.2.1. Klasa abstrakcyjna	450
14.2.2. Kontrola dostępu	451
14.2.3. Rysowanie figur	454
14.2.4. Kopiowanie i zmienność	456
14.3. Klasy bazowe i pochodne	458
14.3.1. Układ obiektu	459
14.3.2. Tworzenie podklas i definiowanie funkcji wirtualnych	461
14.3.3. Przesłanianie	461
14.3.4. Dostęp	463
14.3.5. Czyste funkcje wirtualne	464
14.4. Zalety programowania obiektowego	465

Rozdział 15. Graficzne przedstawienie funkcji i danych **471**

15.1. Wprowadzenie	472
15.2. Rysowanie wykresów prostych funkcji	472
15.3. Typ Function	476
15.3.1. Argumenty domyślne	477
15.3.2. Więcej przykładów	478
15.4. Typ Axis	479
15.5. Wartość przybliżona funkcji wykładniczej	481
15.6. Przedstawianie danych na wykresach	486
15.6.1. Odczyt danych z pliku	487
15.6.2. Układ ogólny	488
15.6.3. Skalowanie danych	489
15.6.4. Budowanie wykresu	490

Rozdział 16. Graficzne interfejsy użytkownika **497**

16.1. Różne rodzaje interfejsów użytkownika	498
16.2. Przycisk Next	499
16.3. Proste okno	500
16.3.1. Funkcje zwrotne	501
16.3.2. Pętla oczekująca	504
16.4. Typ Button i inne pochodne typu Widget	505
16.4.1. Widżety	505
16.4.2. Przyciski	506
16.4.3. Widżety In_box i Out_box	507
16.4.4. Menu	507
16.5. Przykład	508
16.6. Inwersja kontroli	511
16.7. Dodawanie menu	513
16.8. Debugowanie kodu GUI	517

Część III Dane i algorytmy 523

Rozdział 17. Wektory i pamięć wolna 525

17.1. Wprowadzenie	526
17.2. Podstawowe wiadomości na temat typu vector	527
17.3. Pamięć, adresy i wskaźniki	529
17.3.1. Operator sizeof	531
17.4. Pamięć wolna a wskaźniki	532
17.4.1. Alokacja obiektów w pamięci wolnej	533
17.4.2. Dostęp poprzez wskaźniki	534
17.4.3. Zakresy	535
17.4.4. Inicjacja	536
17.4.5. Wskaźnik zerowy	537
17.4.6. Dealokacja pamięci wolnej	538
17.5. Destruktory	540
17.5.1. Generowanie destruktorów	542
17.5.2. Destruktory a pamięć wolna	542
17.6. Dostęp do elementów	544
17.7. Wskaźniki na obiekty klas	545
17.8. Babranie się w typach — void* i rzutowanie	546
17.9. Wskaźniki i referencje	548
17.9.1. Wskaźniki i referencje jako parametry	549
17.9.2. Wskaźniki, referencje i dziedziczenie	550
17.9.3. Przykład — listy	551
17.9.4. Operacje na listach	552
17.9.5. Zastosowania list	554
17.10. Wskaźnik this	555
17.10.1. Więcej przykładów użycia typu Link	557

Rozdział 18. Wektory i tablice 563

18.1. Wprowadzenie	564
18.2. Kopiowanie	564
18.2.1. Konstruktory kopiujące	566
18.2.2. Przypisywanie z kopiowaniem	567
18.2.3. Terminologia związana z kopiowaniem	569
18.3. Podstawowe operacje	570
18.3.1. Konstruktory jawne	571
18.3.2. Debugowanie konstruktorów i destruktorów	573
18.4. Uzyskiwanie dostępu do elementów wektora	575
18.4.1. Problem stałych wektorów	576
18.5. Tablice	577
18.5.1. Wskaźniki na elementy tablicy	578
18.5.2. Wskaźniki i tablice	580

18.5.3. Inicjowanie tablic	582
18.5.4. Problemy ze wskaźnikami	583
18.6. Przykłady — palindrom	586
18.6.1. Wykorzystanie łańcuchów	586
18.6.2. Wykorzystanie tablic	587
18.6.3. Wykorzystanie wskaźników	588

Rozdział 19. Wektory, szablony i wyjątki **593**

19.1. Analiza problemów	594
19.2. Zmienianie rozmiaru	596
19.2.1. Reprezentacja	597
19.2.2. Rezerwacja pamięci i pojemność kontenera	598
19.2.3. Zmienianie rozmiaru	599
19.2.4. Funkcja <code>push_back()</code>	599
19.2.5. Przypisywanie	600
19.2.6. Podsumowanie dotychczasowej pracy nad typem <code>vector</code>	602
19.3. Szablony	602
19.3.1. Typy jako parametry szablonów	603
19.3.2. Programowanie ogólne	605
19.3.3. Kontenery a dziedziczenie	607
19.3.4. Liczby całkowite jako parametry szablonów	608
19.3.5. Dedukcja argumentów szablonu	610
19.3.6. Uogólnianie wektora	610
19.4. Sprawdzanie zakresu i wyjątki	613
19.4.1. Dygresja — uwagi projektowe	614
19.4.2. Wyznanie na temat makr	615
19.5. Zasoby i wyjątki	617
19.5.1. Potencjalne problemy z zarządzaniem zasobami	617
19.5.2. Zajmowanie zasobów jest inicjacją	619
19.5.3. Gwarancje	620
19.5.4. Obiekt <code>auto_ptr</code>	621
19.5.5. Technika RAII dla wektora	622

Rozdział 20. Kontenery i iteratory **629**

20.1. Przechowywanie i przetwarzanie danych	630
20.1.1. Praca na danych	630
20.1.2. Uogólnianie kodu	631
20.2. Ideały twórcy biblioteki STL	634
20.3. Sekwencje i iteratory	637
20.3.1. Powrót do przykładu	639
20.4. Listy powiązane	641
20.4.1. Operacje list	642
20.4.2. Iteracja	643
20.5. Jeszcze raz uogólnianie wektora	645

20.6. Przykład — prosty edytor tekstu	647
20.6.1. Wiersze	649
20.6.2. Iteracja	650
20.7. Typy vector, list oraz string	653
20.7.1. Funkcje insert() i erase()	654
20.8. Dostosowanie wektora do biblioteki STL	656
20.9. Dostosowywanie wbudowanych tablic do STL	658
20.10. Przegląd kontenerów	660
20.10.1. Kategorie iteratorów	662

Rozdział 21. Algorytmy i słowniki **667**

21.1. Algorytmy biblioteki standardowej	668
21.2. Najprostszy algorytm — find()	669
21.2.1. Kilka przykładów z programowania ogólnego	670
21.3. Ogólny algorytm wyszukiwania — find_if()	671
21.4. Obiekty funkcyjne	673
21.4.1. Abstrakcyjne spojrzenie na obiekty funkcyjne	674
21.4.2. Predykaty składowych klas	675
21.5. Algorytmy numeryczne	676
21.5.1. Akumulacja	677
21.5.2. Uogólnianie funkcji accumulate()	678
21.5.3. Iloczyn skalarny	679
21.5.4. Uogólnianie funkcji inner_product()	681
21.6. Kontenery asocjacyjne	681
21.6.1. Słowniki	682
21.6.2. Opis ogólny kontenera map	684
21.6.3. Jeszcze jeden przykład zastosowania słownika	687
21.6.4. Kontener unordered_map	689
21.6.5. Zbiory	691
21.7. Kopiowanie	693
21.7.1. Funkcja copy()	693
21.7.2. Iteratory strumieni	694
21.7.3. Utrzymywanie porządku przy użyciu kontenera set	696
21.7.4. Funkcja copy_if()	696
21.8. Sortowanie i wyszukiwanie	697

Część IV Poszerzanie horyzontów **703**

Rozdział 22. Ideały i historia **705**

22.1. Historia, ideały i profesjonalizm	706
22.1.1. Cele i filozofie języków programowania	706
22.1.2. Ideały programistyczne	708
22.1.3. Style i paradygmaty	714

22.2. Krótka historia języków programowania	717
22.2.1. Pierwsze języki	718
22.2.2. Korzenie nowoczesnych języków programowania	719
22.2.3. Rodzina Algol	724
22.2.4. Simula	731
22.2.5. C	733
22.2.6. C++	736
22.2.7. Dziś	738
22.2.8. Źródła informacji	740

Rozdział 23. Przetwarzanie tekstu **745**

23.1. Tekst	746
23.2. Łańcuchy	746
23.3. Strumienie wejścia i wyjścia	750
23.4. Słowniki	750
23.4.1. Szczegóły implementacyjne	755
23.5. Problem	757
23.6. Wyrażenia regularne	759
23.7. Wyszukiwanie przy użyciu wyrażeń regularnych	761
23.8. Składnia wyrażeń regularnych	764
23.8.1. Znaki i znaki specjalne	764
23.8.2. Rodzaje znaków	765
23.8.3. Powtórzenia	766
23.8.4. Grupowanie	767
23.8.5. Alternatywa	767
23.8.6. Zbiory i przedziały znaków	768
23.8.7. Błędy w wyrażeniach regularnych	769
23.9. Dopasowywanie przy użyciu wyrażeń regularnych	770
23.10. Źródła	775

Rozdział 24. Działania na liczbach **779**

24.1. Wprowadzenie	780
24.2. Rozmiar, precyzja i przekroczenie zakresu	780
24.2.1. Ograniczenia typów liczbowych	783
24.3. Tablice	784
24.4. Tablice wielowymiarowe w stylu języka C	785
24.5. Biblioteka Matrix	786
24.5.1. Wymiary i dostęp	787
24.5.2. Macierze jednowymiarowe	789
24.5.3. Macierze dwuwymiarowe	792
24.5.4. Wejście i wyjście macierzy	794
24.5.5. Macierze trójwymiarowe	795

24.6. Przykład — rozwiązywanie równań liniowych	796
24.6.1. Klasyczna eliminacja Gaussa	797
24.6.2. Wybór elementu centralnego	798
24.6.3. Testowanie	799
24.7. Liczby losowe	800
24.8. Standardowe funkcje matematyczne	802
24.9. Liczby zespolone	803
24.10. Źródła	804

Rozdział 25. Programowanie systemów wbudowanych **809**

25.1. Systemy wbudowane	810
25.2. Podstawy	813
25.2.1. Przewidywalność	815
25.2.2. Ideały	815
25.2.3. Życie z awarią	816
25.3. Zarządzanie pamięcią	818
25.3.1. Problemy z pamięcią wolną	819
25.3.2. Alternatywy dla ogólnej pamięci wolnej	822
25.3.3. Przykład zastosowania puli	823
25.3.4. Przykład użycia stosu	824
25.4. Adresy, wskaźniki i tablice	825
25.4.1. Niekontrolowane konwersje	825
25.4.2. Problem — źle działające interfejsy	826
25.4.3. Rozwiązanie — klasa interfejsu	829
25.4.4. Dziedziczenie a kontenery	832
25.5. Bity, bajty i słowa	834
25.5.1. Bity i operacje na bitach	835
25.5.2. Klasa bitset	839
25.5.3. Liczby ze znakiem i bez znaku	840
25.5.4. Manipulowanie bitami	844
25.5.5. Pola bitowe	846
25.5.6. Przykład — proste szyfrowanie	847
25.6. Standardy pisania kodu	851
25.6.1. Jaki powinien być standard kodowania	852
25.6.2. Przykładowe zasady	854
25.6.3. Prawdziwe standardy kodowania	859

Rozdział 26. Testowanie **865**

26.1. Czego chcemy	866
26.1.1. Zastrzeżenie	867
26.2. Dowody	867
26.3. Testowanie	867
26.3.1. Testowanie regresyjne	868
26.3.2. Testowanie jednostkowe	869

26.3.3. Algorytmy i nie-algorytmy	875
26.3.4. Testy systemowe	882
26.3.5. Testowanie klas	886
26.3.6. Znajdowanie założeń, które się nie potwierdzają	889
26.4. Projektowanie pod kątem testowania	890
26.5. Debugowanie	891
26.6. Wydajność	891
26.6.1. Kontrolowanie czasu	893
26.7. Źródła	895

Rozdział 27. Język C

899

27.1. C i C++ to rodzeństwo	900
27.1.1. Zgodność języków C i C++	901
27.1.2. Co jest w języku C++, czego nie ma w C	903
27.1.3. Biblioteka standardowa języka C	904
27.2. Funkcje	905
27.2.1. Brak możliwości przeciążania nazw funkcji	906
27.2.2. Sprawdzanie typów argumentów funkcji	906
27.2.3. Definicje funkcji	907
27.2.4. Wywoływanie C z poziomu C++ i C++ z poziomu C	909
27.2.5. Wskaźniki na funkcje	911
27.3. Mniej ważne różnice między językami	912
27.3.1. Przestrzeń znaczników struktur	912
27.3.2. Słowa kluczowe	913
27.3.3. Definicje	914
27.3.4. Rzutowanie w stylu języka C	915
27.3.5. Konwersja typu void*	916
27.3.6. Typ enum	917
27.3.7. Przestrzenie nazw	917
27.4. Pamięć wolna	918
27.5. Łańcuchy w stylu języka C	919
27.5.1. Łańcuchy w stylu języka C i const	922
27.5.2. Operacje na bajtach	922
27.5.3. Przykład — funkcja strcpy()	923
27.5.4. Kwestia stylu	923
27.6. Wejście i wyjście — nagłówek stdio	924
27.6.1. Wyjście	924
27.6.2. Wejście	925
27.6.3. Pliki	927
27.7. Stałe i makra	927
27.8. Makra	928
27.8.1. Makra podobne do funkcji	929
27.8.2. Makra składniowe	930
27.8.3. Kompilacja warunkowa	931
27.9. Przykład — kontenery intruzyjne	932

Dodatki **941****Dodatek A Zestawienie własności języka** **943**

A.1. Opis ogólny	944
A.2. Literały	946
A.3. Identyfikatory	950
A.4. Zakres, pamięć oraz czas trwania	950
A.5. Wyrażenia	953
A.6. Instrukcje	962
A.7. Deklaracje	964
A.8. Typy wbudowane	965
A.9. Funkcje	968
A.10. Typy zdefiniowane przez użytkownika	971
A.11. Wyliczenia	972
A.12. Klasy	972
A.13. Szablony	983
A.14. Wyjątki	986
A.15. Przestrzenie nazw	988
A.16. Aliasy	988
A.17. Dyrektywy preprocesora	989

Dodatek B Biblioteka standardowa **991**

B.1. Przegląd	992
B.2. Obsługa błędów	995
B.3. Iteratory	997
B.4. Kontenery	1001
B.5. Algorytmy	1008
B.6. Biblioteka STL	1016
B.7. Strumienie wejścia i wyjścia	1018
B.8. Przetwarzanie łańcuchów	1024
B.9. Obliczenia	1028
B.10. Funkcje biblioteki standardowej C	1032
B.11. Inne biblioteki	1040

Dodatek C Podstawy środowiska Visual Studio **1043**

C.1. Uruchamianie programu	1044
C.2. Instalowanie środowiska Visual Studio	1044
C.3. Tworzenie i uruchamianie programu	1044
C.4. Później	1046

Dodatek D Instalowanie biblioteki FLTK	1047
D.1. Wprowadzenie	1048
D.2. Pobieranie biblioteki FLTK z internetu	1048
D.3. Instalowanie biblioteki FLTK	1048
D.4. Korzystanie z biblioteki FLTK w Visual Studio	1049
D.5. Sprawdzanie, czy wszystko działa	1050
Dodatek E Implementacja GUI	1051
E.1. Implementacja wywołań zwrotnych	1052
E.2. Implementacja klasy Widget	1053
E.3. Implementacja klasy Window	1054
E.4. Klasa Vector_ref	1055
E.5. Przykład — widgety	1056
Słowniczek	1059
Bibliografia	1065
Skorowidz	1069
Zdjęcia	1095

Pisanie programu

Pisać program, znaczy rozumieć.

— Kristen Nygaard

Pisanie programu polega na stopniowym modyfikowaniu swojego wyobrażenia na temat tego, co się chce zrobić i jak się chce to wyrazić. W tym i następnym rozdziale zbudujemy program. Zaczniemy od pierwszego mglistego pomysłu, przejdziemy etapy analizy, projektowania, implementacji, testowania, ponownego projektowania, na ponownej implementacji kończąc. Chcemy pokazać proces myślowy, który ma miejsce podczas tworzenia oprogramowania. W międzyczasie omówimy organizację programu, typy definiowane przez użytkownika oraz techniki przetwarzania danych wejściowych.

6.1. Problem

6.2. Przemyślenie problemu

6.2.1. Etapy rozwoju oprogramowania

6.2.2. Strategia

6.3. Wracając do kalkulatora

6.3.1. Pierwsza próba

6.3.2. Tokeny

6.3.3. Implementowanie tokenów

6.3.4. Używanie tokenów

6.3.5. Powrót do tablicy

6.4. Gramatyki

6.4.1. Dygresja — gramatyka języka angielskiego

6.4.2. Pisanie gramatyki

6.5. Zamiana gramatyki w kod

6.5.1. Implementowanie zasad gramatyki

6.5.2. Wyrażenia

6.5.3. Składniki

6.5.4. Podstawowe elementy wyrażeń

6.6. Wypróbowywanie pierwszej wersji

6.7. Wypróbowywanie drugiej wersji

6.8. Strumienie tokenów

6.8.1. Implementacja typu `Token_stream`

6.8.2. Wczytywanie tokenów

6.8.3. Wczytywanie liczb

6.9. Struktura programu



6.1. Problem

Pisanie programu zaczyna się od postawienia problemu. To znaczy, jest problem, do rozwiązania którego chcemy napisać program. Aby ten program był dobry, należy dokładnie zrozumieć problem. Jeśli program będzie rozwiązywał nie ten problem, co trzeba, to nie będzie przydatny bez względu na to, jak może być elegancki. Czasami zdarzają się szczęśliwe przypadki, że program spełnia jakieś przypadkowe pożyteczne zadanie, mimo że został napisany w całkiem innym celu. Lepiej jednak nie liczyć na takie szczęście. My chcemy napisać taki program, który będzie w prosty i jasny sposób rozwiązywał dokładnie ten problem, dla którego został napisany.

Jaki program byłoby najlepiej napisać na tym etapie nauki? Taki, który:

- Ilustruje techniki projektowania i pisania programów.
- Umożliwia zapoznanie się z charakterem decyzji, które programista musi podejmować, oraz implikacjami, które te decyzje pociągają.
- Nie wymaga zastosowania zbyt wielu nowych konstrukcji programistycznych.
- Jest wystarczająco skomplikowany, aby zmusić nas do przemyślenia jego projektu.
- Można napisać na kilka sposobów.
- Rozwiązuje łatwy do zrozumienia problem.
- Rozwiązuje problem wart uwagi.
- Jest na tyle mały, że można go w całości przedstawić i zrozumieć.

Wybór padł na program „zmuszający komputer do wykonywania typowych działań arytmetycznych na wyrażeniach, które mu podamy” — tzn. chcemy napisać prosty kalkulator. Programy tego typu są użyteczne. Można je znaleźć w każdym komputerze biurkowym, a nawet można znaleźć takie komputery, które pozwalają uruchamiać tylko tego typu programy — kalkulatory kieszonkowe.

Jeśli np. wpiszemy

$$2+3.1*4$$

będziemy oczekiwać, że program zwróci wynik

$$14.4$$

Niestety kalkulator taki nie będzie miał żadnych funkcji, których już nie spełnia nasz komputer, ale to by było zbyt wysokie wymaganie wobec pierwszego programu.

6.2. Przemyślenie problemu

Od czego więc zacząć? Pomyśl o postawionym problemie i tym, jak go rozwiązać. Po pierwsze zdecyduj się, co program ma robić i w jaki sposób będziesz się z nim komunikować. Później będziesz mógł zastanowić się nad tym, jak napisać odpowiedni kod. Opisz wstępną wersję swojego rozwiązania i zastanów się, co należy poprawić. Możesz spróbować przedyskutować problem i jego rozwiązanie z kolegą lub koleżanką. Próba przedstawienia komuś własnych myśli jest doskonałym sposobem na znalezienie słabych punktów własnego rozumowania,

nawet lepszym niż napisanie ich. Papier (lub komputer) nie będzie z Tobą dyskutować i krytycznie oceniać Twoich poglądów. Idealny etap projektowania to taki, który przeprowadza się w towarzystwie.

Niestety nie ma takiej ogólnej strategii rozwiązywania problemów, która odpowiadałaby wszystkim ludziom i pozwalała rozwiązać każdy problem. Istnieje mnóstwo książek, których autorzy twierdzą, że pomogą Ci efektywniej rozwiązywać problemy, oraz cała masa publikacji na temat projektowania programów. Nasza droga nie wiedzie poprzez nie. W zamian opiszemy garść ogólnych wskazówek, które mogą być pomocne w rozwiązywaniu mniejszych problemów. Następnie szybko przejdziemy do wypróbowywania tych sugestii na naszym małym kalkulatorze.

Czytając treść naszych rozważań na temat kalkulatora, oceniaj to, co widzisz, bardzo sceptycznym okiem. Aby zachować realizm, przedstawimy ewolucję naszego programu poprzez szereg wersji. Opiszemy argumenty, które doprowadziły nas do powstania każdej z nich. Oczywiście znaczna część tej argumentacji musi być niekompletna lub błędna, inaczej szybko byśmy skończyli ten rozdział. Naszym celem jest pokazanie przykładowych problemów i procesów myślowych charakterystycznych dla procesu projektowania i implementowania programu. Wersja, z której jesteśmy ostatecznie zadowoleni, zostanie przedstawiona dopiero na końcu następnego rozdziału.

Pamiętaj, że w tym i następnym rozdziale proces dochodzenia do finalnej wersji programu — droga wiodąca przez niepełne rozwiązania, pomysły i błędy — jest co najmniej tak samo ważny, jak wersja ostateczna i ważniejszy niż narzędzia techniczne języka programowania, których będziemy używać (wrócimy do nich później).

6.2.1. Etapy rozwoju oprogramowania

Poniżej przedstawimy nieco terminologii związanej z tworzeniem oprogramowania. Pracując nad rozwiązaniem problemu, wielokrotnie powtarza się następujące fazy:

- *Analiza* — przemyśl, co masz zrobić, i opisz, jak to aktualnie rozumiesz. Taki opis nazywa się **zestawem wymagań** lub **specyfikacją**. Nie będziemy szczegółowo opisywać technik opracowywania i opisywania takich wymogów. Temat tej książki tego nie obejmuje, ale należy pamiętać, że w miarę zwiększania się rozmiaru problemu techniki te nabierają wagi.
- *Projektowanie* — utworzenie ogólnej struktury systemu i podjęcie decyzji, na jakie części podzielić implementację oraz jak powinny się one ze sobą komunikować. W ramach projektowania zastanów się jakie narzędzia — np. biblioteki — możesz wykorzystać do opracowania struktury programu.
- *Implementacja* — napisz kod, usuń błędy oraz sprawdź za pomocą testów, czy robi to, co powinien.



6.2.2. Strategia

Oto garść wskazówek, które — jeśli zostaną zastosowane z rozwagą i wyobraźnią — będą pomocne w wielu projektach programistycznych:

- Jaki problem jest do rozwiązania? Przynajmniej na początku należy starać się dokładnie opisać, co chce się zrobić. Najczęściej oznacza to sporządzenie opisu problemu lub, jeśli ktoś



inny dał nam zadanie do wykonania, próbę odszyfrowania, co dokładnie to oznacza. W tym momencie należy przyjąć punkt widzenia użytkownika (nie programisty czy implementatora). To znaczy, zamiast zastanawiać się, jak rozwiązać problem, pomyśl, co program ma w ogóle robić. Zadawaj pytania typu: „Co ten program może dla mnie zrobić?” albo „W jaki sposób chciałbym komunikować się z tym programem?”. Pamiętaj, że większość z nas może korzystać z własnego bogatego doświadczenia jako użytkownika komputerów.

- Czy problem został jasno nakreślony? W realnym świecie nigdy nie jest. Nawet w takim ćwiczeniu trudno jest opisać go wystarczająco precyzyjnie. Dlatego staramy się wszystko wyjaśnić. Szkoda by było, gdybyśmy rozwiązali nie ten problem, co trzeba. Inna pułapka to wygórowane wymagania. Obmyślając, co byśmy chcieli, łatwo możemy ulec chciwości lub nadmiernym ambicjom. Zawsze lepiej jest obniżyć wymagania, aby ułatwić napisanie specyfikacji programu, jego zrozumienie, użytkowanie oraz (taką trzeba mieć nadzieję) implementację. Gdy zadziała, zawsze można napisać wzbogaconą wersję 2.0.
- Czy problem wydaje się możliwy do rozwiązania w przewidzianym czasie oraz przy określonym zasobie umiejętności i dostępnych narzędziach? Nie ma sensu rozpoczynać projektu, którego nie mamy szans ukończyć. Jeśli jest za mało czasu na implementację (włącznie z testowaniem) wszystkich wymaganych funkcji programu, zazwyczaj lepiej jest go w ogóle nie zaczynać. Lepiej zamiast tego zdobyć więcej zasobów (zwłaszcza czasu) lub (idealnie) zmienić wymagania, aby ułatwić zadanie.
- Spróbuj podzielić program na dające się ogarnąć myślami części. Nawet najmniejszy program rozwiązujący realny problem można podzielić na części.
- Znasz jakieś narzędzia, biblioteki itp., które mogą być pomocne? Odpowiedź prawie zawsze brzmi: tak. Od samego początku nauki programowania masz do dyspozycji zawartość standardowej biblioteki C++. Później poznasz znaczną jej część i dowiesz się, jak poszukiwać jeszcze więcej. Będziesz korzystać z bibliotek graficznych, macierzy itp. Mając odrobinę doświadczenia, będziesz w stanie znaleźć tysiące bibliotek w internecie. Pamiętaj — nie ma sensu wyważać otwartych drzwi, jeśli tworzy się oprogramowanie do realnego użytku. Co innego w czasie nauki programowania. Wówczas takie działania w celu sprawdzenia, jak to zrobili inni, mają głęboki sens. Cały czas, który oszczędzisz dzięki wykorzystaniu istniejących bibliotek, możesz poświęcić na pracę nad innymi częściami problemu albo na odpoczynek. Skąd wiadomo, czy dana biblioteka spełnia nasze wymagania i prezentuje odpowiednią jakość? To trudne pytanie. Można spytać znajomych, popytać na grupach dyskusyjnych i wypróbować kilka krótkich przykładów, zanim się zdecydujemy.
- Wyodrębnij takie części rozwiązania, które można opisać oddzielnie od reszty (i potencjalnie wykorzystać w kilku miejscach programu, a nawet w innych programach). Umiejętność znajdowania takich części przychodzi z doświadczeniem, dlatego w książce tej przedstawiamy wiele takich przykładów. Używaliśmy już wektorów, łańcuchów i strumieni (cin i cout). W tym rozdziale po raz pierwszy przedstawimy przykłady projektów, implementacji i wykorzystania części programów dostępnych jako typy

zdefiniowane przez użytkownika (Token i Token_stream). Jeszcze więcej takich przykładów znajduje się w rozdziałach 8. oraz 13. – 15., w których zostaną opisane techniki ich tworzenia. Na razie zastanów się nad taką analogią — gdybyśmy projektowali samochód, zaczęlibyśmy od opisu jego podzespołów, takich jak koła, silnik, fotele, klamki do drzwi itp. Każdą z nich wyprodukowalibyśmy osobno, aby następnie użyć jej do złożenia całego pojazdu. W nowoczesnym samochodzie wykorzystuje się dziesiątki tysięcy takich części. Realne programy nie różnią się w niczym od samochodów pod tym względem (poza tym, że częściami są fragmenty kodu). Nie przyszło by nam do głowy budować samochodu z surowych materiałów, jak żelazo, plastik i drewno. Analogicznie nie tworzymy poważnych programów, bezpośrednio używając wyrażień, instrukcji typów wbudowanych w język. Projektowanie i implementowanie takich części jest najważniejszym tematem tej książki, a nawet w ogólnie programowania. Zajrzyj też do rozdziałów 9. (typy definiowane przez użytkownika), 14. (hierarchie klas) oraz 20. (typy ogólne).

- Zbuduj niewielką wersję programu z ograniczoną funkcjonalnością, która rozwiązuje najważniejszą część problemu. Rzadko się zdarza, abyśmy od samego początku dokładnie znali charakter problemu. Często nam się tak wydaje (chyba wiadomo, co to jest program kalkulator?), ale w rzeczywistości jest inaczej. Tylko dzięki wytężonemu myśleniu o problemie (analiza) i eksperymentowaniu (projekt i implementacja) można na tyle dokładnie zrozumieć problem, aby napisać dobry program. Dlatego tworzymy ograniczoną wersję, aby:
 - Odkryć własne braki w rozumowaniu, pojmowaniu problemu oraz narzędziach.
 - Sprawdzić, czy nie trzeba zmienić niektórych szczegółów w specyfikacji problemu, aby zadanie było wykonalne. Rzadko się zdarza, aby w czasie analizy i we wstępnej fazie projektowania udało się przewidzieć wszystkie możliwe trudności. Należy skorzystać z informacji, które zyskujemy podczas pisania i testowania kodu.

Taką wstępną wersję o ograniczonej funkcjonalności czasami nazywa się **prototypem**. Jeśli (co jest prawdopodobne) ta pierwsza wersja nie działa lub jest tak brzydka, że nie mamy ochoty się nią dłużej zajmować, wyrzucamy ją i tworzymy nowy prototyp, tym razem wzbogaceni o nowe doświadczenia. Powtarzamy ten proces aż do uzyskania zadowalającego wyniku. Nie kontynuuj pracy w bałaganie, który z czasem tylko się pogorszy.

- Zbuduj pełną wersję, najlepiej wykorzystując w tym celu części z wersji wstępnej. Chodzi o to, aby program tworzyć z działających części, a nie pisać cały kod na raz. Można oczywiście mieć nadzieję, że jakimś cudem nieprzetestowane fragmenty kodu będą działać i na dodatek robić to, co zaplanowano.

6.3. Wracając do kalkulatora

W jaki sposób będziemy komunikować się z kalkulatorem? To łatwe — wiemy już, jak posługiwać się strumieniami `cin` i `cout`, a graficzne interfejsy użytkownika (GUI) zostaną opisane dopiero w rozdziale 16. W związku z tym wybór padł na okno konsoli. Program będzie pobierał z klawiatury wyrażenia, obliczał ich wartość i drukował wynik na ekranie. Na przykład:

Wyrażenie: 2+2
 Wynik: 4
 Wyrażenie: 2+2*3
 Wynik: 8
 Wyrażenie: 2+3-25/5
 Wynik: 0

Wyrażenia, tzn. 2+2 i 2+2*3, powinien wpisywać użytkownik. Reszta należy do programu. Wyświetlenie słowa *Wyrażenie*: będzie zachętą dla użytkownika do wpisania wyrażenia. Moglibyśmy napisać *Proszę wpisać wyrażenie i znak nowego wiersza*:, ale to wydawało nam się zbyt rozwlekłe. Z drugiej strony taki przyjemny znaczek > byłby chyba za bardzo tajemniczy. Takie szkicowanie przykładów użycia we wczesnej fazie pracy jest bardzo ważne. Dzięki temu można się dowiedzieć, jaki jest minimalny zestaw funkcji programu. W projektowaniu i analizie przykłady takie nazywają się **przypadkami użycia**.

Większość ludzi, którzy po raz pierwszy stykają się z problemem kalkulatora, wpada na następujący pomysł, jeśli chodzi o główną logikę programu:

```
wczytaj_wiersz
oblicz      // wykonuje pracę
wydrukuj_wynik
```

Takie zapiski to oczywiście nie jest prawdziwy kod, tylko tzw. *pseudokod*. Stosuje się go we wczesnych fazach projektowania, gdy nie ma jeszcze pewności co do tego, jaką zastosować notację. Np., czy obliczenia ma być wywołaniem funkcji? Jeśli tak, to jakie będzie przyjmować argumenty? Jest po prostu za wcześnie na zadawanie takich pytań.

6.3.1. Pierwsza próba

Na tym etapie nie jesteśmy jeszcze gotowi napisać programu kalkulatora. Nie przemyśleliśmy jeszcze wszystkiego, ale myślenie to ciężka praca i — jak większość programistów — nie możemy się doczekać, żeby już coś napisać. Spróbujemy więc swoich sił i napiszemy prosty kalkulator, aby zobaczyć, do czego nas to doprowadzi. Nasz pierwszy pomysł wygląda tak:

```
#include "std_lib_facilities.h"
int main()
{
    cout << "Wpisz wyrażenie (obsługujemy operatory + i -): ";
    int lval = 0;
    int rval;
    char op;
    int res;
    cin >> lval >> op >> rval; // Wczytuje coś w rodzaju 1 + 3.
    if (op == '+')
        res = lval + rval; // dodawanie
    else if (op == '-')
        res = lval - rval; // odejmowanie
    cout << "Wynik: " << res << '\n';
    keep_window_open();
    return 0;
}
```


Wczytujemy parę wartości oddzielonych operatorem, np. 2+2, obliczamy wynik (tu 4) i drukujemy go na ekranie. Zmienną przechowującą wartość z lewej strony operatora nazwaliśmy `lval`, a z prawej strony `rval`.

To nawet działa! Co z tego, że program nie jest ukończony? To wspaniałe uczucie zrobić coś, co działa! Może to programowanie i informatyka są łatwiejsze, niż głoszą plotki? Możliwe, ale nie dajmy się ponieść emocjom z powodu tego pierwszego sukcesu. Oto lista czynności:

1. Oczyszczyć kod.
2. Dodać obsługę dzielenia i mnożenia (np. **2*3**).
3. Dodać obsługę wyrażeń zawierających więcej niż jeden operand (np. **1+2+3**).

W szczególności pamiętamy, że zawsze należy sprawdzać, czy użytkownik podał sensowne dane (zapomnieliśmy z pośpiechu wcześniej), oraz że porównywanie jednej wartości z wieloma stałymi lepiej wykonać za pomocą instrukcji `switch` niż `if`.

Łączenie działań w łańcuchy, np. 1+2+3+4, obsłużymy, sumując wartości w czasie wczytywania. Tzn. wczytujemy 1, widzimy +2, więc dodajemy 2 do 1 (uzyskując w ten sposób wynik 3). Dalej widzimy +3, a więc dodajemy 3 do poprzedniego wyniku itd. Po kilku falstartach i poprawieniu kilku błędów składni uzyskaliśmy następujący rezultat:

```
#include "std_lib_facilities.h"
int main()
{
    cout << "Wpisz wyrażenie (obsługujemy operatory +, -, * oraz /): ";
    int lval = 0;
    int rval;
    char op;
    cin>>lval; // Wczytywanie pierwszego argumentu wyrażenia z lewej.
    if (!cin) error("Na początku nie ma argumentu wyrażenia.");
    while (cin>>op) { // Wczytywanie operatora i prawego argumentu wyrażenia na zmianę.
        cin>>rval;
        if (!cin) error("Nie ma drugiego argumentu wyrażenia.");
        switch(op) {
            case '+':
                lval += rval; // Dodawanie: lval = lval + rval
                break;
            case '-':
                lval -= rval; // Odejmowanie: lval = lval - rval
                break;
            case '*':
                lval *= rval; // Mnożenie: lval = lval * rval
                break;
            case '/':
                lval /= rval; // Dzielenie: lval = lval / rval
                break;
            default: // Koniec operatorów — drukowanie wyniku.
                cout << "Wynik: " << lval << '\n';
                keep_window_open();
                return 0;
        }
    }
}
```

```

    }
    error("Nieprawidłowe wyrażenie.");
}

```

Wygląda niezłe, ale gdy wpisujemy wyrażenie $1+2*3$, to ujrzymy wynik 9 zamiast 7, którego spodziewalibyśmy się na podstawie wiedzy zdobytej w szkole podstawowej. Analogicznie wynikiem wyrażenia $1-2*3$ będzie -3 zamiast spodziewanego -5. Kalkulator wykonuje działania w złej kolejności — wyrażenie $1+2*3$ jest liczone jako $(1+2)*3$ zamiast $1+(2*3)$. Analogicznie $1-2*3$ jest liczone jako $(1-2)*3$ zamiast $1-(2*3)$. Lipa! Moglibyśmy uznać, że zasada, iż „mnożenie wiąże mocniej niż dodawanie” jest głupią i przestarzałą konwencją, ale nie możemy zignorować wielowiekowej tradycji, aby ułatwić sobie programowanie.

6.3.2. Tokeny

Musimy zatem znaleźć sposób na wczytywanie części wiersza „na zapas”, aby sprawdzić, czy nie ma tam gdzieś operatora $*$ (albo $/$). Jeśli jest, musimy zmienić kolejność wykonywania działań. Niestety próbując wczytać nieco danych z wyprzedzeniem, napotkamy kilka trudności:

1. Nie wymagamy, aby wyrażenie znajdowało się w jednym wierszu. Na przykład poniższe też jest poprawne:

```

1
+
2

```

2. Jak znaleźć znaki $*$ i $/$ wśród cyfr, plusów, minusów i nawiasów w kilku wierszach danych wejściowych?
3. Jak zapamiętać, gdzie znajdował się znaleziony znak $*$?
4. Jak wykonać obliczenia, które nie są ściśle typu „od lewej do prawej”?

Postanowiliśmy być wielkimi optymistami i zająć się tylko punktami 1 – 3. Ostatnim zajmniemy się trochę później.

Poszukamy pomocy. Przecież ktoś na pewno zna typowy sposób wczytywania danych typu liczby i operatory i zapisywania ich w taki sposób, aby można je było łatwo wykorzystać w obliczeniach. Ta konwencjonalna i przydatna technika nazywa się rozbiorem na składniki, czyli tokeny (ang. *tokenize*) — wczytuje się dane i dzieli je na **tokeny**. Na przykład wyrażenie


```
45+11.5/7
```

zostałoby rozłożone na tokeny

```

45
+
11.5
/
7

```

 **Token** to sekwencja znaków, która reprezentuje pewną całość, np. liczbę lub operator. Kompilator C++ dzieli na tokeny kod źródłowy. W istocie różne formy rozkładu na czynniki są podstawą analizy większości rodzajów tekstów. W wyrażeniach matematycznych w języku C++ potrzebujemy trzech rodzajów tokenów:

- literały zmiennoprzecinkowe — zgodnie z definicją w C++, np. 3.14, 0.274e2 i 42;
- operatory — +, -, *, / oraz %;
- nawiasy — (i).

Wydaje się, że trudności mogą nastęrczać literały zmiennoprzecinkowe. Wczytanie liczby 12 wydaje się znacznie łatwiejsze niż 12.3e-3. Ale kalkulatory zazwyczaj wykonują działania na liczbach zmiennoprzecinkowych. Analogicznie podejrzewamy, że aby nasz kalkulator był przydatny, musi obsługiwać nawiasy.

W jaki sposób reprezentuje się takie tokeny w programie? Można spróbować zapamiętywać, gdzie każdy token się zaczyna, a gdzie kończy, ale to może być uciążliwe (zwłaszcza jeśli pozwolimy na wpisywanie wyrażeń obejmujących więcej niż jeden wiersz). Dodatkowo, jeśli będziemy zapisywać wartości jako sekwencje znaków, będziemy musieli później znaleźć sposób na odczytanie tych wartości. To znaczy, jeśli liczbę 42 zapiszemy jako znaki 4 i 2, później będziemy musieli odgadnąć, że te dwa znaki reprezentują liczbę 42 (tzn. $4 \cdot 10 + 2$). Oczywistym i konwencjonalnym rozwiązaniem tego problemu jest przedstawienie każdego tokenu jako pary (**rodzaj, wartość**). Pierwszy element informuje o rodzaju tokenu — liczba, operator, nawias. Drugi natomiast np. w przypadku liczb określa dokładną wartość.

Jak więc wykorzystać pomysł par (**rodzaj, wartość**) w kodzie? Zdefiniujemy typ Token do reprezentowania tokenów. Po co? Przypomnij sobie, po co używamy typów: przechowują potrzebne nam dane i pozwalają wykonywać na nich różne operacje. Na przykład typ `int` pozwala przechowywać liczby całkowite i umożliwia dodawanie, odejmowanie, mnożenie oraz dzielenie tych liczb. Natomiast typ `string` przechowuje łańcuchy znaków i pozwala je np. łączyć. W języku C++ i jego bibliotece standardowej dostępnych jest wiele typów, np. `char`, `int`, `double`, `string`, `vector` i `ostream`. Nie ma jednak typu Token. W istocie można wymienić mnóstwo typów — tysiące, a nawet dziesiątki tysięcy — które chcielibyśmy mieć do dyspozycji, a których nie ma w języku ani jego bibliotece standardowej. Do naszych ulubionych typów, które nie są standardowo dostępne, należą `Matrix` (zobacz rozdział 24.), `Date` (zobacz rozdział 9.) oraz reprezentujące go liczby całkowite nieskończonej precyzji (poszukaj w internecie informacji na temat typu `Bignum`). Jeśli przemyślisz to, dojdiesz do wniosku, że język nie może standardowo obsługiwać dziesiątek tysięcy typów — kto by je zdefiniował i zaimplementował, kto by je potem znalazł, nie mówiąc już o tym, jak gruby musiałby być podręcznik do nauki takiego języka. Język C++ wzorem innych nowoczesnych języków programowania rozwiązuje ten problem, pozwalając użytkownikowi definiować własne (niestandardowe) typy (ang. *user-defined type* — typ zdefiniowany przez użytkownika).



6.3.3. Implementowanie tokenów

Jak powinien wyglądać token? To znaczy, jakie właściwości powinien mieć nasz typ Token? Musi nadawać się do reprezentowania operatorów (np. + i -) i wartości liczbowych (np. 42 i 3.14). Oczywistym rozwiązaniem jest zaimplementowanie czegoś takiego, co może zawierać informację na temat rodzaju tokenu i w razie potrzeby jego wartość:

Token:		Token:	
kind:	plus	kind:	liczba
value:		value:	3.14

Pomysł ten można zaimplementować w języku C++ na wiele sposobów. Przedstawiamy najprostszy, który wydaje nam się użyteczny:

```
class Token { // Bardzo prosty typ zdefiniowany przez użytkownika.
public:
    char kind;
    double value;
};
```

Token to typ (tak samo jak `int` czy `char`), a więc można go używać do definiowania zmiennych i przechowywania wartości. Składa się z dwóch części (nazywanych składowymi) — `kind` (rodzaj) oraz `value` (wartość). Słowo kluczowe `class` oznacza „typ zdefiniowany przez użytkownika”. Wskazuje definicję typu z zerem lub większą liczbą składowych. Pierwsza składowa o nazwie `kind` jest znakiem `char`, a więc można jej użyć do przechowywania znaków `'+'` i `'*'`, które będą reprezentowały operatory. Przy użyciu tego typu można tworzyć następujące instrukcje:

```
Token t; // Zmienna t jest typu Token.
t.kind = '+'; // Zmienna t reprezentuje znak +.
Token t2; // Zmienna t2 jest innym obiektem typu Token.
t2.kind = '8'; // Cyfra 8 oznacza rodzaj (kind) tokenu będący liczbą.
t2.value = 3.14;
```

Aby uzyskać dostęp do składowej, posługujemy się odpowiednią notacją — **nazwa_obiektu.nazwa_składowej**. Tekst `t.kind` można przeczytać jako „rodzaj obiektu `t`”, a `t2.value` jako „wartość obiektu `t2`”. Obiekty typu `Token` można kopiować tak samo jak typu `int`:

```
Token tt = t; // Inicjacja kopii
if (tt.kind != t.kind) error("To niemożliwe!");
t = t2; // przypisanie
cout << t.value; // wydrukuje 3.14
```

Mając typ `Token`, wyrażenie $(1.5+4)*11$ można przedstawić za pomocą siedmiu tokenów:

'('	'8'	'+'	'8'	*)	'8'
	1.5		4		11

Należy zauważyć, że proste tokeny, jak `+`, nie mają wartości, a więc do ich reprezentowania niepotrzebna jest składowa `value`. Potrzebowaliśmy znaku, który oznaczałby liczbę. Wybór padł na `'8'`, ponieważ nie jest to operator ani znak interpunkcyjny. Wykorzystanie `'8'` w taki sposób jest dosyć tajemnicze, ale na razie może być.

`Token` jest przykładem typu zdefiniowanego przez użytkownika w języku C++. Typy takie poza danymi składowymi mogą też zawierać funkcje (operacje) składowe. Istnieje wiele powodów, dla których się je definiuje. My zdefiniujemy tylko dwie, aby ułatwić sobie inicjowanie obiektów typu `Token`:

```
class Token {
public:
    char kind; // Rodzaj tokenu
    double value; // Dla liczb: wartość.
```

```

Token(char ch) // Tworzy Token ze znaku.
    :kind(ch), value(0) { }
Token(char ch, double val) // Tworzy Token ze znaku i wartości typu double.
    :kind(ch), value(val) { }
};

```

Nie są to zwykle funkcje składowe, tylko tzw. **konstruktory**. Mają taką samą nazwę jak ich typ i służą do inicjalizowania (tworzenia) obiektów typu Token. Na przykład:

```

Token t1('+'); // Inicjacja zmiennej t1 — t1.kind = '+'.
Token t2('8',11.5); // Inicjacja zmiennej t2 — t2.kind = '8' i t2.value = 11.5.

```

Tekst `:kind(ch), value(0)` w pierwszym konstruktorze oznacza: „Zainicjuj składową `kind` wartością `ch`, a `value` ustaw na 0”. W drugim konstruktorze znajduje się tekst `:kind(ch), value(val)`, który oznacza: „Zainicjuj składową `kind` wartością `ch`, a `value` ustaw na `val`”. W obu przypadkach nie trzeba robić nic więcej, aby utworzyć obiekt typu Token, dlatego treść funkcji jest pusta — `{ }`. Specjalna składnia inicjująca (ang. *member initializer list* — **lista wartości inicjujących składowe**), która zaczyna się dwukropkiem, jest używana tylko w konstruktorach.

Zauważ, że konstruktor nie zwraca wartości. Dlatego nie trzeba (a nawet nie można) definiować typu zwrotnego konstruktora. Więcej na temat konstruktorów napiszemy w rozdziałach 9.4.2 i 9.7.

6.3.4. Używanie tokenów

Może spróbujemy dokończyć nasz kalkulator! Chociaż warto by było najpierw opracować jakiś plan działania. Jak będziemy wykorzystywać obiekty typu Token w programie? Możemy wczytać dane wejściowe do wektora takich obiektów:

```

Token get_token(); // Wczytuje token ze strumienia cin.
vector<Token> tok; // Tutaj będziemy zapisywać tokeny.
int main()
{
    while (cin) {
        Token t = get_token();
        tok.push_back(t);
    }
    // ...
}

```

Teraz możemy wczytać wyrażenie w całości i dopiero po tym obliczyć jego wartość. W przypadku $11*12$ otrzymamy coś takiego:

'8'	'*'	'8'
11		12

W strukturze tej możemy znaleźć operator i jego operandy. Gdy to zrobimy, z łatwością wykonamy działanie mnożenia, ponieważ liczby 11 i 12 zostały zapisane jako wartości liczbowe, a nie łańcuchy.

Teraz przeanalizujemy bardziej skomplikowane wyrażenie. Dla wyrażenia $1+2*3$ tok będzie zawierać pięć obiektów typu Token:

'8'	'+'	'8'	'*'	'8'
1		2		3

W tym przypadku operator mnożenia można znaleźć za pomocą prostej pętli:

```
for (int i = 0; i<tok.size(); ++i) {
    if (tok[i].kind=='*') { // Znaleźliśmy operator mnożenia!
        double d = tok[i-1].value*tok[i+1].value;
        // Co teraz?
    }
}
```

No dobrze, ale co teraz? Co zrobimy z iloczynem d ? Jak określić kolejność wykonywania działań w wyrażeniu? Operator $+$ znajduje się przed $*$, a więc nie możemy po prostu wykonać wszystkich działań od lewej do prawej. Możemy spróbować od prawej do lewej! To by się sprawdziło w przypadku wyrażenia $1+2*3$, ale już nie dla wyrażenia $1*2+3$. Nie wspominając już o takim czymś, jak $1+2*3+4$. To wyrażenie trzeba obliczyć „od środka” — $1+(2*3)+4$. Jak poradzimy sobie z nawiasami? Wydaje się, że utknęliśmy. Musimy się wycofać, przestać na chwilę programować i pomyśleć nad wczytywaniem oraz tym, jak rozumiemy łańcuch wejściowy i jak obliczamy jego wartość jako wyrażenia.



Pierwsza entuzjastyczna próba rozwiązania problemu (napisania kalkulatora) zakończyła się kląpą. Pierwsze podejścia często tak się kończą i sytuacje takie są nam potrzebne, ponieważ dzięki nim możemy lepiej zrozumieć naturę problemu. W tym przypadku nawet poznaliśmy przydatne pojęcie tokenu, które samo jest przykładem pojęcia pary (**nazwa, wartość**), z którym będziemy jeszcze wielokrotnie się spotykać. Musisz jednak pamiętać, że takie bezmyślne i nieplanowane pisanie kodu nie powinno zajmować zbyt dużo czasu. Przed dokonaniem analizy (próbą zrozumienia problemu) i opracowaniem projektu (opracowaniem ogólnej struktury rozwiązania) powinno się napisać bardzo mało kodu.

WYPRÓBUJ



Z drugiej strony, czemu nie moglibyśmy znaleźć prostego rozwiązania tego problemu? Nie wydaje się aż taki trudny. Podjęcie próby rozwiązania problemu, nawet zakończone fiaskiem, może doprowadzić nas do lepszego zrozumienia problemu i opracowania rozwiązania. Pomyśl, co możesz zrobić od razu. Jako przykład niech posłuży wyrażenie $12.5+2$. Możemy rozbić je na tokeny, dojść do wniosku, że to bardzo proste wyrażenie i obliczyć wynik. Może byłoby trochę bałaganu, ale rozwiązanie jest proste, więc może warto podążyć w tym kierunku i znaleźć coś, co wystarczy! Pomyśl, co byś zrobił, gdybyś znalazł operatory $+$ i $*$ w wierszu $2+3*4$. To także można obliczyć „na piechotę”. Jak byśmy sobie poradzili ze skomplikowanym wyrażeniem typu $1+2*3/4*5+(6-7*(8))$? Jak byśmy radzili sobie z błędami, np. $2+*3$ albo $2&3$? Pomyśl nad tym przez chwilę. Możesz zrobić sobie notatki na kartce, nakreślić możliwe rozwiązania oraz wypisz interesujące lub ważne wyrażenia.

6.3.5. Powrót do tablicy

Jeszcze raz przeanalizujemy problem, tym razem starając się nie wrywać z nieprzemysłanymi pomysłami. Jedyne, co odkryliśmy, to fakt, że obliczenie przez program tylko jednego

wyrażenia sprawia nam trudności. Chcielibyśmy mieć możliwość obliczenia wielu wyrażeń w jednym uruchomieniu programu. W związku z tym wzbogacamy nasz pseudokod w następujący sposób:

```
while (nie_skończone) {
    wczytaj_wiersz
    oblicz          // wykonaj pracę
    wydrukuj_wynik
}
```

To z pewnością komplikuje sprawę, ale musimy wziąć pod uwagę fakt, że kalkulatorów zazwyczaj używa się do wykonywania kilku obliczeń po kolei. Czy mamy kazać użytkownikowi uruchamiać nasz program ponownie, aby wykonać każde obliczenie? Moglibyśmy, ale w wielu nowoczesnych systemach operacyjnych uruchamianie programów trwa za długo, a więc lepiej tego nie robić.

Kiedy patrzymy na nasz pseudokod, nasze początkowe próby rozwiązania problemu i przykłady użycia, nasuwa się nam kilka pytań (i kilka nieśmiałych odpowiedzi):

1. Jeśli użytkownik wpisze $45+5/7$, jak znajdziemy poszczególne elementy — 45, 5, / i 7? Odpowiedź: podzielimy na tokeny!
2. W jaki sposób oznaczymy koniec wyrażenia? Oczywiście znakiem nowego wiersza (zawsze podejrzliwie traktuj zwroty typu „oczywiście” — „oczywiście” to nie żaden powód!
3. Jak zaprezentujemy wyrażenie $45+5/7$ jako dane, aby można było obliczyć wynik? Przed wykonaniem dodawania musimy w jakiś sposób zamienić znaki 4 i 5 w liczbę całkowitą 45 (tj. $4*10+5$). Zatem podział na tokeny jest częścią rozwiązania.
4. Jak sprawić, aby wyrażenie $45+5/7$ było obliczane jako $45+(5/7)$, a nie $(45+5)/7$?
5. Ile wynosi $5/7$? Około .71, a więc to nie jest liczba całkowita. Z doświadczenia wiemy, że użytkownicy kalkulatorów oczekują wyników zmiennoprzecinkowych. Czy powinniśmy pozwolić na wpisywanie liczb zmiennoprzecinkowych? Oczywiście!
6. Czy możemy pozwolić na używanie zmiennych? Moglibyśmy na przykład napisać:

```
v=7
m=9
v*m
```


Dobry pomysł, ale zostawimy to na później. Na razie zajmiemy się podstawową funkcjonalnością.

Najważniejsza decyzja to prawdopodobnie odpowiedź na pytanie w punkcie 6. W rozdziale 7.8 zobaczysz, że odpowiedź ta pociągnie za sobą prawie podwojenie rozmiaru wstępnej wersji projektu. To podwoiłoby czas potrzebny na uruchomienie wstępnej wersji programu. Podejrzewamy, że początkujący potrzebowałby nawet cztery razy więcej czasu i niewykluczone, że straciłby w końcu cierpliwość. We wczesnych fazach prac nad projektem należy zawsze unikać przesady z liczbą funkcji. Wstępna wersja zawsze powinna być prosta i zawierać tylko najważniejsze funkcje. Kiedy uda Ci się zmusić coś do działania, możesz postawić sobie bardziej ambitne wymagania. Budowa programu etapami jest znacznie łatwiejsza niż wszystkiego na raz. Odpowiedź „tak” na pytanie 6. miałyby jeszcze jeden zły wynik: mogłoby być trudno oprzeć



się pokusie dodania jeszcze innych funkcji. Może warto pomyśleć o funkcjach matematycznych albo o pętlach? Gdy zacznie się dodawać kolejne „fajne” funkcje, trudno przestać.

Z programistycznego punktu widzenia najbardziej kłopotliwe są punkty 1, 3 i 4. Ponadto są ze sobą powiązane, ponieważ gdy znajdziemy już 45 i +, co mamy z nimi zrobić? Tzn., jak zapisać je w programie? Oczywiście częściowym rozwiązaniem tego problemu jest podział na tokeny, ale tylko częściowym.



Co zrobiłby doświadczony programista? Gdy mamy do rozwiązania jakiś trudny techniczny problem, często można znaleźć jakieś standardowe rozwiązanie. Wiemy, że ludzie piszą kalkulatory, przynajmniej od kiedy istnieją komputery przyjmujące dane symboliczne z klawiatury, a więc od 50 lat. Musi być jakieś standardowe rozwiązanie! W takiej sytuacji doświadczony programista konsultuje się z kolegami i przeszukuje dostępną literaturę. Byłoby głupstwem myśleć, że w jeden dzień uda się wymyśleć coś lepszego, niż inni wymyślili przez 50 lat.

6.4. Gramatyki

Istnieje standardowa odpowiedź na pytanie, jak rozszyfrować znaczenie wyrażenia: najpierw wprowadzone znaki należy zebrać i podzielić na tokeny (to już sami odkryliśmy). Jeśli użytkownik wpisze:

$$45+11.5/7$$

program powinien utworzyć następującą listę tokenów:

```
45
+
11.5
/
7
```

Token to sekwencja znaków, którą uważamy za jakąś jednostkę, np. operator lub liczbę.

Po utworzeniu tokenów program musi upewniać się, że całe wyrażenie jest poprawnie rozumiane. Na przykład wiemy, że wyrażenie $45+11.5/7$ oznacza $45+(11.5/7)$, a nie $(45+11.5)/7$. Sęk w tym, jak nauczyć program tej przydatnej zasady (dzielenie „wiąże mocniej” niż dodawanie)? Standardowa odpowiedź jest taka, że piszemy **gramatykę** definiującą składnię naszych danych wejściowych, a następnie piszemy program, w którym implementujemy zasady tej gramatyki. Na przykład:

// Prosta gramatyka wyrażeń:

```
Expression:
  Term
  Expression "+" Term // dodawanie
  Expression "-" Term // odejmowanie
Term:
  Primary
  Term "*" Primary // mnożenie
  Term "/" Primary // dzielenie
  Term "%" Primary // reszta z dzielenia (modulo)
Primary:
  Number
```



```
"(" Expression ")" // grupowanie
Number:
    floating-point-literal
```

Jest to zestaw prostych zasad. Ostatnią należy czytać następująco: „Number (liczba) to literał zmiennoprzecinkowy”. Natomiast treść przedostatniej jest taka: „Primary (czynnik) jest liczbą lub znakiem '(', po którym jest wyrażenie i znak ')’”. Reguły dla Expression (wyrażenia) i Term (składnika) są podobne. Każda z nich jest zdefiniowana z uwzględnieniem jednej z reguł, które znajdują się dalej.

Jak pamiętamy z podrozdziału 6.3.2, nasze tokeny — zgodnie z definicją w języku C++ — to:

- literał zmiennoprzecinkowy (zgodny z definicją w języku C++, np. 3.14, 0.274e2 lub 42);
- +, -, *, / oraz % — operatory;
- (i) — nawiasy.

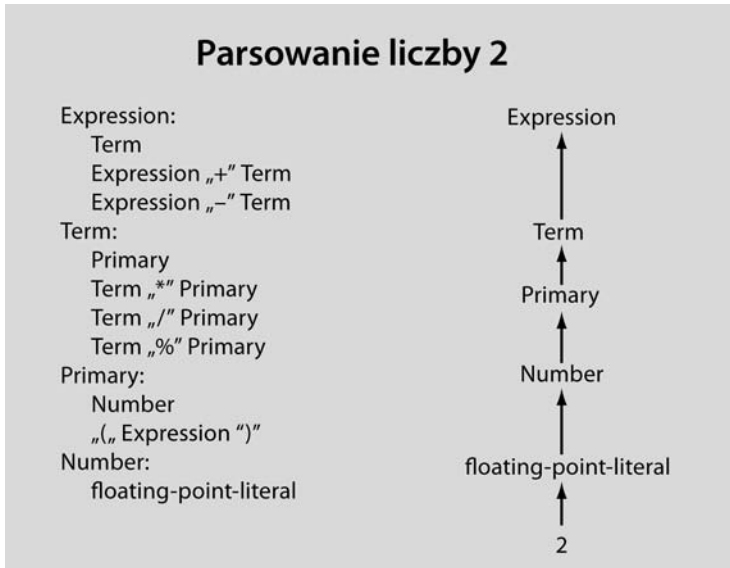
Używając gramatyki i tokenów, zrobiliśmy bardzo duży pojęciowy skok w stosunku do naszego początkowego pseudokodu. Tego rodzaju postępy chcielibyśmy robić zawsze, ale rzadko się to udaje bez pomocy. Do tego właśnie służą doświadczenie, literatura i mentorzy.

Na pierwszy rzut oka gramatyka ta wydaje się bezsensowna. Często tak jest z notacją techniczną. Pamiętaj jednak, że jest to ogólna i elegancka (co w końcu docenisz) notacja do opisu czegoś, co potrafisz robić przynajmniej od czasów szkoły podstawowej. Nie masz problemu z obliczeniem wyrażenia $1-2*3$ albo $1+2-3$ lub $3*2+4/2$. Potrafisz jednak wyjaśnić, jak to robisz? Umiesz to tak wyjaśnić, aby zrozumiał to nawet ktoś, kto nigdy nie miał styczności z konwencjonalną arytmetyką? Czy Twoje wyjaśnienia będą miały zastosowanie dla wszystkich kombinacji operatorów i argumentów? Aby wystarczająco szczegółowo i precyzyjnie objaśnić coś komputerowi, potrzebna jest odpowiednia notacja — a gramatyka należy do najlepszych konwencjonalnych narzędzi do jej tworzenia.

Jak czyta się gramatykę? Mając pewne dane wejściowe, zaczyna się od pierwszej reguły, Expression (wyrażenie), i przeszukuje kolejne, znajdując te, które pasują do tokenów w miarę ich wczytywania. Wczytywanie strumienia tokenów zgodnie z zasadami gramatyki nazywa się **parsowaniem** (analizą składniową), a program, który to robi, nazywamy **parserem** (ang. *parser*) lub **analizatorem składni** (ang. *syntax analyzer*). Nasz analizator odczytuje tokeny od lewej do prawej, dokładnie w takiej kolejności, jak je wpisujemy i czytamy. Wypróbujemy jakiś bardzo prosty przykład: czy 2 jest wyrażeniem?

1. Wyrażenie (Expression) musi być składnikiem (Term) lub kończyć się składnikiem. Składnik musi być czynnikiem (Primary) lub kończyć się czynnikiem. Ten czynnik musi zaczynać się znakiem (lub być liczbą (Number). Oczywiście 2 nie jest znakiem (, tylko literałem zmiennoprzecinkowym, a więc liczbą, która jest czynnikiem.
2. Przed tym czynnikiem (liczba 2) nie ma znaku /, * ani %, a więc jest to kompletny składnik (a nie zakończenie wyrażenia z operatorem /, * lub %).
3. Przed składnikiem tym (Primary 2) nie ma znaku + ani -, a więc jest to pełne wyrażenie (Expression), a nie zakończenie wyrażenia z operatorem + lub -.

W związku z tym zgodnie z naszą gramatyką 2 jest wyrażeniem. Przegląd gramatyki można przedstawić graficznie:

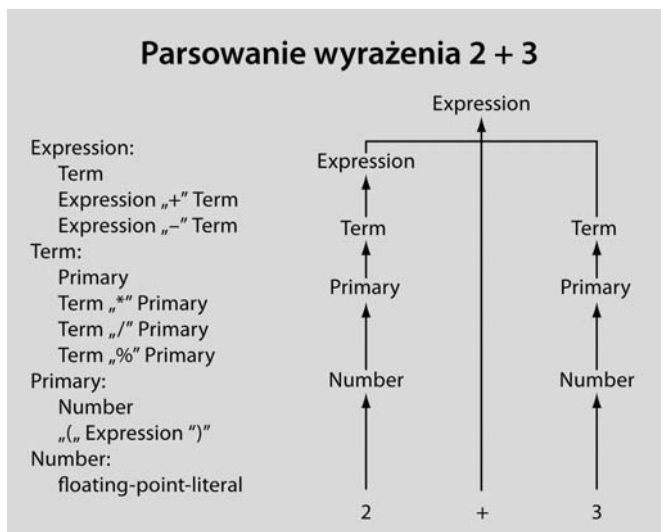


Na rysunku została przedstawiona ścieżka, którą przemierzaliśmy przez definicje. Odwracając nasze rozumowanie, możemy powiedzieć, że 2 jest wyrażeniem, ponieważ jest literałem zmiennoprzecinkowym, który jest liczbą, liczba jest czynnikiem, czynnik jest składnikiem, a składnik wyrażeniem.

Spróbujmy czegoś bardziej skomplikowanego. Czy $2+3$ jest wyrażeniem? Naturalnie znaczna część rozumowania będzie taka sam jak dla 2:

1. Wyrażenie musi być składnikiem lub mieć go na końcu. Składnik musi być czynnikiem lub kończyć się czynnikiem, który z kolei musi zaczynać się od znaku (lub być liczbą. Oczywiście 2 nie jest znakiem (, ale jest literałem zmiennoprzecinkowym, który jest liczbą, ta z kolei jest czynnikiem.
2. Przed tym czynnikiem (liczba 2) nie ma znaku /, * ani %, a więc jest to kompletny składnik (a nie zakończenie wyrażenia z operatorem /, * lub %).
3. Za składnikiem tym (Primary 2) jest znak +, a więc jest to koniec pierwszej części wyrażenia i musimy poszukać składnika za tym znakiem. Dokładnie w taki sam sposób, jak w przypadku 2 dowiadujemy się, że 3 jest składnikiem. Ponieważ za 3 nie ma znaku + ani -, uznajemy, że jest to pełny składnik, a nie pierwsza część wyrażenia z operatorem + lub -. W związku z tym $2+3$ spełnia zasadę `Expression+Term`, a więc jest wyrażeniem.

Znowu nasz tok rozumowania możemy przedstawić graficznie (pomijamy zasadę literału zmiennoprzecinkowego jako liczby dla uproszczenia):

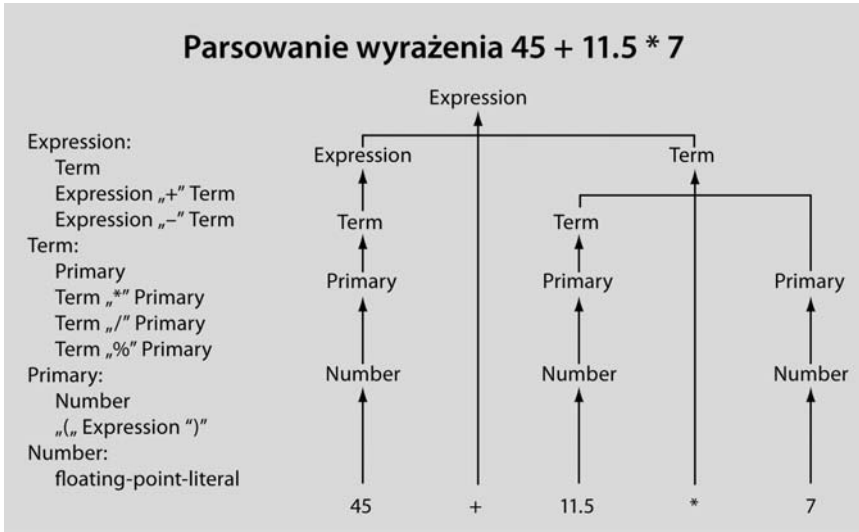


Na rysunku przedstawiliśmy ścieżkę, którą przemierzaliśmy przez definicje. Odwracając nasze rozumowanie, możemy powiedzieć, że $2+3$ jest wyrażeniem, ponieważ 2 jest składnikiem, który jest wyrażeniem, 3 jest składnikiem oraz wyrażenie ze znakiem + i składnikiem również jest wyrażeniem.

Prawdziwym powodem, dla którego zainteresowaliśmy się gramatykami, jest fakt, że mogą one nam pomóc w rozwiązaniu problemu poprawnego przetwarzania wyrażeń z operatorami * i +. Spróbujemy zatem przeanalizować wyrażenie $45+11.5*7$. Zabawa w komputer szczegółowo sprawdzający wszystkie reguły byłaby żmudna. Dlatego pominiemy niektóre pośrednie etapy, które opisaliśmy już przy analizie 2 i $2+3$. Oczywiście 45, 11.5 i 7 to literały zmiennoprzecinkowe, które są liczbami, liczby zaś są czynnikami, a więc możemy zignorować wszystkie reguły poniżej czynnika (Primary). Otrzymujemy:

1. 45 jest wyrażeniem, po którym znajduje się znak +, a więc szukamy wyrazu zamykającego regułę $\text{Expression}+\text{Term}$.
2. 11.5 jest składnikiem, po którym znajduje się znak *, a więc szukamy czynnika kończącego regułę $\text{Term}*\text{Primary}$.
3. 7 jest czynnikiem, a więc $11.5*7$ jest składnikiem zgodnie z regułą $\text{Term}*\text{Primary}$. Teraz widzimy, że $45+11.5*7$ jest wyrażeniem zgodnie z regułą $\text{Expression}+\text{Term}$. Mówiąc dokładniej, jest to wyrażenie, w którym najpierw jest wykonywane mnożenie $11.5*7$, a potem dodawanie $45+11.5*7$, dokładnie tak, jak gdybyśmy napisali $45+(11.5*7)$.

Na następnej stronie przedstawiamy graficzną ilustrację naszego rozumowania (znowu pomijamy zasadę literału zmiennoprzecinkowego jako liczby dla uproszczenia).



Powyższy rysunek obrazuje nasz tok rozumowania. Zauważ, jak reguła $\text{Term} * \text{Primary}$ zapewnia, że 11.5 zostanie pomnożone przez 7 zamiast dodane do 45.

Początkowo może wydawać Ci się to trudne do zrozumienia, ale ludzie czytają gramatyki, a te prostsze łatwo zrozumieć. Naszym celem nie było jednak nauczyć Cię rozumieć wyrażeń $2+2$ czy $45+11.5*7$. To już każdy potrafi. Szukaliśmy sposobu na objaśnienie komputerowi, co oznacza wyrażenie $45+11.5*7$ i wiele innych skomplikowanych wyrażeń, które mogą zostać mu podsunęte do obliczenia. Skomplikowane gramatyki nie nadają się do odczytu przez człowieka, ale komputery radzą sobie z nimi doskonale. Analizują ich reguły szybko i prawidłowo, a przychodzi im to z łatwością. W tym właśnie komputery są dobre — w dokładnym wykonywaniu poleceń.

6.4.1. Dygresja — gramatyka języka angielskiego

Jeśli nigdy dotąd nie miałeś do czynienia z gramatykami, to pewnie czujesz się oszołomiony. W istocie możesz czuć się niepewnie, nawet jeśli już się z czymś takim spotkałeś wcześniej. Spójrz na poniższy fragment gramatyki języka angielskiego:

```

Sentence :
  Noun Verb                               // np. C++ rules
  Sentence Conjunction Sentence           // np. Birds fly but fish swim
Conjunction :
  "and"
  "or"
  "but"
Noun :
  "birds"
  "fish"
  "C++"
Verb :
  "rules"
  "fly"
  "swim"

```

Zdania są zbudowane z części mowy (np. rzeczowników, czasowników i łączników). Można je przeanalizować pod kątem tych reguł, aby sprawdzić, które słowa są rzeczownikami, czasownikami itd. Ta prosta gramatyka zawiera także semantycznie bezsensowne zdania, tj. „C++ fly and birds rules”, ale poprawienie tego to całkiem inna kwestia, którą należałoby się zająć w znacznie bardziej zaawansowanej książce.

Wiele osób uczono tych podstawowych zasad na zajęciach z języka angielskiego w szkole. Istnieją nawet poważne dowody neurologiczne potwierdzające, że takie zasady są zakodowane w naszych mózгах.

Spójrz na podobne do wcześniejszych drzewo parsowania, ale tym razem przedstawiające proste angielskie zdanie:



To nie jest takie skomplikowane. Jeśli miałeś problemy ze zrozumieniem podrozdziału 6.4, wróć do niego teraz i przeczytaj go ponownie. Za drugim razem może być o wiele bardziej zrozumiałe!

6.4.2. Pisanie gramatyki

Skąd wzięliśmy te zasady gramatyki wyrażań? Trzeba przyznać, że pomogło nam w tym doświadczenie. Robimy to w taki sposób, w jaki ludzie zazwyczaj piszą gramatyki wyrażań. Napisanie prostej gramatyki jest łatwe. Wystarczy wiedzieć, jak:

1. Odróżnić zasadę od tokenu.
2. Wstawić jedną zasadę za drugą (**sekwencja**).
3. Wyrazić alternatywne wzorce (**alternacja**).
4. Wyrażać powtarzające się wzorce (**repetycja**).
5. Rozpoznać pierwszą zasadę gramatyki.



W różnych książkach i różnych systemach parsowania stosuje się różne notacje i terminologię. Na przykład niektórzy nazywają tokeny **znakami terminalnymi**, a zasady **znakami nieterminalnymi** lub **produkcjami**. My umieszczamy tokeny w podwójnych cudzysłowach i zaczynamy od pierwszej zasady. Alternatywy znajdują się w osobnych wierszach, np.:

```
List:
    "{" Sequence "}"
Sequence:
    Element
    Element " ," Sequence
Element:
    "A"
    "B"
```

Zatem sekwencja (Sequence) jest elementem (Element) lub elementem i sekwencją oddzielnymi od siebie przecinkiem. Element to litera A lub B. Lista (List) to sekwencja w nawiasach klamrowych. Możemy generować te listy (jak?):

```
{ A }
{ B }
{ A,B }
{A,A,A,A,B }
```

Poniższe natomiast nie są listami (dlaczego?):

```
{ }
A
{ A,A,A,A,B
{A,A,C,A,B }
{ A B C }
{A,A,A,A,B, }
```

Tej sekwencji nie nauczyłeś się w przedszkolu ani nie zakodowałeś w mózgu, chociaż nadal nie jest to nauka najwyższych lotów. W rozdziałach 7.4 i 7.8.1 objaśnimy sposoby wyrażania zasad syntaktycznych za pomocą gramatyk.

6.5. Zamiana gramatyki w kod

Istnieje wiele sposobów na zmuszenie komputera do respektowania zasad gramatyki. Użyjemy najprostszego — napiszemy po jednej funkcji dla każdej zasady i wykorzystamy nasz typ `Token` do reprezentowania tokenów. Program implementujący gramatykę często nazywa się **parserem**.

6.5.1. Implementowanie zasad gramatyki

Do zaimplementowania naszego kalkulatora potrzebujemy czterech funkcji — jednej do wczytywania tokenów i po jednej dla każdej zasady w gramatyce:

```
get_token()    // Wczytuje znaki i tworzy tokeny.
               // Wykorzystuje strumień cin.
expression()   // Obsługuje operatory + i -.
```

```

term()           // Wywołuje funkcje term() i get_token().
                // Obsługuje operatory *, /, i %.
primary()       // Wywołuje funkcje primary() i get_token().
                // Obsługuje liczby i nawiasy.
                // Wywołuje funkcje expression() i get_token().

```

Uwaga: każda funkcja zajmuje się określoną częścią wyrażenia i resztę zostawia pozostałym funkcjom. To radykalnie upraszcza kod funkcji. Metodę tę można porównać ze współpracą grupy ludzi, gdzie każdy wykonuje swoje zadanie, a problemy znajdujące się poza jego specjalnością przekazuje do rozwiązania innym.

Co powinny te funkcje robić? Każda funkcja powinna wywoływać inne funkcje gramatyki zgodnie z zasadą, którą implementuje, oraz funkcję `get_token()`, jeśli wymagany jest token. Jeśli na przykład funkcja `primary()` ma postąpić zgodnie ze swoją zasadą (`Expression`), musi wywołać:

```

get_token()     // Do obsługi znaków ( i ).
expression()    // Do obsługi wyrażenia.

```

Co powinny takie funkcje zwracać? Co z odpowiedzią, której w rzeczywistości oczekujemy? Dla wyrażenia $2+3$ funkcja `expression()` mogłaby zwrócić 5. Istotnie, wszystkie potrzebne informacje są dostępne. Tego spróbujemy! Dzięki temu unikniemy odpowiedzi na jedno z naszych najtrudniejszych pytań: „Jak zaprezentować wyrażenie $45+5/7$ w postaci danych, aby można było obliczyć jego wartość?”. Zamiast zapisywać reprezentację $45+5/7$ w pamięci, obliczamy wartość tego wyrażenia przy wczytywaniu. Ten mały pomysł stanowi prawdziwy przełom! Dzięki temu czterokrotnie zmniejszy się objętość kodu w stosunku do tego, co uzyskalibyśmy, gdyby funkcja `expression()` zwracała coś skomplikowanego do obliczenia później. Właśnie zaoszczędziliśmy sobie około 80 procent pracy.

Do towarzystwa nie pasuje funkcja `get_token()` — ponieważ obsługuje tokeny, a nie wyrażenia, nie może zwracać wartości podwyrażeń. Na przykład znaki `(` i `+` nie są wyrażeniami. Musi zatem zwracać typ `Token`. Dochodzimy do wniosku, że potrzebujemy następujących funkcji:

```

// Funkcje realizujące zasady gramatyczne:
Token get_token() // Wczytuje znaki i tworzy tokeny.
double expression() // Obsługuje znaki + i -.
double term() // Obsługuje znaki *, / i %.
double primary() // Obsługuje liczby i nawiasy.

```

6.5.2. Wyrażenia

Najpierw napiszemy funkcję `expression()`. Gramatyka dla niej jest następująca:

```

Expression:
  Term
  Expression '+' Term
  Expression '-' Term

```

Ponieważ to jest nasza pierwsza próba zamiany zasad gramatyki na kod, przedstawimy kilka błędnych rozwiązań. Tak to zazwyczaj wygląda przy stosowaniu nowych technik, a poza tym jest to dobry sposób na nauczenie się wielu rzeczy. W szczególności początkujący programista

może nauczyć się bardzo dużo, patrząc na dramatycznie różne zachowania podobnych fragmentów kodu. Czytanie kodu to przydatna umiejętność, którą należy pielęgnować.

6.5.2.1. Wyrażenia — pierwsza próba

Patrząc na zasadę **Expression'+' Term**, najpierw próbujemy wywołać funkcję `expression()`, następnie znaleźć znak `+` (`-`) oraz wywołać funkcję `term()`:

```
double expression()
{
    double left = expression(); // Wczytuje i oblicza wartość wyrażenia.
    Token t = get_token();      // następny token
    switch (t.kind) {           // Sprawdza, jaki to jest rodzaj tokenu.
        case '+':
            return left + term(); // Wczytuje i sprawdza wartość składnika,
                                  // następnie wykonuje dodawanie.
        case '-':
            return left - term(); // Wczytuje i sprawdza wartość składnika,
                                  // następnie wykonuje odejmowanie.
        default:
            return left;         // Zwraca wartość wyrażenia.
    }
}
```

Wygląda dobrze. To jest prawie naiwna transkrypcja gramatyki. Jest dość prosta — najpierw wczytuje wyrażenie, a następnie sprawdza, czy za nim znajduje się znak `+` lub `-` i jeśli tak, wczytuje składnik (`Term`).

Niestety w rzeczywistości to jest bez sensu. Skąd wiadomo, gdzie kończy się wyrażenie, dzięki czemu można poszukać znaków `+` i `-`? Przypominam, że nasz program czyta od lewej do prawej i nie może wybiec do przodu, aby sprawdzić, czy nie ma dalej znaku `+`. W istocie ta wersja funkcji `expression()` nigdy nie wyjdzie poza pierwszy wiersz — zaczyna od wywołania funkcji `expression()`, która zaczyna od wywołania funkcji `expression()` itd. w nieskończoność. Nazywa się to **nieskończoną rekurencją** (ang. *infinite recursion*), która jednak będzie miała koniec — gdy skończy się w komputerze pamięć do przechowywania nigdy niekończących się wywołań funkcji `expression()`. Pojęcie **rekurencji** oznacza wywoływanie funkcji przez siebie samą. Nie wszystkie rekurencje są nieskończone, poza tym technika ta jest bardzo przydatna w programowaniu (zobacz rozdział 8.5.8).

6.5.2.2. Wyrażenia — druga próba

Co w takim razie robimy? Każdy składnik jest wyrażeniem, ale nie każde wyrażenie jest składnikiem. Możemy więc zacząć od szukania składnika i przejść do szukania pełnego wyrażenia tylko wówczas, gdy znajdziemy znak `+` lub `-`. Na przykład:

```
double expression()
{
    double left = term();        // Wczytuje składnik i oblicza jego wartość.
    Token t = get_token();      // następny token
    switch (t.kind) {           // Sprawdza, jaki to rodzaj tokenu.
        case '+':
```



```

    return left + expression(); // Wczytuje wyrażenie i sprawdza jego wartość,
                                // a następnie wykonuje dodawanie.
case '-':
    return left - expression(); // Wczytuje wyrażenie i oblicza jego wartość,
                                // a następnie wykonuje odejmowanie.
default:
    return left;                // Zwraca wartość składnika.
}
}

```

To jakoś działa. Wypróbowaliśmy ten kod w ukończonym programie i stwierdziliśmy, że przetwarza wszystkie poprawne wyrażenia (i ani jednego niepoprawnego). W większości przypadków nawet zwraca prawidłowe wyniki. Na przykład wyrażenie $1+2$ jest wczytywane jako składnik (o wartości 1), po którym jest znak $+$ i wyrażenie (które w tym przypadku jest składnikiem o wartości 2) — zwracany wynik to 3. Analogicznie dla wyrażenia $1+2+3$ zwracana jest wartość 6. Możemy długo ciągnąć listę przypadków, w których kod ten zwraca prawidłowe wyniki, ale do rzeczy — co stanie się z wyrażeniem $1-2-3$? Funkcja `expression()` wczyta 1 jako składnik i przejdzie do $2-3$ jako wyrażenia (składającego się ze składnika 2 i wyrażenia 3). Następnie odejmie wartość wyrażenia $2-3$ od 1. Innymi słowy obliczy wartość wyrażenia $1-(2-3)$, która wynosi 2 (liczba dodatnia). Nas jednak w szkole uczono (a może i wcześniej), że $1-2-3$ oznacza tyle, co $(1-2)-3$, a więc wynik powinien być -4 (liczba ujemna).

W ten sposób utworzyliśmy bardzo fajny program, który nie robi tego, co trzeba. Sytuację pogarsza fakt, że w wielu przypadkach zwraca prawidłowy wynik. Na przykład dla wyrażenia $1+2+3$ zostanie zwrócony wynik 6, ponieważ $1+(2+3)$ jest równoważne z $(1+2)+3$. Jaki był nasz podstawowy z programistycznego punktu widzenia błąd? Zawsze należy zadać sobie to pytanie, gdy znajdzie się błąd. Dzięki temu będzie można uniknąć powtórzenia go innym razem.



Naszym problemem jest to, że popatrzyliśmy na kod i postanowiliśmy zgadywać. To rzadko wystarcza! Musimy rozumieć, co robi nasz kod, i umieć wyjaśnić, czemu robi to, co trzeba.

Ponadto analizowanie błędów jest często najlepszym sposobem na znalezienie poprawnego rozwiązania. Nasza funkcja `expression()` szuka najpierw składnika, a następnie, jeśli znajduje się za nim znak $+$ lub $-$, wyrażenia. Jest to w rzeczywistości implementacja nieco innej gramatyki:

```

Expression:
Term
Term '+' Expression // dodawanie
Term '-' Expression // odejmowanie

```

Różnica między tą a naszą gramatyką polega na tym, że my chcieliśmy, aby wyrażenie $1-2-3$ było traktowane jako wyrażenie $1-2$, po którym jest znak $-$ i składnik 3. Natomiast uzyskaliśmy składnik 1, po którym znajduje się znak $-$ i wyrażenie $2-3$. Innymi słowy chcieliśmy, aby $1-2-3$ oznaczało $(1-2)-3$, a nie $1-(2-3)$.

Tak, debugowanie bywa żmudne, trudne i czasochłonne, ale w tym przypadku pracujemy nad zasadami wyuczonymi w podstawówce. Sęk w tym, że musimy „wpoić” je komputerowi, który uczy się znacznie wolniej od nas.

Zauważ, że moglibyśmy zdefiniować $1-2-3$ jako $1-(2-3)$ zamiast $(1-2)-3$ i uniknąć całej tej dyskusji. Często najtrudniejszymi problemami w programowaniu są zasady, które zostały ustalone dla ludzi wiele lat przed tym, jak zaczęliśmy używać komputerów.

6.5.2.3. Wyrażenia — do trzech razy sztuka

Co teraz? Spójrz jeszcze raz na gramatykę (tę poprawną w podrozdziale 6.5.2) — każde wyrażenie zaczyna się składnikiem, po którym może znajdować się znak + lub -. Musimy więc poszukać składnika, sprawdzić, czy znajduje się za nim jeden z tych znaków, i robić tak aż do wyczerpania plusów i minusów. Na przykład:

```
double expression()
{
    double left = term();           // Wczytuje składnik i oblicza jego wartość.
    Token t = get_token();         // następny token
    while ( t.kind=='+' || t.kind=='-' ) { // Szuka znaków + i -.
        if (t.kind == '+')
            left += term();        // Oblicza wartość składnika i wykonuje dodawanie.
        else
            left -= term();        // Oblicza wartość składnika i wykonuje odejmowanie.
        t = get_token();
    }
    return left; // Jeśli nie ma więcej znaków + lub -, zwraca odpowiedź.
}
```

Ten kod jest nieco bardziej skomplikowany, ponieważ musieliśmy wprowadzić pętlę, aby móc wyszukać wszystkie plusy i minusy. Ponadto powtarzamy się trochę — dwa razy sprawdzamy znaki + i - oraz dwa razy wywołujemy funkcję `get_token()`. Ponieważ to tylko zaciemnia kod, pozbędziemy się nadmiarowego testu znaków + i -:

```
double expression()
{
    double left = term(); // Wczytuje składnik i oblicza jego wartość.
    Token t = get_token(); // następny token
    while(true) {
        switch(t.kind) {
            case '+':
                left += term(); // Oblicza wartość składnika i wykonuje dodawanie.
                t = get_token();
                break;
            case '-':
                left -= term(); // Oblicza wartość składnika i wykonuje odejmowanie.
                t = get_token();
                break;
            default:
                return left; // Jeśli nie ma więcej znaków + lub -, zwraca odpowiedź.
        }
    }
}
```

Warto zauważyć, że pomijając pętlę, ta wersja kodu jest podobna do pierwszej próby (podrozdział 6.5.2.1). Usunęliśmy wywołanie funkcji `expression()` wewnątrz funkcji `expression()` i zastąpiliśmy je pętlą. Innymi słowy przekonwertowaliśmy wyrażenie z zasad gramatyki na pętlę szukającą składnika ze znakiem + lub -.

6.5.3. Składniki

Zasada dotycząca składnika (Term) w gramatyce jest bardzo podobna do zasady wyrażenia (Expression):

```
Term:
  Primary
  Term '*' Primary
  Term '/' Primary
  Term '%' Primary
```

W konsekwencji kod również powinien być bardzo podobny. Oto pierwsza wersja:

```
double term()
{
  double left = primary();
  Token t = get_token();
  while(true) {
    switch (t.kind) {
      case '*':
        left *= primary();
        t = get_token();
        break;
      case '/':
        left /= primary();
        t = get_token();
        break;
      case '%':
        left %= primary();
        t = get_token();
        break;
      default:
        return left;
    }
  }
}
```

Niestety ten kod nie da się skompilować — operator reszty z dzielenia (%) nie jest zdefiniowany dla liczb zmiennoprzecinkowych. Zostaniemy o tym uprzejmie poinformowani. Odpowiadając wcześniej „oczywiście” na pytanie dotyczące tego, czy pozwolić też na wpisywanie liczb zmiennoprzecinkowych, nie przemyśleliśmy sprawy dokładnie i padliśmy ofiarą własnej **zachłanności na funkcje**. Tak jest **zawsze!** Co z tym zrobimy? Możemy sprawdzać, czy argumenty operatora % są liczbami całkowitymi i jeśli nie, zgłaszać błąd. Możemy też usunąć ten operator z kalkulatora. Wybierzemy najprostsze rozwiązanie. Dodamy ten operator później (rozdział 7.5).

Po usunięciu operatora % funkcja działa — tzn. prawidłowo analizuje składniki i oblicza ich wartości. Jednak doświadczony programista zauważy pewien niepożądany szczegół, który sprawia, że funkcja term() jest nie do przyjęcia. Co się stanie, jeśli ktoś wpisze 2/0? Nie można dzielić przez zero. Jeśli do tego dopuścimy, komputer wykryje to i zamknie program, wyświetlając raczej mało pomocny komunikat o błędzie. Niedoświadczony programista przekona się o tym na własnej skórze. Dlatego lepiej dodać odpowiedni test i wyświetlić lepszy komunikat o błędzie:



```

double term()
{
    double left = primary();
    Token t = get_token();
    while(true) {
        switch (t.kind) {
            case '*':
                left *= primary();
                t = get_token();
                break;
            case '/':
                { double d = primary();
                  if (d == 0) error("Dzielenie przez zero.");
                  left /= d;
                  t = get_token();
                  break;
                }
            default:
                return left;
        }
    }
}

```

Dlaczego umieściliśmy instrukcje obsługujące operator / w bloku? Ponieważ kompilator nalegał. Jeśli chcesz definiować i inicjować zmienne w instrukcjach switch, musisz umieścić je w blokach.

6.5.4. Podstawowe elementy wyrażeń

Zasada gramatyczna definiująca podstawowe czynniki wyrażeń również jest prosta:

```

Primary:
    Number
    '(' Expression ')'

```

Kod służący do jej implementacji jest nieco zagmatwany, ponieważ jest tu więcej okazji do popełnienia błędu składni:

```

double primary()
{
    Token t = get_token();
    switch (t.kind) {
        case '(': // Obsługa reguły '(' expression ')'.
            { double d = expression();
              t = get_token();
              if (t.kind != ')') error("Oczekiwano ')'.");
              return d;
            }
        case '8': // Za pomocą znaku '8' reprezentujemy liczby.
            return t.value; // Zwraca wartość liczby.
        default:
            error("Oczekiwano czynnika.");
    }
}

```

W zasadzie nie ma tu nic nowego, czego nie ma w funkcjach `expression()` i `term()`. Użyliśmy tych samych narzędzi językowych, zastosowaliśmy tę samą technikę wyodrębniania tokenów oraz te same techniki programistyczne.

6.6. Wypróbowywanie pierwszej wersji

Aby sprawdzić działanie tych funkcji, musimy zaimplementować funkcję `get_token()` i utworzyć funkcję `main()`. To drugie zadanie jest banalne — będziemy wywoływać funkcję `expression()` i drukować zwracany przez nią wynik:

```
int main()
try {
    while (cin)
        cout << expression() << '\n';
        keep_window_open();
}
catch (exception& e) {
    cerr << e.what() << endl;
    keep_window_open ();
    return 1;
}
catch (...) {
    cerr << "Wyjątek \n";
    keep_window_open ();
    return 2;
}
```

Błędy zostaną obsłużone w typowy sposób (rozdział 5.6.3). Opis implementacji funkcji `get_token()` odłożymy do podrozdziału 6.8, a teraz przetestujemy pierwszą wersję naszego kalkulatora.

WYPRÓBUJ

Pierwsza wersja kalkulatora (z funkcją `get_token()`) znajduje się w pliku o nazwie `calculator00.cpp`. Pobierz go i uruchom.

Wcale nie jesteśmy zaskoczeni tym, że program nie działa dokładnie tak, jak oczekiwaliśmy. Wzruszamy tylko ramionami i pytamy sami siebie: „Czemu nie?” albo lepiej: „Dlaczego działa tak, a nie inaczej?”. Wpisz liczbę 2 i znak nowego wiersza. Brak jakiegokolwiek reakcji. Daj jeszcze jeden znak nowego wiersza, aby sprawdzić, czy może zasnął. Nadal nic. Wpisz 3 i znak nowego wiersza. Zero reakcji! Wpisz 4 i znak nowego wiersza. Jest odpowiedź — 2! Teraz ekran wygląda tak:

```
2
3
4
2
```

Kontynuujemy, wpisując 5+6. Program zwraca w odpowiedzi 5, a więc na ekranie widzimy:

```
2
3
4
2
5+6
5
```

Jeśli nie masz doświadczenia programistycznego, pewnie jesteś bardzo skonsternowany! W istocie nawet doświadczony programista mógłby osłupieć. Co tam się dzieje? W tym momencie chcemy zamknąć program. Jak to zrobić? Zapomnieliśmy zaprogramować polecenie zamykające program, ale można go zmusić do zamknięcia za pomocą błędu. Wpisujemy więc *x* i program zakończy działanie komunikatem *Nieprawidłowy token*. W końcu coś zadziałało zgodnie z planem!

Zapomnieliśmy odróżnić na ekranie dane wejściowe od wyjściowych. Zanim przejdziemy do rozwiązywania zagadki, poprawimy to niedociągnięcie, aby lepiej widzieć, co się dzieje. Na razie wystarczy dodanie do danych wyjściowych znaku =:

```
while (cin) cout << "= " << expression() << '\n'; //Wersja 1
```

Teraz po wprowadzeniu dokładnie takiej samej sekwencji znaków, jak wcześniej otrzymamy następujący wynik:

```
2
3
4
= 2
5+6
= 5
x
Nieprawidłowy token
```

Dziwne! Spróbuj odgadnąć, co zrobił program. My wypróbowaliśmy jeszcze kilka innych liczb, ale poprzestańmy na tych, które mamy. To jest zagadka:

Dlaczego program nie zareagował na pierwsze liczby 2 i 3 i znaki nowego wiersza?

Dlaczego program zwrócił wynik 2 zamiast 4, gdy wpisaliśmy 4?

Dlaczego program zwrócił wynik 5 zamiast 11, gdy wpisaliśmy 5+6?

Jest wiele procedur postępowania w takich tajemniczych sytuacjach. Opiszemy je szczegółowo w następnym rozdziale, a na razie tylko pomyślimy. Czy możliwe, żeby program wykonywał nieprawidłowe działania arytmetyczne? To bardzo mało prawdopodobne. Wartość 4 to nie 2, a wyrażenia 5+6 to 11, a nie 5. Sprawdźmy, co się stanie, gdy wpisujemy 1 2 3 4+5 6+7 8+9 10 11 12 i nowy wiersz:

```

1 2 3 4+5 6+7 8+9 10 11 12
= 1
= 4
= 6
= 8
= 10

```

Co takiego? Nie 2 ani 3? Czemu 4, a nie 9 (bo 4+5)? Czemu 6, a nie 13 (bo 6+7)? Przyjrzyj się uważnie — program zwraca co trzeci token! Może „pożera” część wprowadzonych danych bez obliczania ich wartości? Tak właśnie robi. Spójrz na funkcję `expression()`:

```

double expression()
{
    double left = term(); // Wczytuje składnik i oblicza jego wartość.
    Token t = get_token(); // następny token
    while(true) {
        switch(t.kind) {
            case '+':
                left += term(); // Oblicza wartość składnika i wykonuje dodawanie.
                t = get_token();
                break;
            case '-':
                left -= term(); // Oblicza wartość składnika i wykonuje odejmowanie.
                t = get_token();
                break;
            default:
                return left; // Jeśli nie ma więcej znaków + lub -, zwraca odpowiedź.
        }
    }
}

```

Gdy funkcja `get_token()` zwróci inny Token niż `+` lub `-`, po prostu zwracamy wartość. Nie używamy tego tokenu i nie zapisujemy go do użytku przez inną funkcję. To nie było zbyt mądre. Odrzucanie danych wejściowych bez sprawdzenia nawet, co to dokładnie jest, to niezbyt dobry pomysł. Łatwo można się przekonać, że ten sam defekt ma funkcja `term()`. To wyjaśnia, dlaczego nasz kalkulator zjadał dwa tokeny na trzy.

Poprawimy funkcję `expression()`, aby nie zjadała tokenów. Gdzie możemy zapisać następny token (`t`), jeśli program go nie potrzebuje? Moglibyśmy opracować wiele różnych skomplikowanych rozwiązań, ale zastosujemy najbardziej oczywiste (wydaje się oczywiste, gdy się je już zobaczy) — ten token zostanie wykorzystany przez jakąś inną funkcję, a więc umieścimy go z powrotem w strumieniu wejściowym, aby mógł zostać stamtąd ponownie wczytany! W istocie można by było wstawiać znaki do strumienia `istream`, ale nie o to nam chodzi. My chcemy mieć tokeny, a nie bawić się ze znakami. Potrzebujemy strumienia wejściowego, w którym będzie można zapisywać tokeny.

Założmy więc, że mamy strumień tokenów — `Token_stream` — o nazwie `ts`. Przyjmijmy też, że ma on funkcję składową o nazwie `get()`, która zwraca następny token i funkcję składową `putback(t)`, ta zaś wstawia token `t` z powrotem do strumienia. Implementację strumienia `Token_stream` przedstawimy w podrozdziale 6.8, gdy będziemy już wiedzieli, jak go użyć.

Mając strumień `Token_stream`, możemy zmodyfikować funkcję `expression()` w taki sposób, żeby niepotrzebne tokeny zapisywała właśnie w nim:

```
double expression()
{
    double left = term(); // Wczytuje składnik i oblicza jego wartość.
    Token t = ts.get();   // Pobiera następny token ze strumienia tokenów.
    while(true) {
        switch(t.kind) {
            case '+':
                left += term(); // Oblicza wartość składnika i wykonuje dodawanie.
                t = ts.get();
                break;
            case '-':
                left -= term(); // Oblicza wartość składnika i wykonuje odejmowanie.
                t = ts.get();
                break;
            default:
                ts.putback(t); // Wstawia token t z powrotem do strumienia tokenów.
                return left;  // Jeśli nie ma więcej znaków + lub -, zwraca odpowiedź.
        }
    }
}
```

Takie same zmiany muszą zostać wprowadzone w funkcji `term()`:

```
double term()
{
    double left = primary();
    Token t = ts.get(); // Pobiera następny token ze strumienia tokenów.

    while(true) {
        switch (t.kind) {
            case '*':
                left *= primary();
                t = ts.get();
                break;
            case '/':
                { double d = primary();
                  if (d == 0) error("Dzielenie przez zero.");
                  left /= d;
                  t = ts.get();
                  break;
                }
            default:
                ts.putback(t); // Wstawia token t z powrotem do strumienia tokenów.
                return left;
        }
    }
}
```

W ostatniej funkcji naszego parsera, `primary()`, musimy tylko zmienić `get_token()` na `ts.get()`. Funkcja ta używa każdego wczytanego przez siebie tokenu.

6.7. Wypróbowywanie drugiej wersji

Możemy przetestować naszą drugą wersję programu. Wpisz 2 i znak nowego wiersza. Brak reakcji. Wpisz jeszcze jeden znak nowego wiersza, aby sprawdzić, czy program nie zasnął. Nadal nic. Wpisz 3 i znak nowego wiersza. Jest odpowiedź — 2. Spróbuj wyrażenia 2+2 ze znakiem nowego wiersza. Odpowiedź brzmi 3. Teraz na ekranie znajdują się następujące dane:

```
2
3
=2
2+2
=3
```

Hmm. Może wprowadzenie funkcji `putback()` i użycie jej w funkcjach `expression()` i `term()` nie pomogło w rozwiązaniu problemu? Spróbujmy czegoś innego:

```
2 3 4 2+3 2*3
=2
=3
=4
=5
```

Tak, to są poprawne odpowiedzi! Ale brakuje ostatniej (6). Wciąż mamy problem z tokenami, ale tym razem nie chodzi o ich zjadanie, lecz o to, że wynik jest zwracany dopiero po wpisaniu kolejnego wyrażenia. Program nie drukuje od razu wyniku. Opóźnia to do momentu wczytania pierwszego tokenu następnego wyrażenia. Niestety nie widzi go, dopóki nie naciśniemy klawisza *Enter* po wpisaniu następnego wyrażenia. Wniosek taki, że program nie działa źle, tylko odpowiada z opóźnieniem.

Jak to poprawić? Jednym z oczywistych rozwiązań jest wprowadzenie „polecenia drukowania”. Rolę tę niech pełni średnik, którego pojawienie się będzie oznaczało koniec wyrażenia i wymuszało wydruk wyniku. Przy okazji dodamy polecenie zamknięcia programu. Do tego celu doskonale nada się znak *k* (od słowa koniec). Obecnie w funkcji `main()` mamy taki kod:

```
while (cin) cout << "=" << expression() << '\n'; // wersja 1
```

Możemy zamienić ten na bardziej zagmatwany, ale bardziej użyteczny kod:

```
double val = 0;
while (cin) {
    Token t = ts.get();

    if (t.kind == 'k') break; // 'k' = koniec
    if (t.kind == ';' || t.kind == '!') // ';' = drukuj teraz
        cout << "=" << val << '\n';
    else
        ts.putback(t);
    val = expression();
}
```

Teraz kalkulator nadaje się do użytku. Zwróci na przykład następujące wyniki:

```
2;
= 2
2+3;
= 5
3+4*5;
= 23
k
```

W tym momencie mamy już dobrą wstępną wersję programu. Nie jest to jeszcze to, czego chcieliśmy, ale stanowi dobrą bazę do rozszerzania. Co ważniejsze, możemy poprawiać błędy i dodawać nowe funkcje, pracując już z działającym programem.

6.8. Strumienie tokenów

Zanim przejdziemy do dalszego opisu kalkulatora, przedstawimy implementację strumienia `Token_stream`. Jeśli nie dostarczymy poprawnych danych do programu, to nic nam się nie uda. Strumień ten zaimplementowaliśmy już na samym początku, ale najpierw chcieliśmy skupić się na problemach obliczeń w minimalnym rozwiązaniu.

Nasz kalkulator przyjmuje na wejściu sekwencję tokenów, np. $(1.5+4)*11$ z podrozdziału 6.3.3. Potrzebujemy czegoś takiego, co będzie wczytywać znaki ze standardowego strumienia wejściowego i podawać programowi kolejne tokeny, gdy ten o to poprosi. Poza tym zauważyliśmy, że program często czytuje o jeden token za dużo, a więc musimy wygospodarować jakieś miejsce, aby przechować go do późniejszego użytku. Jest to całkowicie normalne. Jeśli wczytujemy wyrażenie $1.5+4$ od lewej do prawej, skąd mamy wiedzieć, że liczba 1.5 to już cała liczba, jeśli nie wczytamy znajdującego się za nią znaku $+$? Dopóki go nie zobaczymy, równie dobrze możemy podejrzewać, że jesteśmy w trakcie wczytywania liczby 1.55555 . Dlatego potrzebujemy „strumienia”, który zwraca token, gdy go zażądamy za pomocą funkcji `get()`, i w którym możemy przechowywać nadmiarowe tokeny, zapisując je tam za pomocą funkcji `putback()`. Wszystko w języku C++ ma określony typ, dlatego musimy zacząć od zdefiniowania typu `Token_stream`.

Pewnie zauważyłeś słowo `public` w definicji typu `Token`. Tam nie miało to żadnego widocznego znaczenia. Natomiast w typie `Token_stream`, o którym teraz mowa, słowo to będzie miało ważne zastosowanie. W języku C++ typy zdefiniowane przez użytkownika często składają się z dwóch części — interfejsu publicznego (oznaczonego etykietą `public:`) i prywatnego (oznaczonego etykietą `private:`). Chodzi o oddzielenie tego, co jest potrzebne użytkownikowi do wygodnego korzystania z typu, od implementacji typu, w której użytkownik nie powinien mieć możliwości grzebania:

```
class Token_stream {
public:
    // interfejs użytkownika
private:
    // szczegóły implementacyjne
    // (bezpośrednio niedostępne użytkownikom typu Token_stream)
};
```

Oczywiście użytkownik i implementator to często jedna i ta sama osoba odgrywająca różne role. Należy jednak zaznaczyć, że rozróżnienie między interfejsem publicznym przeznaczonym

dla użytkowników i prywatnym dla implementatorów jest doskonałym narzędziem wspomagającym strukturalizację kodu. Interfejs publiczny powinien zawierać wyłącznie to, co jest użytkownikowi potrzebne, a więc najczęściej zestaw funkcji, także konstruktorów służących do inicjowania obiektów. Implementację prywatną stanowi kod tych publicznych funkcji. Najczęściej są to dane i funkcje wykonujące skomplikowane działania, o których użytkownik nie musi wiedzieć i których nie musi bezpośrednio używać.



Rozszerzymy trochę nasz typ `Token_stream`. Jakie wymagania powinien spełniać? Bez wątplenia potrzebujemy funkcji `get()` i `putback()`, ponieważ to one były powodem, dla którego w ogóle wymyśliliśmy coś takiego jak strumień tokenów. Zadaniem typu `Token_stream` jest wytworzenie tokenów ze znaków wczytywanych na wejściu. Zatem nasz strumień musi wczytywać dane ze strumienia `cin`. Oto najprostsza możliwa wersja typu `Token_stream`:

```
class Token_stream {
public:
    Token_stream();           // Tworzy obiekt typu Token_stream, który wczytuje dane ze strumienia
    cin.
    Token get();             // Daje token (obiekt typu Token).
    void putback(Token t);   // Wkłada token z powrotem.
private:
    // szczegóły implementacyjne
};
```

To wszystko, czego użytkownik potrzebuje do korzystania z typu `Token_stream`. Doświadczony programista mógłby się zastanawiać, dlaczego `cin` jest jedynym możliwym źródłem znaków — przypominamy, że zdecydowaliśmy się na razie pobierać dane tylko z klawiatury. Zrewidujemy tę decyzję w rozdziale 7.

Dlaczego zastosowaliśmy długawą nazwę `putback()` zamiast krótszej `put()`? Chcieliśmy podkreślić asymetrię między funkcjami `get()` i `putback()` — to jest strumień wejściowy, a nie coś, co można wykorzystać także do ogólnych celów związanych z wysyłaniem danych na wyjście. Poza tym w bibliotece `istream` także znajduje się funkcja `putback()`, a spójność nazw jest jedną z pożądanych cech każdego systemu. Dzięki temu łatwiej jest zapamiętywać te nazwy i unikać błędów.

Po tym wstępie możemy już utworzyć typ `Token_stream` i użyć go:

```
Token_stream ts;           // Obiekt typu Token_stream o nazwie ts.
Token t = ts.get();       // Daje następny token ze strumienia ts.
// ...
ts.putback(t);           // Wkłada token t z powrotem do strumienia ts.
```

Mamy już wszystko, czego potrzebujemy do napisania pozostałej części kalkulatora.

6.8.1. Implementacja typu `Token_stream`

Zaimplementujemy trzy wymienione funkcje strumienia `Token_stream`. Jak będzie się ten strumień prezentował? Tzn., jakie dane muszą być w nim przechowywane, aby spełniał swoje zadanie? Potrzebujemy miejsca do przechowywania tokenów, które włożymy do niego. Dla uproszczenia założymy, że można w nim przechowywać tylko jeden token na raz. Na potrzeby naszego programu to wystarczy (i na potrzeby wielu innych też). W związku z tym potrzebujemy

miejsca do przechowania jednego tokenu i wskaźnika oznaczającego, czy to miejsce jest wolne czy zajęte:

```
class Token_stream {
public:
    Token_stream();           // Tworzy obiekt typu Token_stream, który wczytuje dane ze strumienia
    cin.
    Token get();             // Daje token (funkcja get() została zdefiniowana w innym miejscu).
    void putback(Token t);   // Wkłada token z powrotem.
private:
    bool full;              // Informuje, czy w buforze jest token.
    Token buffer;           // Miejsce do przechowywania tokenu zapisanego w strumieniu za pomocą funkcji
                             // putback().
};
```

Teraz możemy zdefiniować (napisać) nasze trzy funkcje składowe. Konstruktor i funkcja `putback()` będą łatwe, ponieważ mają mało do zrobienia. Dlatego zajmiemy się nimi na początku.

Konstruktor ustawia tylko zmienną `full` na wartość oznaczającą, że bufor jest pusty:

```
Token_stream::Token_stream()
    :full(false), buffer(0) // Bufor jest pusty.
{
}
```

Jeśli definicja składowej znajduje się poza definicją klasy, należy wskazać, do której klasy ma należeć. Służy do tego następująca notacja:

```
nazwa_klasy::nazwa_składowej
```

Tutaj definiujemy konstruktor klasy `Token_stream`. Konstruktor to funkcja składowa o takiej samej nazwie, jak klasa, do której należy.

Po co definiować składową poza klasą? Przede wszystkim dla zachowania przejrzystości — w definicji klasy można znaleźć głównie informacje o tym, co klasa potrafi „robić”. Definicje funkcji składowych to implementacje określające sposób, w jaki są robione różne rzeczy. Wolimy umieścić je gdzieś indziej, aby nie przeszkadzały. Ideałem, do którego dążymy, jest, aby każda jednostka logiczna programu mieściła się w całości na ekranie. Definicje klas zazwyczaj spełniają ten wymóg, ale tylko jeśli definicje ich funkcji składowych przeniesie się gdzieś indziej, poza klasę.

Składowe klasy inicjuje się w liście inicjatorów składowych (podrozdział 6.3.3). Instrukcja `full(false)` ustawia składową `full` na `false`, a `buffer(0)` inicjuje składową `buffer` „udawanym tokenem”, który wymyśliliśmy na poczekaniu. Z definicji typu `Token` (podrozdział 6.3.3) wynika, że każdy token musi zostać zainicjowany, dlatego nie mogliśmy zignorować `Token_stream::buffer`.

Funkcja składowa `putback()` zapisuje swój argument w buforze strumienia `Token_stream`:

```
void Token_stream::putback(Token t)
{
    buffer = t; // Kopiuje t do bufora.
    full = true; // Bufor jest pełny.
}
```

Słowo kluczowe `void` (oznaczające „nic”) wskazuje, że funkcja `putback()` nie zwraca żadnej wartości. Aby upewnić się, że funkcja ta nie zostanie wywołana dwa razy bez odczytania (za pomocą funkcji `get()`) tego, co zostało zapisane w strumieniu między tymi wywołaniami, można dodać specjalny test:

```
void Token_stream::putback(Token t)
{
    if (full) error("Wywołanie funkcji putback(), gdy bufor jest pełny.");
    buffer = t; // Kopiuje t do bufora.
    full = true; // Bufor jest pełny.
}
```

Test składowej `full` polega na sprawdzeniu warunku wstępnego: „Nie ma żadnego tokenu w buforze”.

6.8.2. Wczytywanie tokenów

Całą prawdziwą pracę wykonuje funkcja `get()`. Jeśli w zmiennej `Token_stream::buffer` nie ma tokenu, funkcja ta musi wczytać znaki ze strumienia `cin` i złożyć z nich tokeny:

```
Token Token_stream::get()
{
    if (full) { // Sprawdzenie, czy jest gotowy token.
        // Usunięcie tokenu z bufora.
        full=false;
        return buffer;
    }

    char ch;
    cin >> ch; // Uwaga: >> pomija białe znaki (spacje, nowe wiersze, tabulatory itp.).

    switch (ch) {
        case ';': // drukowanie
        case 'k': // koniec
        case '(': case ')': case '+': case '-': case '*': case '/': case '%':
            return Token(ch); // Każdy znak reprezentuje sam siebie.
        case '.':
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
            {
                cin.putback(ch); // Wstawia cyfrę z powrotem do strumienia wejściowego.
                double val;
                cin >> val; // Wczytuje liczbę zmiennoprzecinkową.
                return Token('8',val); // '8' reprezentuje „liczbę”.
            }
        default:
            error("Nieprawidłowy token.");
    }
}
```

Przeanalizujemy szczegółowo funkcję `get()`. Najpierw sprawdza, czy w buforze jest token. Jeśli tak, zwraca go:

```
if (full) { // Sprawdzenie, czy jest gotowy token.
    // Usunięcie tokenu z bufora.
    full=false;
    return buffer;
}
```

Znakami musimy zajmować się tylko wówczas, gdy `full` ma wartość `false` (tzn. nie ma tokenu w buforze). W takim przypadku wczytujemy znak i postępujemy z nim odpowiednio do potrzeb. Szukamy nawiasów, operatorów i liczb. Jakikolwiek inny znak powoduje wywołanie funkcji `error()`, która zamyka program:

```
default:
    error("Nieprawidłowy token.");
}
```

Funkcja `error()` została opisana w rozdziale 5.6.3 i jest dostępna w pliku nagłówkowym `std_lib_facilities.h`.

Musieliśmy zdecydować się na jakiś sposób reprezentowania każdego rodzaju tokenu, tzn. trzeba było dobrać wartości dla składowej `kind`. Dla uproszczenia i ułatwienia debugowania zdecydowaliśmy się, że nawiasy i operatory same będą reprezentować swój rodzaj tokenu. Dzięki temu ich przetwarzanie jest bardzo łatwe:

```
case '(': case ')': case '+': case '-': case '*': case '/':
    return Token(ch); // Każdy znak reprezentuje sam siebie.
```

Mówiąc szczerze, w pierwszej wersji programu zapomnieliśmy o znakach: `';` oznaczającym drukowanie i `'k'` oznaczającym koniec programu. Dodaliśmy je, dopiero gdy były potrzebne w drugiej wersji.

6.8.3. Wczytywanie liczb

Zostały nam jeszcze liczby, z którymi wcale tak łatwo nie pójdzie. Jak dowiedzieć się, jaka jest wartość 123? To tyle samo, co $100+20+3$, ale co zrobić z 12.34? Jest też pytanie, czy zezwalać na stosowanie notacji naukowej, np. $12.34e5$. Aby poradzić sobie z tym wszystkim, moglibyśmy potrzebować wielu godzin, a nawet dni. Na szczęście nie jest tak źle. Strumienie w C++ „wiedzą”, jak wyglądają literały, i potrafią zamieniać je na wartości typu `double`. Musimy tylko znaleźć sposób na zmuszenie strumienia `cin` do robienia tego w funkcji `get()`:

```
case '.':
case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7':
case '8': case '9':
    { cin.putback(ch); // Wstawia cyfrę z powrotem do strumienia wejściowego.
      double val;
      cin >> val; // Wczytuje liczbę zmiennoprzecinkową.
      return Token('8',val); // '8' reprezentuje „liczbę”.
    }
```

Decyzja o wyborze znaku `'8'` do reprezentowania „liczb” w tokenach została podjęta arbitralnie.

Skąd wiadomo, że ma pojawić się liczba? Zgadując na podstawie własnych doświadczeń lub zaglądając do podręcznika języka C++ (np. w dodatku A), można stwierdzić, że literał liczbowy musi zaczynać się od cyfry lub znaku `.` (kropki dziesiętnej). Sprawdzamy więc, czy tak jest. Następnie chcemy, aby liczbę tę wczytał strumień `cin`, ale wczytaliśmy już pierwszy znak (cyfrę lub kropkę), a więc pozwalając temu strumieniowi zająć się resztą, uzyskalibyśmy nieprzewidywalne wyniki. Moglibyśmy spróbować połączyć wartość pierwszego znaku z „resztą” wczytaną przez `cin`. To znaczy, jeśli wpisano by 123, otrzymalibyśmy 1 i strumień `cin` wczytałby 23. Wówczas trzeba by było dodać 100 do 23. A fuj! A to tylko banalny przykład. Na szczęście (i nie przypadkowo) `cin` działa bardzo podobnie do `Token_stream` pod tym względem, że także pozwala wstawić odczytany znak z powrotem. Zamiast więc wykonywać zagmatwane obliczenia arytmetyczne, wstawiamy pierwszą cyfrę z powrotem do `cin` i odczytujemy z niego całą liczbę.

Zauważ, że cały czas unikamy skomplikowanej pracy i znajdujemy prostsze rozwiązania — często w tym celu wykorzystując zawartość biblioteki. To jest istota programowania — nieustanne poszukiwanie jak najprostszycy rozwiązań. Niektórzy — nie zawsze słusznie — wyrażają to słowami: „Dobry programista to leniwy programista”. W tym, i tylko w tym, sensie rzeczywiście powinniśmy być leniwi. Po co pisać dużo kodu, skoro można to samo zrobić, pisząc go mniej?



6.9. Struktura programu

Jest takie przysłowie, które głosi, że czasami trudno zauważyć las, ponieważ zasłaniają go drzewa. W analogiczny sposób można stracić z oczu program, patrząc na te jego wszystkie funkcje, klasy itp. Spójrzmy zatem na nasz program, pomijając chwilowo szczegóły:

```
#include "std_lib_facilities.h"

class Token { /*...*/ };
class Token_stream { /*...*/ };

Token_stream::Token_stream() :full(false), buffer(0) { /*...*/ }
void Token_stream::putback(Token t) { /*...*/ }
Token Token_stream::get() { /*...*/ }

Token_stream ts; // Udostępnia funkcje get() i putback().
double expression(); // Deklaracja umożliwiająca funkcji primary() wywoływanie funkcji expression().

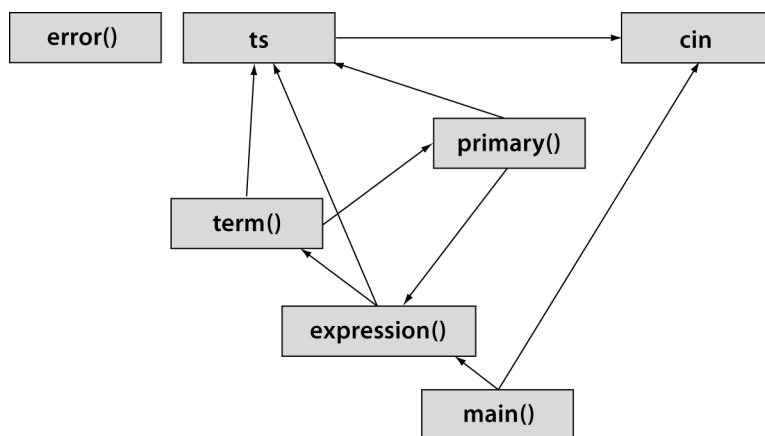
double primary() { /*...*/ } // Obsługa nawiasów i liczb.
double term() { /*...*/ } // Obsługa operatorów *, / i %.
double expression() { /*...*/ } // Obsługa operatorów + i -.

int main() { /*...*/ } // Główna pętla i obsługa błędów.
```

Kolejność deklaracji ma znaczenie. Nie można użyć nazwy, zanim zostanie zadeklarowana. Dlatego `ts` musi zostać zadeklarowana, zanim zostanie użyta w `ts.get()`, a funkcja `error()` musi zostać zadeklarowana przed funkcjami parsera, ponieważ wszystkie z niej korzystają. Na schemacie wywołań można zauważyć ciekawe zapętlenie: funkcja `expression()` wywołuje `term()`, która wywołuje `primary()`, a ta z kolei wywołuje `expression()`.



Można to przedstawić graficznie (wywołania funkcji `error()` odkładamy na bok, ponieważ wszystkie funkcje ją wywołują):



To oznacza, że nie możemy po prostu zdefiniować tych trzech funkcji — nie da się ustalić takiej kolejności, w której każda z nich zostałaby zdefiniowana przed pierwszym użyciem. Potrzebujemy przynajmniej jednej deklaracji, która nie jest równocześnie definicją. Zdecydowaliśmy się zadeklarować z wyprzedzeniem funkcję `expression()`.

Czy to działa? Jeśli odpowiednio zdefiniuje się słowo „działa”, można powiedzieć, że tak. Przechodzi kompilację, da się uruchomić, poprawnie oblicza wyniki wyrażeń i zgłasza sensowne komunikaty o błędach. Ale czy działa tak, jak sobie życzymy? Nie będzie zaskoczeniem, gdy powiem „nie za bardzo”. Pierwszą wersję wypróbowaliśmy w podrozdziale 6.6. Wówczas usunęliśmy poważny błąd. Druga wersja (podrozdział 6.7) nie jest o wiele lepsza. Ale w porządku, tego się spodziewaliśmy. Program zadowalająco spełnia swoje główne zadanie, czyli pozwala zweryfikować nasze podstawowe pomysły i zorientować się, co robić dalej. Pod tym względem odnieśliśmy sukces, ale spróbuj z niego skorzystać — bez problemu doprowadzi Cię do szafu!

WYPRÓBUJ

Uruchom powyższą wersję kalkulatora i sprawdź, co robi. Spróbuj dojść, dlaczego tak działa.

Ćwiczenia

Celem tego zestawu ćwiczeń jest poprawienie błędów w programie, aby zamienić go w coś użytecznego.

1. Weź kalkulator z pliku `calculator02buggy.cpp`. Spraw, żeby dał się skompilować. Musisz znaleźć i poprawić kilka błędów. Nie ma ich w tekście książki.
2. Zmień znak polecenia zamknięcia programu na `x`.

3. Zmień znak polecenia drukowania na =.
4. Dodaj do funkcji `main()` komunikat powitalny:

*„Witaj w naszym prostym kalkulatorze.
W wyrażeniach stosuj liczby zmiennoprzecinkowe.”*
5. Dodaj do komunikatu powitalnego informację o tym, jakie operatory są obsługiwane oraz jak drukować wynik i zakończyć działanie programu.
6. Znajdź trzy błędy logiczne w programie, które zostały tam przemyślnie ukryte, i popraw je, aby kalkulator zwracał prawidłowe wyniki.

Powtórzenie

1. Co rozumiemy pod pojęciem „Programować to zrozumieć”? Wymień trzy główne fazy produkcji oprogramowania.
2. W rozdziale tym został szczegółowo opisany proces tworzenia kalkulatora. Napisz krótką specyfikację wymagań dla takiego programu.
3. W jaki sposób dzieli się problem na mniejsze, łatwiejsze do ogarnięcia części?
4. Dlaczego utworzenie ograniczonej wersji programu jest dobrym pomysłem?
5. Co jest złego w mnożeniu wymagań dotyczących funkcjonalności na początku pracy nad programem?
6. Co to jest „przypadek użycia”?
7. Jaki jest cel przeprowadzania testów?
8. Posiłkując się informacjami zawartymi w rozdziale, opisz różnicę między składnikiem (Term), wyrażeniem (Expression), liczbą (Number) i czynnikiem (Primary).
9. Dane wejściowe kalkulatora rozkładaliśmy na następujące elementy: składnik, wyrażenie, czynnik i liczba. Rozłóż w ten sposób wyrażenie $(17+4)/(5-1)$.
10. Dlaczego w programie nie ma funkcji o nazwie `number()`?
11. Co to jest token?
12. Co to jest gramatyka? Co to jest reguła gramatyki?
13. Co to jest klasa? Do czego służą klasy?
14. Co to jest konstruktor?
15. Dlaczego w funkcji `expression()` klauzula `default` instrukcji `switch` wstawia token z powrotem do strumienia?
16. Co to znaczy „wczytać z wyprzedzeniem”?
17. Co robi funkcja `putback()` i dlaczego jest przydatna?
18. Co nastęcza trudności w implementacji operatora `%` (modulo) w funkcji `term()`?
19. Do czego służą dwie zmienne składowe klasy `Token`?
20. Dlaczego czasami składowe klasy dzieli się na publiczne i prywatne?
21. Co dzieje się w klasie `Token_stream`, gdy w buforze jest token i zostanie wywołana funkcja `get()`?
22. Po co zostały dodane znaki `';` i `'k'` do instrukcji `switch` w funkcji `get()` w klasie `Token_stream`?

23. Kiedy powinno się zacząć testowanie programu?
24. Co to jest „typ zdefiniowany przez użytkownika”? Do czego może się przydać?
25. Co to jest interfejs do „typu zdefiniowanego przez użytkownika” w języku C++?
26. Dlaczego powinno się używać kodu z bibliotek?

Terminologia

analiza	implementacja	przypadek użycia
analizator składniowy	interfejs	pseudokod
class	parser	public
dzielenie przez zero	private	składowa klasy
funkcja składowa	projekt	token
gramatyka	prototyp	zmienna składowa klasy

Praca domowa

1. Jeśli jeszcze tego nie zrobiłeś, rozwiąż wszystkie ćwiczenia *Wypróbuj*.
2. Dodaj możliwość używania w programie zarówno nawiasów okrągłych (), jak i kłamrowych {}, aby można było pisać wyrażenia typu $\{(4+5)*6\}/(3+4)$.
3. Dodaj operator silni ! jako operator przyrostkowy. Na przykład 7! oznacza $7*6*5*4*3*2*1$. Niech operator ten wiąże mocniej niż * i /. To znaczy, $7*8!$ powinno oznaczać $7*(8!)$, a nie $(7*8)!$. Zaczynj od dodania operatora wyższego poziomu do gramatyki. Aby pozostać w zgodzie ze standardową matematyczną definicją silni, niech 0! wynosi 1.
4. Zdefiniuj klasę Name_value przechowującą łańcuch i wartość. Utwórz konstruktor (podobny jak w klasie Token). Zmodyfikuj ćwiczenie 19. z rozdziału 4., używając vector<Name_value> zamiast dwóch wektorów.
5. Dodaj do gramatyki języka angielskiego z podrozdziału 6.4.1 przedimek **the**, aby można było za jej pomocą opisywać zdania typu „The birds fly but the Fish swim”.
6. Napisz program sprawdzający poprawność zdania zgodnie z gramatyką z podrozdziału 6.4.1. Przyjmij założenie, że każde zdanie kończy się kropką otoczoną białymi znakami, np. **birds fly but the fish swim** . jest zdaniem, a **birds fly but the fish swim** (brak kropki na końcu) i **birds fly but the fish swim.** (brak spacji przed kropką) nie. Dla każdego wpisanego zdania program niech zwraca tylko prostą odpowiedź Dobrze lub Źle. Wskazówka: nie zwracaj sobie głowy tokenami, wystarczy wczytać dane do łańcucha za pomocą operatora >>.
7. Napisz gramatykę dla wyrażeń logicznych. Są one podobne do arytmetycznych, tylko posługują się operatorami ! (nie), ~ (uzupełnienie), & (i), | (lub) oraz ^ (lub wyłączające). Operatory ! i ~ są jednoargumentowe i prefiksowe. Operator ^ ma pierwszeństwo przed | (podobnie jak * przed +), a więc $x|y^z$ oznacza $x|(y^z)$, a nie $(x|y)^z$. Operator & ma pierwszeństwo przed ^, a więc $x^y&z$ oznacza $x^(y&z)$.
8. Przerób grę „Byki i krowy” z ćwiczenia 12. w rozdziale 5., używając liter zamiast cyfr.
9. Napisz program wczytujący cyfry i składający z nich liczby całkowite. Na przykład 123 zostanie wczytane jako znaki 1, 2 i 3. Odpowiedź programu powinna być następująca:

Dekompozycja liczby 123: liczba setek: 1; liczba dziesiątek: 2; liczba jednostek: 3. Liczba ma być wysyłana na wyjście jako typ int. Obsłuż liczby jedno-, dwu-, trzy- i czterocyfrowe. Wskazówka: aby uzyskać całkowitoliczbową wartość 5 znaku '5', odejmij od niego '0', tzn. '5' - '0' = 5.

10. Permutacja to uporządkowany podzbiór pewnego zbioru. Wyobraź sobie na przykład, że chcesz zdobyć szyfr do sejf. Jest 60 możliwych liczb, a kombinacja, której potrzebujesz, składa się z trzech różnych liczb. Jest $P(60,3)$ permutacji dla tej kombinacji, gdzie P definiuje następujący wzór:

$$P(a, b) = \frac{a!}{(a - b)!}$$

We wzorze tym $!$ oznacza przyrostkowy operator silni. Na przykład $4!$ wynosi $4 \cdot 3 \cdot 2 \cdot 1$. Kombinacje są podobne do permutacji, z tą różnicą, że nie jest w nich ważna kolejność elementów. Gdybyś na przykład robił sobie deser lodowy, chcąc użyć trzech różnych smaków lodów z pięciu dostępnych, nie zależałoby Ci, czy bananowy znajdzie się na wierzchu czy na samym dole, oby gdzieś był. Wzór kombinacji jest następujący:

$$C(a, b) = \frac{P(a, b)}{b!}$$

Zaprojektuj program proszący użytkownika o podanie dwóch liczb, pytający, czy ma obliczyć permutacje czy kombinacje i drukujący wynik. To będzie wymagało podzielenia go na kilka części. Wykonaj analizę opisanych wyżej wymagań. Napisz, co dokładnie program ma robić. Napisz pseudokod i podziel go na części. Ten program powinien mieć wbudowany mechanizm sprawdzania błędów. Spraw, aby dla każdego rodzaju błędnych danych były zwracane odpowiednie komunikaty o błędzie.

Podsumowanie

Jedną z podstawowych czynności programistycznych jest odpowiednie rozpoznanie danych wejściowych. W taki czy inny sposób musi poradzić sobie z tym problemem każdy program. Do najtrudniejszych zadań należy rozszyfrowanie tego, co wytworzył bezpośrednio człowiek. Na przykład ciągle problemy sprawia wiele aspektów technologii rozpoznawania głosu. Proste wersje tego problemu, jak nasz kalkulator, można rozwiązać za pomocą gramatyk definiujących wprowadzane dane.

Skorowidz

- , 109, 110
-- , 109
! , 109
!= , 110
#define , 928, 930, 989
#else , 931
#endif , 931
#ifdef , 931
#include , 66, 78, 249, 255, 931, 989
#undef , 930
% , 110, 225
%= , 89
& , 529, 549, 836, 956, 960
&& , 110, 956, 961
() , 674
* , 108, 110, 534
*= , 89
*this , 601
/ , 110
/* */ , 232, 904, 946
// , 65, 904, 946
/= , 89
:: , 280, 296, 973
:alpha , 769
? : , 258, 957
[] , 534, 614
^ , 836, 956, 960
| , 836, 956, 960
|| , 110, 956, 961
~ , 836, 977
+ , 108, 110
++ , 89, 90, 109
+= , 89
< , 110
<< , 68, 110, 293, 338, 747, 750, 836
<= , 110
<algorithm> , 1008
<cctype> , 1024
<complex> , 803, 1030

<float.h> , 783
<limits.h> , 783
<limits> , 783, 835
<numeric> , 1031
<studio.h> , 924
= , 85
-= , 89
== , 110
> , 110
-> , 546, 973
>= , 110
>> , 79, 81, 88, 110, 129, 356, 367, 372, 750, 836, 1021

A

A Mathematical Programming Language , 735
abort() , 1040
abs() , 258, 802, 804, 1029, 1031
abstract , 450
abstrakcja , 33, 106, 464, 712
abstrakcja danych , 714, 715
abstrakcyjna klasa bazowa , 464
accumulate() , 668, 677, 1031
 uogólnienie , 678
akumulator , 677
acos() , 802, 1029
action , 721
Ada , 729
Ada Joint Program Office , 729
adaptery kontenerów , 1002, 1017
add() , 446
adjacent_difference() , 677, 1031
adjacent_find() , 1008
adres , 529, 825
advance() , 651, 999
AJPO , 729
akcja , 721
akumulacja , 677

akumulator , 677
Algol , 724
Algol W , 728
Algol60 , 724
Algol68 , 727
algorithm , 605
ALGOrithmic Language , 724
algorytmy , 132, 605, 667, 1008
 accumulate() , 668, 677, 1031
 adjacent_difference() , 677, 1031
 adjacent_find() , 1008
 akumulacja , 677
 binary_search() , 1012
 copy() , 668, 693, 1009
 copy_backward() , 1009
 copy_if() , 693, 696
 count() , 668, 1009
 count_if() , 668, 1009
 działające na zbiorach , 1013
 equal() , 668, 1008
 equal_range() , 668, 698, 1013
 fill() , 1011
 fill_n() , 1011
 find() , 668, 669, 698, 1008
 find_end() , 1009
 find_first_of() , 1008
 find_if() , 668, 671, 1008
 for_each() , 1008
 generate() , 1011
 generate_n() , 1011
 iloczyn skalarny , 679
 includes() , 1014
 inner_product() , 668, 677, 680, 1031
 inplace_merge() , 1013
 iter_swap() , 1011
 kopiowanie , 693
 lexicographical_compare() , 1015
 lower_bound() , 1012

- algorytmy
 - make_heap(), 1014
 - max(), 1015
 - max_element(), 1015
 - merge(), 668, 1013
 - min(), 1015
 - min_element(), 1015
 - mismatch(), 1008
 - modyfikujące sekwencje, 1009
 - next_permutation(), 1015
 - niemodyfikujące sekwencje, 1008
 - nth_element(), 1012
 - numeryczne, 676, 1031
 - operacje sterzy, 1014
 - pair(), 1013
 - partial_sort(), 1012
 - partial_sort_copy(), 1012
 - partial_sum(), 677, 1031
 - partition(), 1013
 - permutacje, 1015
 - pomocnicze algorytmy, 1011
 - pop_heap(), 1014
 - prev_permutation(), 1015
 - push_heap(), 1014
 - remove(), 1010
 - remove_copy(), 1010
 - remove_copy_if(), 1010
 - replace(), 1010
 - replace_copy(), 1010
 - reverse(), 1010
 - reverse_copy(), 1010
 - rotate(), 1010
 - search(), 1008, 1009
 - search_n(), 1009
 - serach(), 698
 - set_difference(), 1014
 - set_intersection(), 1014
 - set_symmetric_difference(), 1014
 - set_union(), 1014
 - sort(), 668, 697, 1012
 - sort_heap(), 1014
 - sortowanie, 697, 1012
 - stable_partition(), 1013
 - stable_sort(), 1012
 - swap(), 1011
 - swap_ranges(), 1011
 - TEA, 847
 - transform(), 1009
 - uninitialized_copy(), 1011
 - uninitialized_fill(), 1011
 - unique(), 1009, 1010
 - unique_copy(), 668, 693, 695, 1010
 - upper_bound(), 1012
 - wyszukiwanie, 669, 671, 697, 1012
 - algorytmy ogólne, 634
 - aliasy, 988
 - allocator, 611
 - allocator_type, 1004
 - alnum, 769, 1027
 - alokacja obiektów w pamięci wolnej, 533
 - alokacja pamięci, 819, 918, 961
 - alokacja zmiennych w pamięci, 531
 - alokator, 612
 - alpha, 1027
 - alternacja, 193
 - AMPL, 735
 - analiza, 55, 177
 - analiza składniowa, 189
 - analiza tekstu, 746
 - analizator składni, 189
 - AND, 110, 956
 - ANFOR, 944
 - ANSI, 730
 - aparaty cyfrowe, 810
 - aplikacje, 1044
 - append(), 747, 1025
 - application operator, 674
 - arg(), 804, 1031
 - argumenty, 105, 123, 141, 261
 - domyślne argumenty, 477, 970
 - formalne argumenty, 123
 - konwersja, 271
 - main(), 945
 - przekazywanie przez referencję, 267
 - przekazywanie przez stałą referencję, 265
 - przekazywanie przez wartość, 264
 - szablony, 157, 610, 984
 - array, 577, 660, 738, 784, 1001
 - Array_ref, 830
 - artificial intelligence, 724
 - arytmetyczny if, 258
 - arytmetyka modularna, 842
 - arytmetyka wskaźników, 579
 - as_bytes(), 364
 - as_int(), 416
 - ASCII, 354, 948
 - asctime(), 1039
 - assembler, 718
 - asercje, 165, 817
 - asin(), 802, 1029
 - aspect-based programming, 714
 - assert, 165
 - assert(), 933
 - assertion, 165
 - associative container, 681
 - AT&T Bell Labs, 735
 - AT&T Labs, 735
 - at(), 1005
 - atan(), 802, 1030
 - atof(), 1037
 - atoi(), 1037
 - atol(), 1037
 - attach(), 393, 437, 446
 - auto_ptr, 621, 622
 - autokod, 718
 - automatic garbage collection, 539
 - automatic storage, 532
 - automatyczne usuwanie nieużytków, 539
 - auxiliary function, 311
 - awaria, 816
 - AWK, 735, 759
 - Axis, 389, 392, 408, 475, 479
- ## B
- Babbage Charles, 729
 - back(), 662, 1005
 - back_inserter(), 1016
 - back_substitution(), 798
 - Backus John, 721
 - Backus-Naur Form, 721
 - bad(), 330, 1020
 - bad_alloc, 961, 996
 - bad_cast, 996
 - bad_expression, 769
 - bad_from_string, 749
 - Bad_image, 438
 - Bad_size, 986
 - bad_typeid, 996
 - bajt, 834
 - Basic CPL, 735
 - basic guarantee, 621
 - BCPL, 735

- begin, 638
 - begin(), 130, 746, 752, 998, 1005, 1025
 - Bell Labs, 735, 900
 - Bentley John, 816
 - beźpośrednie wyrażanie myśli, 711
 - biała skrzynka, 868
 - białe znaki, 81, 374
 - biblioteka standardowa C, 904
 - czas, 1038
 - daty, 1038
 - funkcje, 1032
 - łańcuchy, 1036
 - pamięć, 1037
 - pliki, 1032
 - printf(), 1033
 - biblioteka standardowa C++, 323, 525, 992
 - algorytmy, 668, 993, 1008
 - algorytmy ogólne, 634
 - erase(), 654
 - insert(), 654
 - iteratory, 637, 993, 997
 - kontenery, 526, 634, 660, 993, 1001
 - list, 653
 - listy powiązane, 641
 - obliczenia, 1028
 - obsługa błędów, 995
 - operowanie na danych, 634
 - pliki nagłówkowe, 993
 - przetwarzanie łańcuchów, 994, 1024
 - sekwencje, 637
 - słowniki, 681
 - string, 653
 - strumienie, 1018
 - strumienie wejścia i wyjścia, 993
 - tablice, 658
 - vector, 653
 - wektor, 656
 - wyjątki, 996
 - biblioteka STL, 1016
 - adaptery, 1017
 - działania arytmetyczne, 1017
 - obiekty funkcyjne, 1016
 - pair, 1017
 - predykaty, 1016
 - wstawianie, 1016
 - biblioteki, 70, 178, 852
 - FLTK, 387, 1047
 - graficzne, 383
 - GUI, 383, 384, 387
 - Matrix, 786
 - wejście i wyjście, 354
 - bidirectional iterator, 661, 663
 - binarne drzewa
 - zrównoważone, 684
 - binarne operacje wejścia i wyjścia, 363
 - binary balanced tree, 684
 - binary_search(), 698, 869, 1012
 - bind1st(g,x), 1017
 - bind2nd(g,y), 1017
 - bit, 94, 834
 - bitset, 839
 - flagi, 844
 - manipulowanie, 844
 - operacje, 835
 - bit znaku, 840
 - bitowe operacje logiczne, 960
 - bitset, 838, 839
 - black-box testing, 868
 - blank, 769, 1027
 - bloki, 121, 163, 279
 - zagnieżdżone bloki, 260
 - błędy, 71, 140
 - asercje, 165
 - debugowanie, 161
 - komunikaty, 163
 - obsługa, 151, 220
 - podejścia wytwarzania oprogramowania, 141
 - pomyłka o jeden, 153
 - raportowanie, 149, 299
 - specyfikacja, 141
 - szacowanie, 159
 - testowanie, 168
 - usuwanie, 161
 - wartości brzegowe, 153
 - warunki końcowe, 167
 - warunki wstępne, 165
 - wyjątki, 151
 - zawężanie zakresu, 156
 - źródła błędów, 141
 - błędy czasu wykonania, 70, 146
 - raportowanie błędów, 149
 - rozwiązywanie problemu przez wywołującego, 147
 - rozwiązywanie problemu przez wywołwanego, 148
 - błędy działania, 140
 - błędy kompilacji, 70, 140, 142
 - błędy konsolidacji, 70, 140, 145
 - błędy logiczne, 70, 140, 142, 157
 - przykłady, 157
 - błędy przejściowe, 535, 817
 - błędy składni, 140, 142
 - błędy typów, 140, 143
 - błędy zakresu, 152, 153
 - błękitna księga, 45
 - BNF, 721
 - bool, 80, 83, 93, 835, 903, 965
 - boolalpha, 1022
 - boost::regex, 759, 762
 - bounds error, 153
 - break, 116, 118, 963
 - broadcast function, 791
 - bsearch(), 1040
 - BSI, 944
 - budowa, 47
 - budowanie programu wykonywalnego, 1046
 - bufor, 324
 - buforowanie, 324, 376
 - bug, 161, 722
 - Build Solution, 1046
 - Burroughs Corporation, 726
 - Button, 409, 505, 506, 1057
 - bzip2, 719
- ## C
- C, 20, 733, 899
 - <studio.h>, 924
 - biblioteka standardowa, 904
 - const, 922
 - definicje, 914
 - deklaracje, 914
 - enum, 917
 - funkcje, 905
 - komentarze, 904
 - kompilacja warunkowa, 931
 - kontenery intruzyjne, 932
 - konwersja typu void*, 916
 - łańcuchy, 919, 922
 - makra, 927, 928
 - makra składniowe, 930
 - operacje bajtowe, 922

C

pamięć wolna, 918
 pliki, 927
 przeciążanie funkcji, 906
 przestrzenie nazw, 917
 przestrzeni znaczników
 struktur, 912
 rzutowanie, 915
 słowa kluczowe, 913
 sprawdzanie typów
 argumentów funkcji, 906
 stałe, 927
 stdio, 924
 struktury, 912
 styl, 923
 styl definiowania, 908
 wejście, 925
 wskaźniki, 923
 wskaźniki na funkcje, 911
 wyjście, 924
 wywołanie C++, 909
 zgodność z językiem C++,
 901
 C#, 20, 728, 739
 c_str(), 363, 366, 746, 1025
 C++, 20, 736, 737, 900
 C++0x, 737, 738
 C++98, 737
 C89, 900, 903, 914
 C99, 900
 calc(), 538
 callback, 497, 501
 calloc(), 1038
 Cambridge University, 717
 canvas, 388
 capacity(), 598, 1006
 case, 116, 227, 963
 cast, 547
 CAT, 810
 catch, 151, 156, 234, 963
 cb_next(), 502, 503, 510, 1052
 cb_quit(), 510
 CCA, 846
 cechy iteratorów, 890
 ceil(), 802, 1029
 center(), 430
 centrale telefoniczne, 810
 cerr, 1019
 char, 80, 83, 93, 835, 965
 CHAR_BIT, 1029
 CHAR_MAX, 1029

CHAR_MIN, 1029
 chronione składowe, 458
 cin, 79, 129, 132, 1019
 cin.clear(), 334
 cin.unsetf(), 357
 cipher(), 851
 Circle, 56, 400, 408, 430, 431,
 452, 459
 draw_lines(), 460
 class, 184, 289, 857, 971, 976
 class template, 605
 classical_elimination(), 798
 clear(), 331, 1006
 clock(), 894, 1039
 clock_t, 1038
 CLOCKS_PER_SEC, 894
 close(), 327, 328
 Closed_polyline, 408, 419
 Cmp_by_addr, 676
 Cmp_by_name, 676
 cntrl, 769, 1027
 COBOL, 161, 721, 739
 CODASYL, 722
 code storage, 532
 Color, 56, 389, 408, 414, 415, 444
 Common Business-Oriented
 Language, 721
 compacting garbage collection,
 821
 complex, 803
 Complex, 56
 complex<double>, 1030
 complex<float>, 1030
 complex<long double>, 1030
 complex<Scalar>, 1030
 computer science, 725
 computing, 725
 conforming, 944
 conj(), 804, 1031
 const, 109, 228, 253, 903, 927,
 960, 965
 const_cast, 547, 548, 903, 954,
 962
 CONST_CAST, 916
 const_iterator, 661, 1004
 const_pointer, 1004
 const_reference, 1004
 const_reverse_iterator, 1004
 constant expression, 960
 constraints-based
 programming, 714

constructor, 294
 continue, 963
 convenience function, 311
 copy constructor, 566
 copy(), 566, 668, 693, 905, 1009
 copy_backward(), 1009
 copy_if(), 693, 696
 cos(), 802, 1029
 cosh(), 802, 1030
 count(), 668, 1009
 count_if(), 668, 1009
 cout, 65, 78, 79, 132, 1019
 CPL, 735
 Cricle, 429
 ctime(), 1039
 czarna skrzynka, 868
 czas, 1038
 czas trwania, 950, 952
 część rzeczywista, 803, 804,
 1031
 część urojona, 803, 804, 1031
 czyste funkcje wirtualne, 450,
 464
 czyste interfejsy, 465
 czytniki kodów kreskowych,
 810

D

d, 1027
 Dahl Ole-Johan, 731
 dane, 78, 342
 przetwarzanie, 630
 składowe, 289
 wejściowe, 56, 78, 104, 876
 wyjściowe, 354, 876
 datalogi, 725
 Date, 292, 312
 daty, 1038
 DBL_EPSILON, 1029
 DBL_MAX, 1029
 DBL_MIN, 1029
 dealokacja pamięci wolnej, 538
 debugging, 71, 161
 debugowanie, 71, 161, 891
 destruktory, 573
 kod GUI, 517
 konstruktory, 573
 zalecenia, 162
 dec, 355, 356, 357, 1023
 decipher(), 851

- dedukcja argumentów
 - szablonu, 610
 - dedukcja zakresu, 954
 - default, 963
 - definicja, 79, 80, 92, 236, 249, 250, 964
 - funkcje, 122, 123, 907
 - klasy, 289
 - struktury, 291
 - definiowanie
 - funkcje składowe, 296
 - funkcje wirtualne, 461
 - makra, 990
 - operatory, 958
 - operatory wejściowe, 339
 - operatory wyjściowe, 338
 - składowe klas, 975
 - deklaracja, 70, 92, 112, 249, 250, 964
 - funkcje, 125, 126, 968
 - stałe, 252
 - using, 280
 - zmiennne, 252
 - delete, 538, 539, 545, 618, 815, 819, 820, 903, 953, 955, 961
 - delete(), 655
 - delete[], 539, 541, 955
 - deque, 660, 1001
 - dereference operator, 530
 - dereferencja, 530, 534, 966
 - derywacja, 458
 - destroy(), 934
 - destructor, 541
 - destruktory, 540, 541, 571, 961, 977
 - debugowanie, 573
 - generowanie, 542
 - pamięć wolna, 542
 - wirtualne, 451, 543
 - deszyfrowanie, 847
 - device register, 846
 - Dialog_box, 56
 - difference(), 999
 - difference_type, 1003
 - difftime(), 1039
 - digit, 769, 1027
 - Dijkstra Edsger, 725, 726
 - Distribution, 487
 - divides<T>(), 1017
 - do, 962
 - dodawanie, 446
 - dokumenty HTML, 383
 - dołączanie plików, 66, 989
 - domyślna inicjacja, 254
 - domyślne argumenty, 970
 - domyślne konstruktory, 307
 - dopasowanie wyrażień
 - regularnych, 1025
 - dopełnienie, 836
 - dostawa na czas, 709
 - dostęp do danych, 631
 - dostęp do elementów
 - kontenera, 1005
 - dostęp do elementów wektora, 544, 575
 - dostęp do nazw z przestrzeni, 280
 - dostęp do nazwanych tablic, 578
 - dostęp do składowych, 184, 292, 463, 546, 955, 973
 - dostęp poprzez wskaźniki, 534
 - dostępność nazw, 258
 - double, 80, 83, 84, 93, 526, 780, 948, 965
 - double precision floating point number, 80
 - doubly-linked list, 551, 641
 - Dow Jones Industrial, 680
 - draw(), 454, 481, 606
 - draw_lines(), 414, 421, 426, 429, 434, 454, 460, 462
 - drzewa
 - czerwono-czarne, 681, 684
 - niezrównoważone, 685
 - zrównoważone, 685
 - duże zadania obliczeniowe, 106
 - dynamic dispatch, 458
 - dynamic_cast, 621, 815, 954, 961
 - dyrektywa using, 280
 - dyrektywy, 989
 - #define, 989
 - #include, 66, 67, 989
 - działania arytmetyczne, 84, 132
 - działania na liczbach, 779
 - dziedziczenie, 458, 550, 716, 832, 979
 - kontenery, 607
 - referencje, 550
 - wielokrotne, 979
 - wskaźniki, 550
 - dziedziczenie implementacji, 465
 - dziedziczenie interfejsu, 465
 - dziel i rządź, 106
- ## E
- ECMAScript, 764
 - edytor tekstu, 647
 - iteracja, 650
 - wiersze, 649
 - efekt przekroczenia zakresu liczby, 781
 - efekt stałości rozmiaru liczb całkowitych, 781
 - efektywność, 709
 - ekran, 388, 1019
 - elementy wyliczenia, 300
 - elementy wyrażień, 200
 - elim_with_partial_pivot(), 799
 - eliminacja Gaussa, 797
 - elipsa, 431
 - Ellipse, 400, 408, 431, 432
 - else, 113
 - else if, 115
 - empty(), 1006
 - encapsulation, 458
 - end, 638
 - end of file, 88
 - end(), 130, 746, 752, 998, 1005, 1025
 - endl, 1023
 - ends, 1023
 - enum, 300, 917, 972
 - EOF, 927, 1035
 - eof(), 330, 332, 366, 375, 1020
 - equal(), 668, 1008
 - equal_range(), 668, 698, 699, 755, 1007, 1013
 - equal_to<T>(), 1016
 - erase(), 654, 655, 656, 662, 747, 1006, 1025
 - errno, 802
 - error(), 147, 154, 332, 341, 927
 - estimation, 160
 - etapy rozwoju
 - oprogramowania, 177
 - Euler, 728
 - exit(), 1040
 - exp(), 802, 1030
 - explicit, 572

explicit type conversion, 547
 expN(), 482
 extern, 251
 extern "C", 909, 971

F

fac(), 482
 fail(), 330, 331, 332, 344, 1020
 Fast Light Tool Kit, 387, 1048
 fclose(), 927, 1032
 field, 360
 figury geometryczne, 388
 FILE, 927, 1032
 fill(), 1011
 fill_n(), 1011
 filozofie języków
 programowania, 706
 find(), 668, 669, 689, 698, 715,
 747, 905, 1007, 1008, 1025
 find_end(), 1009
 find_first_of(), 1008
 find_from_addr(), 754, 756
 find_if(), 668, 671, 1008
 find_subject(), 756
 fixed, 358, 359, 1023
 Fl::wait(), 504
 fl_arc(), 431
 Fl_Color, 416
 Fl_GIF_Image, 438
 Fl_JPEG_Image, 438
 fl_line(), 420
 Fl_linestyle, 417
 flagi, 844
 float, 780, 782, 948, 965
 floatfield, 358
 floor(), 802, 1029
 FLOW-MATIC, 722
 FLT_DIG, 1029
 FLT_MAX, 1029
 FLT_MAX_10_EXP, 1029
 FLT_MIN, 1029
 FLTK, 387, 401, 886, 1048
 instalacja, 1048
 pobieranie biblioteki
 z internetu, 1048
 Visual Studio, 1049
 flush, 1023
 FMA, 792
 fmtflags(), 358
 fopen(), 1032

for, 121, 903, 962
 for_each(), 1008
 formalna specyfikacja, 869
 format pliku, 325
 formatowane wstawianie, 1021
 formatowanie danych
 wyjściowych, 354, 366
 formaty liczb
 zmiennoprzecinkowych, 358
 Fortran, 20, 720, 739
 Fortran77, 720
 Fortran90, 720
 forward iterator, 663
 ForwardIterator, 869
 fprintf(), 925, 1033
 fragmentacja pamięci, 819, 820
 free(), 903, 918, 1038
 friend, 463, 975
 from_string(), 748, 749
 front(), 662, 1005
 front_inserter(), 1016
 fstream(), 1020
 Function, 408, 476, 482
 function activation record, 273
 function call operator, 674
 function template, 605
 functional programming, 714
 funkcje, 122, 229, 291, 857, 968
 argumenty, 123, 261
 biblioteka standardowa C,
 1032
 C, 905
 czysto wirtualne, 981
 definicja, 122, 123, 907
 deklaracja, 125, 968
 deklaracja argumentów, 261
 error(), 147
 inline, 298
 lista parametrów, 123
 lokalne funkcje, 260
 main(), 66
 matematyczne, 802, 1029
 parametry, 123, 262
 parametryzowane, 605
 pomocnicze, 310
 prototyp, 907
 przekazywanie argumentów
 przez referencję, 267
 przekazywanie
 argumentów przez stałą
 referencję, 265

przekazywanie argumentów
 przez wartość, 264
 rekord aktywacji, 273
 rekurencja, 275
 return, 263
 rozgłoszeniowe, 791
 sort(), 130
 sprawdzanie argumentów,
 271
 sprawdzanie typów
 argumentów, 906
 stos rekordów aktywacji,
 275
 szablony, 605
 szablonowe, 716
 treść, 123
 typ zwrotny, 261
 void, 262
 wynik działania, 123
 wywołanie, 128, 261, 272,
 968
 zagnieżdżone funkcje, 260
 zastosowanie, 124
 zwracanie wartości, 123, 263
 funkcje przecięzione, 969
 argumenty domyślne, 970
 dokładne dopasowanie, 969
 dopasowanie przy użyciu
 konwersji zdefiniowanych
 przez użytkownika, 969
 dopasowanie przy użyciu
 promocji, 969
 dopasowanie przy użyciu
 standardowych konwersji,
 969
 dopasowanie przy użyciu
 wielokropka, 969
 niepodawanie listy
 argumentów, 970
 rozpoznawanie, 969
 wielokropek, 970
 wywołanie, 969
 funkcje składowe, 259, 293
 definiowanie, 296, 297
 konstruktory, 294
 stałe funkcje składowe, 309
 funkcje wirtualne, 450, 456,
 458, 460, 716, 980
 wywołanie, 459
 funkcje zwrotne, 501

G

- g++, 34
- garbage collection, 539
- GB, 51
- GCC, 909
- gcount(), 1021
- general, 358, 359
- generalizacja, 464
- generate(), 1011
- generate_n(), 1011
- generator typów, 604
- generowanie
 - destruktory, 542
 - typy, 604
- get(), 750, 1021
- get_encoding(), 439
- get_int(), 749
- get_string(), 507
- getc(), 925, 1036
- getchar(), 925, 926, 1036
- getline(), 373, 747, 750, 1021
- gets(), 880, 925, 926
- GIF, 438
- gigabajt, 51
- Gill D. J., 718
- globalna inicjacja, 277
- gmtime(), 1039
- GNU C++, 34
- good(), 330, 332, 1020
- Google, 51
- goto, 963
- graficzne przedstawianie danych, 471
 - Axis, 479
 - Distribution, 487
 - etykiety, 491
 - Function, 476
 - odczytywanie danych z pliku, 487
 - osie, 479
 - rysowanie wykresów funkcji, 472
 - skalowanie danych, 489
 - tworzenie wykresu, 490
 - układ ogólny, 488
 - wartość przybliżona funkcji wykładniczej, 481
 - wykresy, 486
- graficzny interfejs użytkownika, 179, 382, 497
- grafika, 382, 400
 - Axis, 392
 - biblioteka GUI, 387
 - elipsa, 431
 - figury geometryczne, 388
 - Image, 399
 - kanwa rysunku, 388
 - klasy figur geometrycznych, 389
 - klasy graficzne, 407
 - kolory, 414
 - linia, 410
 - obrazy, 399, 436
 - okrąg, 430
 - operacje, 445
 - piksel, 388
 - Polygon, 389
 - prezentacja danych, 394
 - projektowanie klas, 444
 - prostokąty, 395, 422
 - punkt, 410
 - Rectangle, 396
 - rysowanie figur, 454
 - rysowanie wykresu funkcji, 394
 - Shape, 389, 448
 - tekst, 398, 428
 - wielokąty, 394, 420
 - współrzędne, 388
 - wypełnianie kolorem, 397
- gramatyka, 188, 191
 - elementy wyrażeń, 200
 - implementacja zasad gramatyki, 194
 - język angielski, 192
 - nieskończona rekurencja, 196
 - parser, 194
 - pisanie, 193
 - podstawowe czynniki wyrażeń, 200
 - produkcja, 194
 - składniki, 199
 - wyrażenia, 195, 196
 - zamiana w kod, 194
 - znaki nieterminalne, 194
 - znaki terminalne, 194
- graph, 769, 1027
- Graph_lib, 384, 390
 - Window, 500
- greater_equal<T>(), 1017
- greater<T>(), 1016
- grep, 759
- GUI, 179, 382, 497, 498
 - Axis, 392
 - biblioteka, 387
 - Button, 505, 506
 - debugowanie kodu, 517
 - figury geometryczne, 388
 - FLTK, 387
 - funkcje zwrotne, 501
 - Graph_lib, 384, 390
 - Image, 399
 - implementacja wywołań zwrotnych, 1052
 - In_box, 507
 - inwersja kontroli, 511
 - kanwa rysunku, 388
 - klasy figur geometrycznych, 389
 - klasy graficzne, 408
 - kontrolki, 505
 - Lines_window, 509, 510
 - menu, 507, 513
 - Menu, 507
 - nagłówki graficzne, 390
 - obrazy, 399
 - okno, 385, 390, 500
 - Out_box, 507
 - pętla oczekująca, 504
 - piksel, 388
 - pliki źródłowe, 402
 - Point, 384
 - Polygon, 385, 389, 394
 - prostokąty, 395
 - przycisk Next, 499
 - przyciski, 506
 - punkt, 384
 - Rectangle, 396
 - Shape, 389
 - Simple_window, 390, 500
 - tekst, 398
 - testowanie, 883
 - uruchamianie programu, 401
 - Vector_ref, 1055
 - wait_for_button(), 390
 - Widget, 505, 1053
 - widgety, 505, 1056
 - wielokąty, 394
 - Window, 1054
 - współrzędne, 388

GUI

- wypełnianie kolorem, 397
- wyświetlanie obiektów na ekranie, 386

- gwarancja niezgłaszania wyjątku, 621
- gwarancje, 620

H

- hash table, 681, 689
- hash value, 689
- hash_map, 691
- haszowanie, 690
- header, 66
- header file, 66
- heap, 1014
- Hejlsberg Anders, 728
- helper function, 311
- hermetryzacja, 458
- hex, 355, 356, 357, 947, 1023
- hide(), 505
- hierarchia klas, 464
- hierarchia strumieni wejścia i wyjścia, 1019
- high(), 715
- historia języków programowania, 717
- History of Programming Languages, 717
- Hopper Grace, 161, 722
- HTML, 383

I

- I/O, 105
- Ichbiah Jean, 729
- IDE, 34, 70, 1044
- ideały programistyczne, 54, 708
 - bezpośrednie wyrażanie myśli, 711
 - minimalizm, 713
 - modułowość, 712
 - poziom abstrakcji, 712
 - spójność, 713
- identyfikatory, 950
 - słowa kluczowe, 950
- if, 113, 880, 962
 - else, 113
 - else if, 115

- IFIP, 724
- ifstream, 326, 329
- iloczyn logiczny, 110
- iloczyn skalarny, 679
- iloczyn wewnętrzny, 679
- imag(), 803, 804, 1031
- Image, 399, 408, 436
- implementacja, 55, 177, 289
- implementacja GUI, 1051
- implementacja tokenów, 183
- implementacja wywołań funkcji, 272
- implementacja wywołań zwrotnych, 1052
- implementacja zasad gramatyki, 194
- implementation defined, 944
- importowanie obrazów z plików, 399
- In_box, 409, 507
- INCITS, 944
- includes(), 1014
- indeksowanie, 534, 966
- infinite recursion, 196
- informacja zwrotna, 55
- informacje, 51
- informacje o stanie strumienia, 331
- informatyka, 35, 45, 725
- inicjacja, 85, 87, 536
 - obiekty, 294
 - składowe, 979
 - tablice, 582
 - wartości domyślne, 254
 - wskaźniki, 536
- inicjatory składowych, 979
- inkrementacja, 89, 110
- inline function, 903, 905
- inner product, 680
- inner_product(), 668, 677, 680, 1031
 - uogólnienie, 681
- inplace_merge(), 1013
- input iterator, 663
- insert(), 654, 655, 656, 662, 686, 687, 746, 747, 1006, 1025
- inserter(), 1016
- instalacja
 - FLTK, 1047, 1048
 - Visual Studio, 1044

- instrukcje, 66, 67, 69, 112, 962
 - if, 113
 - porządek wykonywania, 276
 - pusta instrukcja, 113
 - wyrażeniowe, 112
 - złożone, 121
- instrukcje dla konsolidatora, 971
- instrukcje maszynowe, 718
- int, 66, 68, 78, 80, 82, 83, 84, 92, 780, 782, 835, 965
- INT_MAX, 1029
- INT_MIN, 1029
- integrated development environment, 34, 70
- inteligentne karty, 810
- interactive development environment, 70
- interaktywne środowisko programistyczne, 70, 1044
- interfejs, 289
- interfejs klasy, 303
- interfejs prywatny, 206
- interfejs publiczny, 206, 289
- interfejs użytkownika, 498
 - GUI, 498
 - konsola, 498
 - przeglądarki internetowe, 498
- internal, 1023
- intersect(), 421
- intrusive container, 932
- invalid_argument, 996
- invariant, 295
- inwersja kontroli, 511
- inżynieria oprogramowania, 33
- ios_base, 331
- ios_base::app, 361, 1020
- ios_base::ate, 361, 1020
- ios_base::badbit, 332
- ios_base::binary, 361, 363, 1020
- ios_base::failure, 332, 996
- ios_base::in, 361, 1020
- ios_base::out, 361, 1020
- ios_base::trunc, 361, 1020
- iostream, 332, 370, 750, 765
- is_open(), 1020
- is_palindrome(), 587, 588
- is_whitespace(), 373
- isalnum(), 369, 1024
- isalpha(), 240, 369, 758, 1024

isctrl(), 369, 1024
 isdigit(), 369, 758, 1024
 isgraph(), 369, 1024
 islower(), 369, 1024
 isprint(), 369, 1024
 ispunct(), 369, 1024
 isspace(), 369, 1024
 istream, 323, 324, 325, 329, 370,
 750, 1019
 istream_iterator, 694
 istream_iterator<>, 694
 istringstream, 365, 371
 isupper(), 369, 1024
 isxdigit(), 369, 1024
 iter_swap(), 1011
 iteracja, 119, 132
 iteration, 119
 iterator, 646, 661, 1003
 iterator traits, 890
 iteratory, 637, 638, 997, 1004
 begin, 638
 const_iterator, 661
 dostęp swobodny, 661, 663,
 1000
 dwukierunkowe, 661, 663,
 1000
 end, 638
 istream_iterator, 694, 695
 istream_iterator<T>, 694
 iterator, 661
 kategorie, 662, 1000
 operacje, 638, 999
 ostream_iterator, 694
 ostream_iterator<T>, 694
 postępujące, 1000
 przechodzące do przodu,
 663
 string, 662
 strumienie, 694
 valarray, 662
 wbudowane tablice, 662
 wejściowe, 663, 1000
 wstawiające, 1016
 wyjściowe, 663, 1000
 ITU, 40

J

Java, 20, 739
 jawna konwersja typów, 547
 jednolity rozkład, 801

jednostki translacji, 69, 145
 język programowania, 34, 64,
 706, 852
 Ada, 729
 Algol, 724
 Algol60, 724
 Algol68, 727
 BCPL, 735
 C, 733, 735, 899
 C++, 20, 736
 cechy, 706
 COBOL, 161, 721
 do celów ogólnych, 724
 FLOW-MATIC, 722
 Fortran, 720
 Lisp, 721, 723
 Pascal, 727
 PERL, 759
 Simula, 715, 731
 Simula 67, 731
 Turbo Pascal, 728
 język angielski, 192
 JPEG, 438

K

K&R, 900
 kalkulator, 179, 180
 kanwa rysunku, 388
 karty kredytowe, 810
 kategorie iteratorów, 662
 Keep It Simple, Stupid, 714
 Kernighan Brian, 734, 900
 key_comp(), 1007
 key_compare, 1004
 key_type, 1004
 KISS, 714
 klasa pamięci, 950
 klasy, 184, 289, 731, 857, 972
 Axis, 392
 bazowe, 410, 458, 979
 bitset, 839
 Date, 312
 definicja, 289
 definiowanie funkcji
 składowych, 296
 definiowanie składowych,
 975
 derywacja, 458
 destruktor, 540, 542, 977
 dostęp do składowych, 184,
 292, 973

dziedziczenie, 979
 funkcje pomocnicze, 310
 funkcje składowe, 293
 funkcje wirtualne, 458, 980
 hermetyzacja, 458
 hierarchia, 464
 implementacja, 289
 inicjacja składowych, 979
 inicjatory składowych, 979
 interfejs, 289, 303
 konkretne, 450
 konstruktory, 185, 293, 976
 konstruktory domyślne, 306
 konstruktory jawne, 571
 kontrola dostępu, 451
 lista wartości inicjujących
 składowe, 185
 lokalne klasy, 260
 model dostępu do
 składowych, 463
 nadklasy, 458
 operacje generowane, 981
 parametryzowane, 605
 pochodne, 458, 979
 podklasy, 458
 predykaty składowych, 675
 przeciążanie operatorów, 302
 przesłanianie, 455, 461
 przyjaciele, 974
 raportowanie błędów, 299
 składowe, 289, 290, 972
 stałe funkcje składowe, 309
 szablony, 605, 983
 testowanie, 886
 this, 974
 tworzenie, 291
 typy argumentów, 304
 ukrywanie szczegółów, 294
 unie, 983
 vector, 128
 vector_ref, 1055
 Widget, 1053
 widoczność składowych, 290
 Window, 1054
 wirtualny destruktor, 543
 wywołanie funkcji
 składowej, 128
 zagnieżdżone klasy, 260
 klasy abstrakcyjne, 450, 981
 czyste funkcje wirtualne,
 464

- klasy graficzne, 407
 - Axis, 408, 479
 - Circle, 408, 430
 - Closed_polyline, 408, 419
 - Color, 414
 - Ellipse, 408, 431
 - figury geometryczne, 389
 - Function, 408, 476
 - Image, 408, 436
 - Line, 408, 410
 - Line_style, 408, 416
 - Lines, 408, 412
 - Mark, 408, 435
 - Marked_polyline, 408, 433
 - Marks, 408, 434
 - Open_polyline, 408, 418
 - Point, 408, 410
 - Polygon, 408, 420
 - Rectangle, 408, 422
 - Shape, 410, 448
 - Simple_window, 500
 - Text, 408, 428
- klasy składowe, 260
- klasy znaków, 768, 769, 1027
- klasyczna eliminacja Gaussa, 797
- klasyfikacja znaków, 368, 1024
- klawiatura, 1019
- klucze, 685
- kod, 67
- kod maszynowy, 67
- kod obiektowy, 67
- kod ogólny, 446
- kod pocztowy, 757
- kod wykonywalny, 67
- kod źródłowy, 44, 67
- kolejki, 1005
- kolekcje danych, 152, 630
- kolizja nazw, 257
- kolor tekstu, 399
- kolory, 414
- komentarze, 65, 162, 231, 904, 945
- kompilacja, 67, 161
 - warunkowa kompilacja, 931
- kompilator, 34, 67, 71, 909
- komputer, 39, 41, 46, 104
- komunikaty o błędach, 163
- konflikt nazw, 257
- koniec pliku, 88, 927
- konkatenacja łańcuchów, 85
- konkretyzacja szablonu, 604, 985
- konsola, 498
- konsolidacja, 161
 - błędy, 145
- konsolidacja pamięci, 821
- konsolidator, 70, 909, 971
- konstruktory, 185, 293, 294, 571, 976
 - debugowanie, 573
 - domyślny konstruktor, 254, 306, 308
 - jawne, 571
 - kopiujące, 456, 566
- kontenery, 152, 525, 526, 595, 596, 629, 634, 660, 661, 1001
 - adaptery, 1002
 - allocator_type, 1004
 - array, 660, 1001
 - asocjacyjne, 681, 1001, 1002
 - const_iterator, 1004
 - const_pointer, 1004
 - const_reference, 1004
 - const_reverse_iterator, 1004
 - deque, 660, 1001
 - destruktory, 1004
 - difference_type, 1003
 - dostęp do elementów, 1005
 - dziedziczenie, 607, 832
 - intruzyjne, 932
 - iterator, 1003
 - iteratory, 1004
 - key_compare, 1004
 - key_type, 1004
 - kolejki, 1005
 - konstruktory, 1004
 - list, 660, 1001
 - listy, 1006
 - map, 660, 681, 684, 1001
 - mapped_type, 1004
 - multimap, 660, 681, 1001
 - multiset, 660, 681, 1001
 - nieintruzyjne, 932
 - operacje kolejek, 1005
 - operacje kontenerów asocjacyjnych, 1007
 - operacje listowe, 1006
 - operacje stosowe, 1005
 - pointer, 1004
 - pojemność, 598, 1006
 - porównywanie, 1006
 - priority_queue, 1002
 - przypisania, 1004
 - queue, 1002
 - reference, 1004
 - reverse_iterator, 1004
 - rozmiar, 1006
 - sekwencyjne, 1001
 - set, 660, 681, 696, 1001
 - size_type, 1003
 - słowniki, 682
 - stack, 1002
 - stos, 1005
 - string, 1002
 - typy składowych, 1003
 - unordered_map, 660, 681, 689, 1001
 - unordered_multimap, 660, 681, 1001
 - unordered_multiset, 660, 681, 1002
 - unordered_set, 660, 681, 1002
 - unsorted_map, 690
 - valarray, 1002
 - value_type, 1003
 - vector, 660, 1001
 - wbudowana tablica, 1002
 - wektor, 526
 - zamienianie, 1006, 1007
 - zbiory, 691
- kontrola dostępu, 451
- kontrola typów, 94
- kontrolki, 505
- konwersje argumentów, 271
- konwersje typów, 111, 958, 961
 - arytmetyczne, 959
 - bezpieczne dla typów, 95
 - całkowitoliczbowe, 958
 - const_cast, 962
 - dynamic_cast, 961
 - niebezpieczne dla typów, 96
 - operatory, 961
 - między typami zmiennoprzecinkowymi i całkowitoliczbowymi, 959
 - referencje, 959
 - reinterpret_cast, 961
 - static_cast, 961
 - void*, 916
 - wartości logiczne, 959
 - wskaźniki, 959

zawężające, 96
 zdefiniowane przez
 użytkownika, 960
 zmiennoprzecinkowe, 959
 kopiowanie, 306, 456, 564, 693,
 976, 978
 głębokie, 569
 konstruktory kopiujące, 566
 płytkie, 569
 przez konstruowanie, 567
 przypisywanie z
 kopiowaniem, 567
 korzeń, 685
 krój pisma, 399
 kryterium porównywania, 866
 kursor, 65

L

label.set_color(), 393
 laptop, 41
 leak detector, 574
 left, 1023
 legal, 944
 length(), 369, 746, 1025
 length_error, 996
 less_equal<T>(), 1017
 less<T>(), 1017
 lexicographical order, 920
 lexicographical_compare(),
 1015
 liczby, 78, 93, 779
 bez znaku, 840
 binarne, 94, 947
 całkowite, 81, 780
 float, 782
 funkcje matematyczne, 802
 int, 782
 limity, 1028
 macierze, 784, 786
 mantysa, 782
 Matrix, 786
 obliczenia, 782
 ograniczenia typów
 liczbowych, 783
 ósemkowe, 355
 precyzja, 358, 780
 przekroczenie zakresu, 780,
 781
 rozmiar, 780
 rzeczywiste, 780

systemy liczbowe, 946
 szesnastkowe, 355, 947
 tablice, 784
 ujemne, 224
 wykładnik, 782
 ze znakiem, 840
 zmiennoprzecinkowe, 358,
 780, 781, 948
 zmiennoprzecinkowe
 podwójnej precyzji, 80
 liczby losowe, 800
 rand(), 801
 RAND_MAX, 801
 liczby zespolone, 803, 1030
 complex, 803
 część rzeczywista, 1031
 część urojona, 1031
 funkcje matematyczne, 803
 operacje, 1030
 operatory, 804
 sprzężenie, 1031
 typy, 1030
 limity typów liczbowych, 1028
 Line, 389, 408, 410, 411, 444
 Line_style, 389, 408, 416, 417
 Lines, 389, 408, 412, 452
 draw_lines(), 414
 Lines_window, 509, 510, 515
 linia, 410
 styl, 416
 Link, 551, 552, 557, 641
 insert(), 556
 linkage specification, 971
 linked list, 641
 linker, 70
 lint, 733, 906
 Lisp, 721, 723
 list, 642, 653, 660, 1001
 lista parametrów, 66, 123
 lista wartości inicjujących
 składowe, 185
 listy, 551, 596, 653
 add(), 553
 advance(), 553
 dwukierunkowe, 551, 932
 erase(), 553, 554
 find(), 553, 554
 insert(), 554
 jednokierunkowe, 551
 Link, 551, 552
 ogniwo, 551

operacje, 552
 zastosowanie, 554
 listy powiązane, 641
 dwustronnie, 641
 iteracja, 643
 iterator, 643
 jednostronnie, 641
 list, 642
 operacje, 642, 647
 usuwanie elementów, 647
 wstawianie elementów, 647
 literały, 946
 liczby całkowite, 946
 liczby
 zmiennoprzecinkowe, 948
 logiczne, 948
 łańcuchowe, 949
 wskaźnikowe, 949
 znakowe, 948
 locale, 376
 localtime(), 1039
 log(), 802, 1030
 log10(), 802, 1030
 logic programming, 714
 logical_and<T>(), 1017
 logical_not<T>(), 1017
 logical_or<T>(), 1017
 lokalne obiekty statyczne, 953
 Lola, 728
 long, 780, 965
 long double, 948
 long int, 835
 LONG_MAX, 1029
 LONG_MIN, 1029
 losowość, 800
 lower, 769, 1027
 lower_bound(), 699, 1007, 1012
 lub wykluczające, 956
 Lucent Bell Labs, 733, 735
 ludzie, 42
 l-wartość, 107

Ł

łagodne ograniczenia czasowe,
 814
 łańcuchy, 84, 130, 586, 746,
 949, 1024
 dopasowanie wyrażeń
 regularnych, 1025
 klasyfikacja znaków, 1024

- łańcuchy
 - konkatenacja, 85
 - małe litery, 1024
 - operacje, 746, 1025
 - przetwarzanie, 1024
 - strumienie, 365
 - styl języka C, 582, 919, 1036
 - wielkie litery, 1024
- łatwość utrzymania, 709
- łączenie, 69

- M**
- macierze, 56, 596, 715, 784, 786
 - błędy zakresu, 788
 - dostęp, 787
 - dwuwymiarowe, 792
 - FMA, 792
 - indeksowanie, 788
 - jednowymiarowe, 789
 - kopiowanie danych, 792
 - liczba elementów, 789
 - Matrix, 787
 - Matrix_error, 788
 - metoda eliminacji Gaussa, 797
 - mnożenie przez wektor, 800
 - operacje, 793
 - operatory przypisania, 791
 - rozwiązywanie równań
 - liniowych, 796
 - trójwymiarowe, 795
 - wejście, 794
 - wybór elementu
 - centralnego, 798
 - wycinki, 790
 - wyjście, 794
 - wymiary, 787
 - zamiana kolumn miejscami, 793
 - zamiana wierszy miejscami, 793
- macro substitution, 989
- Maddock John, 759
- magiczne stałe, 109
- Mail_file, 752, 755
- main(), 66, 78, 229, 945
 - argumenty, 945
- make_heap(), 1014
- make_pair(), 686, 687, 1018
- make_vec(), 620, 621

- makra, 615, 904, 927, 928, 989
 - definiowanie, 990
 - podobieństwo do funkcji, 929
 - zastosowanie, 928
- makra składniowe, 930
- malloc(), 903, 918, 1038
- manager pamięci wolnej, 539
- manipulatory, 355, 1022
- manipulowanie bitami, 844
- mantysa, 782
- map, 596, 660, 681, 682, 684, 822, 904, 1001
- mapped_type, 1004
- Mark, 400, 408, 435
- Marked_polyline, 408, 433
- Marks, 408, 434, 452
- maskowanie, 845
- matches.size(), 762
- matrix, 784
- Matrix, 56, 786, 787
- Matrix_error, 788
- MAX, 928
- max(), 258, 1015
- max_element(), 1015
- max_size(), 1006
- max_x(), 410
- max_y(), 410
- McCarthy John, 723
- McIlroy Doug, 734, 908
- mechanizm wyświetlający, 383
- mechanizm wywołań
 - wirtualnych, 543
- medycyna, 50
- mem_fun(mf), 1017
- mem_fun_ref(mf), 1017
- member, 289
- member access operator, 546
- member function, 293
- member initializer list, 185
- memchr(), 923, 1038
- memcmp(), 922, 1038
- memcpy(), 922, 1037
- memmove(), 922, 1037
- memory exhaustion, 560
- memset(), 923, 1038
- menu, 507, 513
 - tworzenie, 513
- Menu, 409, 507
- merge(), 668, 1013
- metoda eliminacji Gaussa, 797
- metody, 294
 - random_shuffle(), 1010
 - rotate_copy(), 1010
- metody nauczania, 32
- Microsoft Visual C++, 71
- Microsoft Visual Studio, 34
- miejsce na dane, 78
- miejsce rozpoczęcia programu, 66
- mierzenie czasu działania
 - funkcji, 1039
- mieszanie, 690
- min(), 1015
- min_element(), 1015
- minimalizm, 713
- minus<T>(), 1017
- mismatch(), 1008
- MIT McCarthy, 724
- mit samotnie pracującego
 - programisty, 43
- mktime(), 1039
- mnożenie macierzy przez
 - wektor, 800
- model dostępu do składowych, 463
- model graficzny, 383
- model iteratorów, 998
- model strumieni wejścia
 - i wyjścia, 323
- Modula, 728
- Modula-2, 728
- modular arithmetic, 842
- modulus<T>(), 1017
- modułowość, 712
- modyfikacja wyjściowych
 - liczb całkowitych, 356
- monitorowanie, 48
- monitorowanie podsystemów, 818
- monitory jakości linii
 - produkcyjnych, 810
- monitory medyczne, 810
- move(), 424, 506
- MSC++, 909
- multimap, 660, 681, 1001
- multi-paradigm programming, 717
- multiple inheritance, 979
- multiplies<T>(), 1017
- multiset, 660, 681, 1001

N

- nadklasy, 458
 - nadmiernie inteligentni programiści, 852
 - nagłówki, 66, 254
 - namespace, 261, 279, 988
 - narzędzia, 178
 - NASA, 52
 - natychmiastowe reagowanie, 812
 - nauczanie, 29
 - Naur Peter, 725
 - nawiasy, 108, 163
 - nazwy, 80, 90, 162, 164, 240, 446, 855, 988
 - makra, 928
 - predefiniowane, 242
 - przestrzeń nazw, 261
 - słowa kluczowe, 91
 - styl oddzielania wyrazów, 92
 - symboliczne, 718
 - wielkość liter, 91
 - zakres dostępności, 258
 - nazwy w pełni kwalifikowane, 280
 - negate<T>(), 1017
 - new, 533, 545, 612, 621, 815, 819, 820, 903, 952, 955, 961
 - Next, 499
 - next(), 503, 504
 - next_permutation(), 1015
 - niechlujny kod, 710
 - niedouczeni programiści, 852
 - nieformatowane wstawianie, 1021
 - niejawna konwersja typów, 958
 - niekompletne programy, 141
 - niekontrolowane konwersje, 825
 - nieoczekiwane argumenty, 141
 - nieodpowiednie standardy kodowania, 852
 - nieokreślony, 945
 - nieprawidłowe argumenty, 151
 - nieprawidłowe dane wejściowe, 154
 - nieprzerwana praca, 813
 - nieskończona rekurencja, 196
 - niespodziewane dane wejściowe, 142
 - niespodziewany stan, 142
 - niestandardowe separatory, 370
 - niezdefiniowane, 945
 - niezgodne, ale niewymagające diagnostyki, 945
 - niezmiennik, 295
 - noboolalpha, 1022
 - non-intrusive container, 932
 - norm(), 804
 - noshowbase, 356, 1022
 - noshowpoint, 1022
 - noshowpos, 1022
 - noskipws, 1022
 - NOT, 110
 - not conforming but not requiring diagnostic, 945
 - not_equal_to<T>(), 1016
 - not1(g), 1017
 - not2(g), 1017
 - notacja *this, 601
 - notacja Backusa-Naura, 721
 - nouppercase, 1023
 - nowoczesne języki programowania, 719
 - nth_element(), 1012
 - null pointer, 537
 - number_of_points(), 414
 - numeric_limits<T>, 1028
 - Nygaard Kristen, 731
- ## O
- Oberon, 728
 - Oberon-2, 728
 - obiektowość, 33
 - obiekty, 78, 92
 - auto_ptr, 621
 - automatyczne, 953
 - czas trwania, 952
 - destruktory, 540, 977
 - dostęp do składowych, 184
 - funkcje składowe, 293
 - inicjalizacja, 294
 - konstruktory, 185, 293, 976
 - konstruktory domyślne, 306
 - kopiowanie, 306, 456, 976, 978
 - kopiowanie głębokie, 569
 - kopiowanie płytkie, 569
 - lista wartości inicjujących składowe, 185
 - lokalne, 953
 - lokalne statyczne, 953
 - nazwy, 80
 - odwołanie do bieżącego obiektu, 298
 - poprawny stan, 295
 - przeźreń nazw, 953
 - przyjaciele, 974
 - referencje, 968
 - stan, 295
 - this, 555, 974
 - tworzenie, 961, 976
 - tymczasowe, 270, 953
 - usuwanie, 953, 976
 - w pamięci wolnej, 953
 - wskaźniki, 545
 - wykorzystanie obiektów bez nazw, 426
 - obiekty funkcyjne, 673, 674, 1016
 - object-oriented programming, 458
 - obliczanie wartości skrótu, 690
 - obliczenia, 104, 105, 159, 782, 1028
 - szacowanie, 159
 - obrazy, 399, 436
 - obsługa błędów, 151, 220
 - operacje wejścia i wyjścia, 330
 - strumienie, 1020
 - testowanie, 221
 - obsługa wyjątków, 151
 - oct, 355, 356, 357, 1023
 - oczyszczanie kodu, 227
 - funkcje, 229
 - komentarze, 231
 - stałe symboliczne, 227
 - układ kodu, 230
 - od dołu do góry, 32, 710
 - od góry do dołu, 33, 710
 - odczytywanie dane z pliku, 326, 328, 487
 - sekwencje znaków, 757
 - struktury wartości, 344
 - oddzielanie warstwy komunikacyjnej od funkcji, 337
 - odporność na błędy, 813
 - odtwarzacze MP3, 810

- odwołanie do bieżącego obiektu, 298
 - odzyskiwanie sprawności po wystąpieniu błędu, 233
 - off-by-one error, 153
 - ofstream, 326, 329
 - ogniwo, 551
 - ogólny kod, 446
 - ograniczenia, 615
 - ograniczenia czasowe, 814
 - ograniczenia typów liczbowych, 783
 - okno, 385, 390, 425, 500
 - okrąg, 430
 - opcjonalne sprawdzanie, 614, 615
 - open(), 327, 328, 1019, 1020
 - Open_polyline, 408, 418, 429, 452
 - draw_lines(), 434
 - operacje, 82, 84, 570
 - bajtowe, 922
 - bitowe, 835, 836
 - generowane, 981
 - liczby zespolone, 1030
 - łańcuchowe, 746, 1025
 - na iteratorach, 638, 999
 - na zbiorach danych, 634
 - tablicowe, 786
 - operandy, 107
 - operator, 303, 958
 - operator overloading, 302
 - operator[], 575, 614
 - operator=, 600
 - operatory, 82, 84, 109
 - !=, 110
 - %, 225
 - &, 529, 549
 - &&, 110
 - (), 674
 - *, 534
 - [], 534
 - ||, 110
 - ++, 89, 90, 109
 - <, 110
 - <<, 68, 110
 - <=, 110
 - =, 85
 - ==, 110
 - >, 110
 - >, 546, 973
 - >=, 110
 - >>, 79, 81, 110
 - addytywne, 955
 - adres, 529
 - aplikacji, 674
 - const_cast, 547
 - definiowanie przez użytkownika, 958
 - delete, 538, 545, 573, 953, 961
 - dereferencja, 530, 534
 - dostęp do składowych, 546
 - indeksowanie, 534
 - inkrementacja, 89
 - konwersja typów, 961
 - liczby zespolone, 804
 - logiczne, 960
 - new, 533, 545, 961
 - porównania, 956
 - przeciążanie, 302, 971
 - przesunięcie, 955
 - przypisanie, 85, 957
 - reinterpret_cast, 531, 547
 - relacyjne, 956
 - sizeof, 364, 531, 960
 - static_cast, 547
 - tryby otwierania, 1032
 - typedef, 646
 - wejściowe, 339
 - wielokrotność, 955
 - wyjście, 68
 - wyjściowe, 338
 - wyłuskanie, 530
 - wywołanie funkcji, 674
 - zasięg, 973
 - zawartość, 530
 - złożone operatory przypisania, 89
 - oprogramowanie, 40
 - or, 956
 - OR, 110
 - ostre ograniczenia czasowe, 814
 - ostream, 288, 323, 325, 354, 750, 1018, 1019
 - ostream_iterator, 694
 - ostream_iterator<>, 694
 - ostreamstringstream, 365, 366
 - otwieranie plików, 326, 361, 927, 1032
 - tryby otwierania, 361
 - otwieranie strumienia, 1019
 - Out_box, 409, 507
 - out_of_range, 153, 154, 615, 996
 - output iterator, 663
 - overflow_error, 996
 - overload resolution, 969
 - overriding, 455, 460
- ## P
- pair, 1017
 - pair(), 1013
 - pair<>, 686
 - paleta kolorów, 415
 - palindrom, 586
 - wykorzystanie łańcuchów, 586
 - wykorzystanie tablic, 587
 - wykorzystanie wskaźników, 588
 - pamięć, 342, 526, 529, 950, 951, 1037
 - automatyczna, 532, 818, 951
 - dynamiczna, 818
 - kod, 532
 - statyczna, 532, 818, 952
 - tekst, 532
 - pamięć wolna, 532, 819, 918, 952
 - alokacja obiektów, 533
 - dealokacja, 538
 - destruktory, 542
 - rezerwowanie pamięci, 533
 - paradygmaty programowania, 714, 715
 - parametric polymorphism, 605
 - parametrized class, 605
 - parametrized function, 605
 - parametrized type, 605
 - parametry, 123, 262
 - referencje, 549
 - szablony, 603
 - wskaźniki, 549
 - PARC, 708
 - parser, 194
 - parsowanie, 189
 - partial_sort(), 1012
 - partial_sort_copy(), 1012
 - partial_sum(), 677, 1031
 - partition(), 1013
 - Pascal, 726, 727, 728

- PC, 40
- PDA, 810
- pełna gwarancja, 621
- PERL, 739, 759
- permutacje, 1015
- Petersen „Pete” Lawrence, 38
- pętla oczekująca, 504
- pętle, 88, 119, 132, 879
 - for, 121
 - przepełnienie bufora, 880
 - treść, 120
 - while, 88, 119
 - zmienna iteracyjna, 119
 - zmienna kontrolna, 119
- PFN, 846
- PHP, 739, 759
- piksel, 388
- pisanie gramatyki, 193
- pisanie kodu, 44
- pisanie programu, 175, 176
 - biblioteki, 178
 - etapy rozwoju
 - oprogramowania, 177
 - gramatyka, 188
 - implementacja tokenów, 183
 - narzędzia, 178
 - obsługa błędów, 220
 - oczyszczanie kodu, 227
 - odzyskiwanie sprawności
 - po wystąpieniu błędu, 233
 - parsowanie, 189
 - problem, 176
 - prototyp, 179
 - przypadki użycia, 180
 - pseudokod, 180, 187
 - rozwiązanie problemu, 177
 - strategia, 177
 - struktura programu, 211
 - strumienie tokenów, 206
 - szczegóły techniczne, 248
 - testowanie, 221
 - tokeny, 182
 - używanie tokenów, 185
 - wczytywanie liczb, 210
 - wczytywanie tokenów, 209
 - wersja „1.0”, 224
 - wersja z ograniczoną
 - funkcjonalnością, 179
 - wypróbowywanie drugiej
 - wersji, 205
 - wypróbowywanie pierwszej
 - wersji, 201
- zamiana gramatyki
 - w kod, 194
 - zestaw zasad, 189
- PL/360, 728
- planowanie rejsów, 47
- pliki, 325, 1032
 - C, 927
 - cpp, 67
 - EOF, 927
 - FILE, 927
 - format, 325
 - h, 67
 - ifstream, 326, 329
 - koniec, 88
 - odczytywanie danych, 326, 328
 - ofstream, 326, 329
 - otwieranie, 326, 361, 927, 1032
 - pozycjonowanie, 361, 365
 - strukturalne, 341
 - strumienie, 326
 - tryby otwierania, 361
 - zamykanie, 327, 927, 1032
 - zapisywanie danych, 326, 328
- pliki binarne, 362
 - odczytywanie, 363
 - operacje wejścia i wyjścia, 363
 - otwieranie, 363
 - zapisywanie, 363
- pliki nagłówkowe, 66, 254
- plus<T>(), 1017
- pobieranie danych
 - wejściowych, 79
- poddopasowanie, 762
- podejście do pisania
 - poprawnego
 - oprogramowania, 710
 - od dołu, 710
 - od góry, 710
- podklasy, 458
- podstawowa gwarancja, 621
- podstawowe czynniki wyrażień, 200
- Point, 384, 389, 397, 408, 410, 444
- point(), 414
- pointer, 459, 528, 529, 1004
- pointer arithmetic, 579
- pointer semantics, 570
- pojemność kontenera, 598
- pojęcie abstrakcyjne, 464
- poła, 360
- poła bitowe, 846, 982
- polar(), 804, 1031
- polimorfizm, 606
 - ad hoc, 605
 - czasu działania programu, 458
 - parametryczny, 605
- Polygon, 385, 389, 394, 408, 420, 452
 - add(), 422
- pomyłka o jeden, 153
- pool, 823
- pop_back(), 662, 1005
- pop_front(), 1005
- pop_heap(), 1014
- popelnianie błędów, 141
- poprawność, 709, 813
- poprawny stan obiektu, 295
- porównania, 956
- porządek leksykograficzny, 920
- porządek wykonywania
 - instrukcji, 276
 - globalna inicjacja, 277
 - wartościowanie wyrażień, 277
- porządkowanie łańcuchów, 697
- POSIX, 764
- postdekrementacja, 109
- postinkrementacja, 109
- poważne błędy, 870
- powiększanie wektora, 127
- powtarzające się słowa, 87
- powtarzanie, 119
- poziom abstrakcji, 712
- pozycjonowanie, 361, 365
- prawa optymalizacji, 816
- prawdopodobne błędy, 870
- prawie kontenery, 1002
- pre-condition, 166
- precyzja liczb
 - zmiennoprzecinkowych, 358, 359
- precyzja obliczeń, 780
- predefiniowane nazwy, 242
- predekrementacja, 109
- predykaty, 1016
- predykaty składowych klas, 675

- preinkrementacja, 109
- preprocessor, 855, 989
 - dyrektywy, 989
- preprocessing, 255
- prev_permutation(), 1015
- prezentacja danych, 394
- Princeton University, 723, 735
- print, 769, 1027
- print(), 843, 906
- printf(), 376, 924, 925, 926, 1033
 - specyfikatory konwersji, 1033
- priority_queue, 1002
- private, 206, 289, 463, 973
- problem, 176
- problem stałych wektorów, 576
- problem zawiązujących konwersji, 98
- problemy z pamięcią wolną, 819
- problemy z wyjątkami, 619
- proces tworzenia programu, 55
- produkcje, 194
- profesjonalizm, 706
- program, 34, 63, 64, 944
 - cena, 54
 - dobrze zaprojektowany, 54
 - główny program, 883
 - kompilacja, 67
 - łączenie, 69
 - niezawodność, 54
 - pisanie, 176
 - poprawność, 54
 - proces tworzenia, 55
 - projekt, 1044
 - punkt startowy, 66
 - struktura, 211
 - szkielet, 229
 - tworzenie, 1044
 - uruchamianie, 945, 1044
 - utrzymanie, 54
 - Witaj, świecie, 64
 - zamykanie, 945
 - zgodny program, 944
- programista, 43
- programowanie, 19, 30, 34, 39, 40, 44, 55, 56, 64
 - aspektowe, 714
 - funkcyjne, 714, 721
 - grafika, 382
 - logiczne, 714
 - obiektywne, 458, 465, 606, 714, 715
 - ogólne, 605, 606, 670, 714, 715
 - ograniczeniowe, 714
 - proceduralne, 714, 715
 - regulowe, 714
 - systemy, 736
 - systemy wbudowane, 809
 - wieloparadygmatowe, 717
- projekt, 47, 1044
 - dodawanie pliku źródłowego C++, 1045
 - tworzenie, 1044
- projektowanie, 55, 177
- projektowanie klas, 444
 - abstrakcyjna klasa bazowa, 464
 - chronione składowe, 458
 - czyste funkcje wirtualne, 464
 - definiowanie funkcji wirtualnych, 461
 - dostęp do danych, 448
 - dostęp do składowych, 463
 - dziedziczenie implementacji, 465
 - dziedziczenie interfejsu, 465
 - hermetyzacja, 458
 - hierarchia klas, 464
 - klasy bazowe, 458
 - klasy graficzne, 384, 443
 - klasy pochodne, 458
 - kontrola dostępu, 451
 - kopiowanie, 456
 - nazewnictwo, 446
 - operacje, 445
 - prywatne składowe, 458
 - przesłanie, 455, 461
 - Shape, 448
 - tworzenie podklas, 461
 - typy, 444
 - układ obiektu, 459
 - zmienność, 448, 456
- projektowanie pod kątem testowania, 890
- promocje, 958
- promotion, 958
- prostokąty, 395, 422
- protected, 450, 463, 973, 979
- prototyp, 179
- prototyp funkcji, 907
- prywatne składowe, 458
- przechowywanie dane, 630
 - sekwencje znaków, 653
 - zmiennie, 240
- przechwytywanie wyjątków, 151
- przeciążanie funkcje, 903, 906
 - operatory, 302, 971
- przecinek, 957
- przedstawianie danych na wykresach, 486
- przekazywanie argumentów przez referencję, 267, 269
 - przez stałą referencję, 265
 - przez wartość, 264, 269
- przekroczenie zakresu liczby, 780, 781
- przeñośność kodu, 34
- przepelnienie bufora, 880
- przesłanie, 455, 460, 461
- przestrzeń nazw, 261, 279, 917, 988
 - fully qualified name, 280
 - nazwa w pełni kwalifikowana, 280
 - std, 995
 - using, 280
 - zakres, 951
- przestrzeń znaczników struktur, 912
- przesunięcie, 845
- przesunięcie bitowe, 836, 955
- przeszukiwanie binarne, 698
- przetwarzanie danych, 52, 630, 634
- przetwarzanie tekstu, 745, 1024
 - from_string(), 748
 - isalpha(), 758
 - isdigit(), 758
 - łańcuchy, 746
 - operacje na łańcuchach, 746
 - słowniki, 750
 - string, 746
 - stringstream, 747
 - strumienie wejścia i wyjścia, 750

to_string(), 748
 Unicode, 747
 wyrażenia regularne, 758, 759
 wyszukiwanie, 761
 przetwarzanie wstępne, 255
 przewidywalność, 814, 815
 awaria, 816
 błędy przejściowe, 817
 ideały, 815
 przyciski, 506
 Next, 499
 przyjaciele, 974
 przyjmowanie na wejściu
 liczb całkowitych, 356
 przypadki testowe, 168, 873
 przypadki użycia, 180
 przypisanie, 85, 107, 110, 957
 przypisanie kopiujące, 567, 568, 570
 przypisanie wektorów, 600
 pseudokod, 180, 187
 public, 206, 289, 463, 973
 pula, 822, 823
 punct, 769, 1027
 punkt, 384, 410
 punkt startowy programu, 66
 pure virtual function, 450
 push_back(), 127, 128, 526, 528, 595, 599, 655, 662, 1005, 1025
 push_front(), 644, 1005
 push_heap(), 1014
 pusta instrukcje, 113
 put_on_top(), 425
 putback(), 332
 putchar(), 1036
 putchar(), 1036

Q

qsort(), 1040
 queue, 1002

R

radius(), 430
 RAI, 586, 620, 622, 879, 987
 rand(), 801, 1040
 RAND_MAX, 801
 random_matrix(), 800
 random_shuffle(), 1010

random_vector(), 800, 801
 random-access iterator, 661, 663
 range error, 153
 Range_error, 616
 raport o błędzie, 869
 raport techniczny, 658
 raportowanie błędów, 149, 299, 869
 rbegin(), 1005
 read(), 364, 1021
 Reading, 344
 real(), 803, 804, 1031
 realloc(), 919, 1038
 Rectangle, 389, 396, 408, 422, 427, 452
 red-black tree, 681, 684
 reference, 1004
 reference semantics, 570
 reference_to(), 1052
 referencje, 265, 267, 270, 548, 903, 968
 dziedziczenie, 550
 parametry, 549
 przekazywanie argumentów, 267, 269
 zasady stosowania, 270
 regex, 762, 905
 regex_error, 996
 regex_match(), 771, 1026
 regex_search(), 762, 770, 1026
 regexps, 764
 regexs, 764
 regression testing, 868
 regular expressions, 758, 764
 regulatory pump, 810
 regulatory zapór wodnych, 810
 reinterpret_cast, 531, 547, 825, 903, 954, 961
 REINTERPRET_CAST, 916
 rejestry, 718
 rejestry urzędzenia, 846
 rekord aktywacji funkcji, 273
 rekurencja, 196, 275
 remove(), 1010
 remove_copy(), 1010
 remove_copy_if(), 1010
 rend(), 1005
 repetition, 119
 repetycja, 193
 replace(), 1010
 replace_copy(), 1010

replikacja, 817
 reprezentacja, 597
 reprezentacja danych
 w pamięci, 342
 reserve(), 598, 599, 658, 1006
 resetiosflags(), 1023
 resize(), 595, 596, 599, 1006
 Resource Acquisition Is
 Initialization, 586, 620, 879
 reszta z dzielenia, 225
 return, 64, 123, 126, 263, 963
 return(), 622
 reverse(), 1010
 reverse_copy(), 1010
 reverse_iterator, 1004
 rezerwacja pamięci, 533, 598
 RFC2822, 751
 right, 1023
 Ritchie Dennis, 733, 900
 roboty składające samochody, 810
 roboty spawające, 810
 root node, 685
 rotate(), 1010
 rotate_copy(), 1010
 rozgałęzianie, 880
 rozmiar znaków, 399
 rozpoznawanie funkcji
 przeciążonych, 969
 rozwiązywanie problemów, 35
 rozwiązywanie równań
 liniowych, 796
 rozwijanie stosu, 987
 rozwój oprogramowania, 177
 równania liniowe, 796
 równoległy program, 815
 rule-based programming, 714
 run-time dispatch, 458
 run-time polymorphism, 458
 runtime_error, 154, 155, 332
 r-wartość, 107
 rysowanie
 figury, 454
 wykresy funkcji, 394, 472
 rzutowanie, 546, 547, 904, 961
 const_cast, 547, 548
 nowy styl, 915
 reinterpret_cast, 547
 static_cast, 547
 styl języka C, 915
 styl szablonowy, 915

S

- s, 1027
- samochody, 810
- samokontrola, 817
- samoloty, 810
- samoprzypisanie, 601
- samotnie pracujący programista, 43
- scale_and_add(), 792
- scanf(), 376, 880, 925, 926
- scientific, 358, 359, 1023
- scope resolution operator, 973
- search(), 1008, 1009
- search_n(), 1009
- seekg(), 365
- seekp(), 365
- see-out, 65
- sekwencje, 193, 637
- sekwencje losowe, 874
- selekcja, 113
 - if, 113
 - switch, 115
- semantyka
 - referencyjna, 570
 - wartościowa, 570
 - wskaźnikowa, 570
- serach(), 698
- set, 660, 681, 691, 696, 905, 1001
- set_color(), 386, 393, 414, 418
- set_difference(), 1014
- set_fill_color(), 397, 423
- set_font(), 399
- set_font_size(), 399
- set_intersection(), 1014
- set_label(), 393
- set_major(), 432
- set_mask(), 437
- set_minor(), 432
- set_point(), 452
- set_radius(), 430
- set_style(), 397, 416, 417
- set_symmetric_difference(), 1014
- set_union(), 1014
- setbase(), 1023
- setf(), 851
- setfill(), 1023
- setiosflags(), 1023
- setprecision(), 1023
- setw(), 360, 1023
- Shape, 56, 389, 410, 444, 448, 449, 461, 715, 887
 - draw(), 454
 - draw_lines, 454
 - klasa abstrakcyjna, 450
 - kontrola dostępu, 451
 - rysowanie figur, 454
 - układ obiektu, 459
- short, 835, 965
- show(), 505
- showbase, 356, 1022
- showpoint, 1022
- showpos, 1022
- signed, 843, 965
- silnik, 47
- Simple_window, 385, 389, 390, 409, 500, 835
- Simula, 715, 731
- Simula 67, 726, 731
- sin(), 394, 802, 1029
- singly-linked list, 551, 641
- sinh(), 802, 1030
- size(), 130, 526, 565, 662, 746, 1006, 1025
- size_type, 646, 841, 1003
- sizeof, 364, 531, 580, 904, 954, 960, 976
- skalowanie, 786
- skalowanie danych, 489
- skanery medyczne, 810
- skipws, 1022
- składnia, 142
- składniki, 199
- składowe, 184, 289, 972
 - chronione, 463
 - dostęp do wartości, 292
 - funkcje pomocnicze, 310
 - inicjacja, 979
 - prywatne, 463
 - publiczne, 463
 - static const, 976
 - typy, 289
 - widoczność, 290
- składowe składowych, 290
- skrót, 690
- słabe projektowanie, 893
- słowa kluczowe, 66, 91, 950
 - break, 116, 118
 - C, 913
 - case, 116, 227
 - catch, 151, 156
 - class, 184, 289, 857
 - const, 109, 253, 927, 960
 - else, 113
 - enum, 300
 - explicit, 572
 - extern, 251
 - for, 121
 - friend, 975
 - if, 113
 - int, 66
 - namespace, 261, 279, 988
 - operator, 303
 - private, 463
 - protected, 979
 - return, 123, 126, 263
 - signed, 843
 - static, 502
 - struct, 290, 291, 912
 - switch, 115
 - template, 608, 983
 - throw, 151, 986
 - try, 151
 - unsigned, 837
 - using, 280, 988
 - virtual, 451, 456, 980, 981
 - void, 262
 - while, 119
- słowniki, 596, 667, 681, 682
 - binarne drzewa
 - zrównoważone, 684
 - map, 682, 684
 - nieuporządkowane, 691
 - przetwarzanie tekstu, 750
 - zastosowanie, 687
- słowo, 834
- smatch, 762
- solve_random_system(), 799
- sort(), 130, 132, 526, 668, 697, 711, 715, 905, 1012
- sort_heap(), 1014
- sortowanie, 130, 675, 697, 1012
- space, 769, 1027
- specialization, 604
- specjalizacja, 604, 985
- specjalne przypadki, 871
- specyfikacja, 141, 177
- sposoby przechowywania sekwencji znaków, 653
- spójność, 713

- sprawdzanie
 - argumenty, 149, 271
 - białe znaki, 374
 - przecięcia, 422
 - typy argumentów funkcji, 906
 - zakres, 613
- sprintf(), 1033, 1036
- sprzęt, 813
- sprzężenie, 804, 1031
- sqrt(), 802, 1029
- srand(), 801, 1040
- stable_partition(), 1013
- stable_sort(), 1012
- stack, 824, 1002
- Stack, 824
- stack storage, 532
- stack unwinding, 987
- stała, 108, 960
 - C, 927
 - deklaracja, 252
- stałe funkcje składowe, 309
- stałe magiczne, 109
- stałe symboliczne, 108, 227
- stałe wektory, 576
- stałe wyrażeń, 109
- stan, 104, 288, 295
- stan strumienia, 330
- standard template library, 634
- standardowa pętla wejściowa, 340
- standardowe funkcje
 - matematyczne, 1029
- standardowy strumień wejściowy, 79, 1019
- standardowy strumień wyjściowy, 65, 1019
- standardowy strumień wyjściowy błędów, 1019
- standardy kodowania, 851, 852, 859
 - funkcje, 857
 - klasy, 857
 - nazwy, 855
 - preprocesor, 855
 - systemy o krytycznym znaczeniu, 858
 - systemy o ostrych ograniczeniach czasowych, 858
 - ściśle zasady, 854
 - układ, 855
 - wyrażenia, 857
 - zalecenia, 854
 - zasady dobrego stylu kodowania, 852
 - zasady ogólne, 855
- Start Without Debugging, 1046
- stary styl definiowania, 908
- statement, 67
- static, 502, 952
- static storage, 532
- static_cast, 364, 547, 903, 954, 961
- statyczne składowe klas, 953
- statyczny system typów, 733
- std, 995
- std::bitset, 835
- std::cout, 280
- std::out_of_range, 615
- std::set, 835
- std::string, 280
- std::vector, 835
- std_lib_facilities.h, 1045
- stdin, 924
- stdio, 1036
- stdout, 924
- sterowniki urządzeń, 322
 - drukarki, 810
 - napędy dyskowe, 810
- sterta, 532, 818, 952, 1014
- STL, 525, 634, 721
- stopień abstrakcji, 712
- storage class, 950
- stos, 275, 532, 818, 822, 824, 1005
- stos rekordów aktywacji, 275
- str(), 366, 373
- str_to_double(), 366
- strategia testowania, 870
- strcat(), 920, 921, 1037
- strchr(), 920, 922, 1037
- strcmp(), 920, 921, 1037
- strcpy(), 920, 921, 923, 1037
- stream state, 330
- streambuf, 376
- string, 80, 82, 83, 93, 288, 367, 527, 564, 653, 662, 746, 765, 905, 1002
- stringstream, 747, 749, 750, 1019
- strlen(), 581, 583, 920, 1037
- strncat(), 920, 921, 1037
- strncmp(), 920, 921, 1037
- strncpy(), 920, 921, 1037
- strong guarantee, 621
- Stroustrup Bjarne, 37, 725, 736
- strpbrk(), 1037
- strrchr(), 1037
- strstr(), 920, 1037
- strtod(), 1037
- strtol(), 1037
- strtoul(), 1037
- struct, 290, 291, 344, 912, 971, 982
- struct tag, 912
- struktura kodu, 710
- struktura programu, 211
- struktury, 290, 291, 912
- struktury danych, 104, 551, 596
- strumienie, 321, 1018
 - bufor, 324
 - buforowanie, 324, 376
 - cerr, 1019
 - cin, 1019
 - cout, 65, 1019
 - definiowanie operatorów wejściowych, 339
 - definiowanie operatorów wyjściowych, 338
 - formatowane wstawianie, 1021
 - formatowanie, 1022
 - hierarchia strumieni, 1019
 - ifstream, 326
 - istream, 323, 324, 1019
 - istreamstream, 365
 - iteratory, 694
 - manipulatory, 1022
 - nieformatowane wstawianie, 1021
 - obsługa błędów, 1020
 - obsługa błędów wejścia i wyjścia, 330
 - ofstream, 326
 - operacje wejściowe, 1021
 - operacje wyjściowe, 1022
 - operatory wejściowe, 339
 - ostream, 323, 325, 1018, 1019
 - ostreamstream, 365
 - otwieranie, 1019
 - plikowe, 326

- strumienie
 - separatory, 370
 - stan, 330
 - standardowa pętla wejściowa, 340
 - standardowe manipulatory, 1022
 - streambuf, 376
 - stringstream, 747
 - tekst, 750
 - tryby, 1020
 - wczytywanie pliku strukturalnego, 341
 - wczytywanie pojedynczej wartości, 333
 - wejście, 79, 322, 1021
 - wyjście, 65, 322, 323, 1022
 - znak kończący, 341
 - strumienie łańcuchowe, 365
 - formatowanie danych wyjściowych, 366
 - odczytywanie danych, 366
 - strumienie tokenów, 206
 - styl linii, 395, 397, 416
 - style programowania, 714, 715
 - programowanie obiektowe, 606
 - subclass, 458
 - suma logiczna, 110
 - superclass, 458
 - swap(), 1007, 1011
 - swap_columns(), 794
 - swap_ranges(), 1011
 - swap_rows(), 793
 - switch, 115, 116, 880, 962
 - break, 116, 118
 - case, 116
 - etykiety, 116
 - sync_with_stdio(), 924, 1035
 - syntax analyzer, 189
 - system nazw, 242
 - system testing, 868
 - system(), 1040
 - systematyczne wyszukiwanie błędów, 867
 - systemy liczbowe, 355, 946
 - systemy nagłaśniające, 810
 - systemy o krytycznym znaczeniu, 858
 - systemy o ostrych ograniczeniach czasowych, 858
 - systemy wbudowane, 810
 - adresy, 825
 - alokacja pamięci, 819
 - Array_ref, 830
 - bezpośrednia konserwacja, 812
 - bitset, 839
 - dziedziczenie, 832
 - interfejsy, 826
 - klasa interfejsu, 829
 - kontenery, 832
 - łagodne ograniczenia czasowe, 814
 - manipulowanie bitami, 844
 - natychniastowe reagowanie, 812
 - niekontrolowane konwersje, 825
 - nieprzerwana praca, 813
 - niezawodność, 811
 - odporność na błędy, 813
 - ograniczenia czasowe, 814
 - operacje bitowe, 835
 - ostre ograniczenia czasowe, 814
 - pamięć dynamiczna, 819
 - pamięć statyczna, 818
 - pamięć stosowa, 818
 - poła bitowe, 846
 - poprawność, 813
 - przewidywalność, 814, 815
 - rejstry urządzenia, 846
 - sprzęt, 813
 - standardy kodowania, 851
 - tablice, 825
 - wskaźniki, 825
 - współbieżność, 815
 - zarządzanie pamięcią, 818
 - zasoby, 812
 - zasoby fizyczne, 825
 - źle działające interfejsy, 826
 - szablony, 602, 607, 983
 - algorytmy, 605
 - alokator, 612
 - argumenty, 610, 984
 - dedukcja argumentów, 610
 - dziedziczenie, 607
 - funkcje, 605
 - generowanie typów, 604
 - klasy, 605
 - kompilacja kodu, 607
 - konkretyzacja, 604, 985
 - liczby całkowite jako parametry, 608
 - parametry, 603
 - polimorfizm parametryczny, 605
 - programowanie ogólne, 605
 - specjalizacja, 604, 985
 - tworzenie specjalizacji, 985
 - typy jako parametry, 603
 - typy składowe, 985
 - uogólnianie wektora, 610
 - szacowanie, 159
 - szczegóły techniczne, 248
 - szkielet programu, 229
 - szkieletowy system telekomunikacyjny, 49
 - sztuczna inteligencja, 724
 - szyfr TEA, 848
 - szzyfrowanie, 847
- ## Ś
- ściśle zasady kodowania, 854
 - średnik, 69, 112
 - środowisko programistyczne, 70
- ## T
- tablica skrótów, 689
 - tablica symboli, 240
 - tablice, 563, 577, 587, 784, 825, 904, 967
 - dostęp do elementów, 578
 - dostosowywanie do biblioteki STL, 658
 - elementy, 577, 967
 - inicjowanie, 582
 - numeracja elementów, 577, 967
 - problemy ze wskaźnikami, 583
 - wskaźniki, 580
 - wskaźniki na elementy, 578
 - zakres, 967
 - tablice asocjacyjne, 681, 750
 - tablice mieszające, 660, 690
 - tablice skrótów, 681, 750
 - tablice wbudowane, 784
 - tablice wielowymiarowe, 784
 - styl języka C, 785

- tan(), 802, 1029
 - tanh(), 802, 1030
 - TB, 51
 - TEA, 719, 847
 - Technical Report, 658, 691
 - technika RAI, 622
 - technika wielokrotnego
 - wykorzystania, 712
 - techniki obliczeniowe, 780
 - techniki programowania, 34
 - techniki tworzenia obiektów, 526
 - Technische Hogeschool Eindhoven, 726
 - tekst, 130, 398, 428, 746
 - Unicode, 747
 - zestawy znaków, 747
 - tekst programu, 67
 - telefony, 48, 810
 - telekomunikacja, 48
 - template, 603, 608, 983
 - template argument, 157
 - template instantiation, 604, 985
 - terabajt, 51
 - term(), 482
 - test case, 168
 - testowanie, 55, 168, 221, 711, 799, 865, 867
 - algorytmy, 875
 - biała skrzynka, 868
 - czarna skrzynka, 868
 - dane wejściowe, 876
 - dane wyjściowe, 876
 - debugowanie, 891
 - dowody, 867
 - GUI, 883
 - jednostkowe, 868, 869, 886
 - kategorie testów, 872
 - klasy, 886
 - nie-algorytmy, 875
 - pętle, 879
 - poważne błędy, 870
 - prawdopodobne błędy, 870
 - program główny, 883
 - projektowanie pod kątem testowania, 890
 - przypadki testowe, 168, 873
 - raport o błędzie, 869
 - regresyjne, 868
 - rozgałęzianie, 880
 - sekwencje losowe, 874
 - specjalne przypadki, 871
 - strategia, 870
 - systemowe, 868, 882
 - wydajność, 891
 - wykrywanie przypadków łamania wymagań interfejsu, 889
 - zależności, 877
 - zarządzanie zasobami, 878
 - zasoby, 876
 - zastrzeżenie, 867
 - znajdowanie założeń, które się nie potwierdzają, 889
 - testy prawdopodobieństwa, 817
 - testy systemowe, 868, 882
 - TeX, 383
 - Text, 389, 408, 428
 - draw_lines(), 429
 - text storage, 532
 - Text_iterator, 650
 - THE, 726
 - this, 555, 974
 - throw, 151, 619, 621, 957, 986, 987
 - time(), 1039
 - time_t, 1038
 - Tiny Encryption Algorithm, 847
 - tm, 1038
 - to_string(), 748, 749
 - Token_stream, 207
 - tokenize, 182
 - tokeny, 182
 - implementacja, 183
 - stosowanie, 185
 - wczytywanie, 209
 - tolower(), 369, 1024
 - toupper(), 369, 1024
 - transform(), 1009
 - transformata
 - Burrowsa-Wheelera, 719
 - transient bug, 535
 - transmisja sygnałów, 53
 - transmutacja, 922
 - Transparency, 416
 - transport, 47
 - treść funkcji, 66, 123
 - Troff, 383
 - try, 151, 234, 962
 - tryby otwierania plików, 361, 1032
 - tryby strumieni, 1020
 - try-catch, 987
 - turbiny wiatrowe, 810
 - Turbo Pascal, 728
 - tworzenie
 - klasy, 291
 - menu, 513
 - obiekty, 961, 976
 - podklasy, 461
 - program, 55, 1044
 - projekt, 1044
 - type generator, 604
 - typedef, 646, 656, 904, 918
 - typeid, 954
 - typy argumentów, 304
 - typy danych, 78, 80, 81, 92
 - bool, 80
 - char, 80, 93
 - double, 80, 93
 - enum, 300
 - int, 78, 80, 92
 - klasy, 972
 - kontrola typów, 94
 - konwersje, 111
 - konwersje bezpieczne dla typów, 95
 - konwersje niebezpieczne dla typów, 96
 - rzutowanie, 546
 - string, 80, 93
 - tablice, 967
 - unie, 983
 - valarray, 1031
 - void*, 546
 - wbudowane, 965
 - wskaźniki, 965
 - wyliczenia, 972
 - zmiennoprzecinkowe, 965
 - typy parametryzowane, 605
 - typy składowe szablonów, 985
 - typy strumieniowe, 1019
 - typy wbudowane, 288
 - typy zdefiniowane przez użytkownika, 183, 288, 971
- ## U
- uczenie się, 29
 - UDT, 288
 - układ kodu, 163, 230, 855

układ obiektu, 459
 układ równań liniowych, 796
 ukrywanie danych, 716
 ukrywanie szczegółów, 294
 umiejętności programistyczne, 44
 undefined, 945
 ungetc(), 331
 ungetc(), 1036
 Unicode, 354, 747
 unie, 983
 uninitialized_copy(), 1011
 uninitialized_fill(), 1011
 union, 971
 unique(), 1009, 1010
 unique_copy(), 668, 693, 695, 1010
 unit testing, 868
 University of California, 728
 University of Cambridge
 Computer Laboratory, 718
 University of Texas, 726
 Unix, 733
 unordered_map, 660, 681, 689, 690, 738, 1001
 unordered_multimap, 660, 681, 1001
 unordered_multiset, 660, 681, 1002
 unordered_set, 660, 681, 1002
 unsetf(), 357
 unsigned, 837, 965
 unsigned int, 839
 unsorted_map, 690
 unspecified, 945
 uogólnianie kodu, 631
 uogólnianie wektora, 610, 645
 uogólnienie, 464
 upper, 769, 1027
 upper_bound(), 699, 1007, 1012
 uppercase, 1022
 uruchamianie programu, 945, 1044, 1046
 urządzenia, 104
 kuchenne, 810
 PDA, 810
 sterujące wirówkami, 810
 user-defined type, 183, 288
 using, 280, 988
 deklaracja, 280
 dyrektywa, 280, 281, 988

usuwanie, 653
 obiekty, 953, 976, 977
 powtarzające się słowa, 87
 usuwanie błędów, 71, 161
 usuwanie nieużytków, 539
 usuwanie z konsolidacją
 pamięci, 821
 utrzymanie, 54
 utrzymywanie porządku, 696

V

valarray, 662, 1002, 1031
 valid, 944
 value semantics, 570
 value_comp(), 1007
 value_type, 646, 659, 661, 1003
 vector, 30, 126, 127, 132, 288, 526, 527, 543, 564, 593, 601, 653, 660, 784, 822, 904, 1001
 capacity(), 598
 dostęp do elementów, 544
 elementy, 528
 kopiowanie, 564
 operator(), 576
 operator[], 575
 push_back(), 127, 528, 599
 reserve(), 598
 resize(), 596
 rozmiar, 528
 size(), 130
 Vector, 616
 Vector_ref, 409, 427, 1055
 virtual, 451, 456, 980, 981
 virtual function table, 459
 virtual pointer, 459
 virtual table, 459
 Visual Studio, 1044
 Add New Item, 1045
 Application Settings, 1045
 budowanie programu
 wykonywalnego, 1046
 Build Solution, 1046
 Debug, 1046
 dodawanie pliku
 źródłowego C++
 do projektu, 1045
 FLTK, 1049
 instalacja, 1044
 kreator aplikacji Win32,
 1045

New, 1044
 Start Without Debugging,
 1046
 std_lib_facilities.h, 1045
 tworzenie programu, 1044
 tworzenie projektu, 1044
 uruchamianie programu,
 1044, 1046
 Win32 Console Application,
 1045
 wprowadzanie kodu
 źródłowego, 1045
 zapisywanie programu, 1046
 void, 262
 void*, 546, 965
 vptr, 459
 vtbl, 459, 460

W

w, 1027
 wait(), 504, 512
 wait_for_button(), 386, 390, 393, 499
 warstwa oprogramowania, 382
 warstwowa technika budowy
 aplikacji, 755
 warstwy, 755
 wartości, 80, 92
 wartości brzegowe, 153
 wartościowanie wyrażeń, 277
 wartość mieszająca, 689
 wartość obiektu, 295
 wartość przybliżona funkcji
 wykładniczej, 481
 wartość skrótu, 689
 wartość zwrotna funkcji, 261
 warunki końcowe, 167
 warunki wstępne, 165, 167
 wczytywanie
 dane, 333
 liczby, 128, 210
 pliki strukturalne, 341
 pojedyncze wartości, 333
 tokeny, 209
 wejście, 66, 218, 322, 925, 1019, 1021
 przyjmowanie na liczb
 całkowitych, 356
 wejście macierzy, 794
 wejście-wyjście, 105

- wektor, 126, 153, 427, 526, 594, 653, 784
 - destruktory, 540
 - dostęp do elementów, 526, 544, 575
 - dostosowanie do biblioteki STL, 656
 - get(), 544
 - konstruktor kopiujący, 566
 - kopiowanie, 565
 - liczba elementów, 595
 - operacje, 570
 - pamięć, 526
 - powiększanie, 127, 526
 - problem stałych wektorów, 576
 - push_back(), 127, 526
 - rozmiar, 526
 - set(), 544
 - size(), 526
 - technika RAII, 622
 - techniki tworzenia obiektów, 526
 - tworzenie, 127
 - typ elementów, 127
 - uogólnianie, 610, 645
 - vector, 127, 527
 - wyciek pamięci, 540
 - zmiana rozmiaru, 595, 596
 - zmiennność, 594
- wersja „1.0”, 224
- węzeł, 685
- WG21, 759
- Wheeler David, 718, 847
- while, 88, 119, 234, 962
- white-box testing, 868
- wiadomości e-mail, 751
- wiązanie, 446
- Widget, 502, 505, 1053
- widgety, 1056
 - In_box, 507
 - menu, 507
 - Out_box, 507
 - przyciski, 506
- wielkość liter, 91
- wielokąty, 385, 394, 420
- wielokropek, 970
- wielokrotne dziedziczenie, 979
- wielokrotne wykorzystanie, 712
- wiersze, 649
- Wilkes Maurice, 718
- Win32 Console Application, 1045
- Window, 389, 409, 425, 444, 1054
 - put_on_top(), 425
- windy, 810
- Wirth Niklaus, 727
- wirtualny destruktor, 543
- Witaj, świcie, 64
- wizualizacja danych, 472
- właściwości języka, 132
- Word, 383
- wprowadzanie
 - dane wierszami, 367
 - kod źródłowy, 1045
 - nazwy, 240
- write(), 364
- ws, 1023
- wskaźnik wirtualny, 459
- wskaźnik zerowy, 584
- wskaźniki, 459, 528, 529, 588, 825, 904, 949, 965
 - arytmetyka, 579
 - auto_ptr, 621, 622
 - C, 923
 - dereferencja, 966
 - dostęp poprzez do obiektów, 534
 - dostęp poprzez usunięty wskaźnik, 585
 - dziedziczenie, 550
 - elementy tablicy, 578
 - indeksowanie, 966
 - inicjowanie, 584
 - na funkcje, 911
 - na obiekty, 545
 - operacje, 966
 - parametry, 549
 - problemy, 583
 - referencje, 548
 - tablice, 580
 - this, 555, 974
 - zakres, 535
 - zerowy wskaźnik, 537
 - zmiennie lokalne, 585
- współbieżność, 815
- współbieżny program, 815
- współrzędne, 388, 410
- wstawianie, 653, 1016
- wycieki pamięci, 538, 539, 540
- wyczerpanie pamięci, 560
- wydajność, 614, 615, 709, 891
 - kontrola czasu, 893
- wyjątki, 151, 613, 617, 903, 986
 - biblioteka standardowa C++, 996
 - błędy zakresu, 152
 - błędy zawężania zakresu, 156
 - catch, 151, 156
 - nieprawidłowe argumenty, 151
 - nieprawidłowe dane wejściowe, 154
 - obsługa, 151
 - out_of_range, 153, 154
 - przechwytywanie, 151
 - rozwijanie stosu, 987
 - runtime_error, 154, 155
 - throw, 151, 987
 - try, 151
 - zasoby, 617
 - zgłaszanie, 151, 617, 986
- wyjście, 66, 218, 322, 924, 1019, 1022
 - wysyłanie liczb zmiennoprzecinkowych, 357
- wyjście macierzy, 794
- wyjście ze źle działającego kodu, 817
- wykluczające lub, 836
- wykładnik, 782
- wykonywanie obliczeń, 104, 105, 634
- wykorzystanie obiektów bez nazw, 426
- wykres funkcji, 394
- wykresy, 472, 486
- wykrywacz wycieków, 574
- wykrywanie przypadków łamania wymagań interfejsu, 889
- wyliczenia, 300, 972
 - definicja, 300
 - treść, 300
 - zastosowanie, 301
- wyłuskanie, 530
- wymiar, 785, 787
- wynik działania funkcji, 123
- wyniki, 105
- wypełnianie kolorem, 397

- wypróbowywanie drugiej wersji, 205
 - wypróbowywanie pierwszej wersji, 201
 - wyrażanie myśli, 711
 - wyrażanie obliczeń, 105
 - wyrażenia, 107, 195, 196, 857, 953
 - jednoargumentowe, 954
 - logiczne, 960
 - przecinek, 957
 - przyrostkowe, 954
 - wyrażenia regularne, 758, 759, 764
 - alnum, 768
 - alternatywa, 767
 - bad_expression, 769
 - błędy, 769
 - dopasowywanie, 770, 771, 1025
 - grupowanie, 760, 767
 - klasy znaków, 768, 769, 1027
 - matches.size(), 762
 - notacje, 764
 - poddopasowanie, 762
 - powtórzenia, 766, 1027
 - przedziały znaków, 768
 - regex, 762
 - regex_match(), 771, 1026
 - regex_replace(), 1026
 - regex_search(), 762, 770, 1026
 - rodzaje znaków, 765
 - składnia, 764
 - skrótów klas znaków, 1027
 - smatch, 762
 - symbol wieloznaczny, 1026
 - wyszukiwanie, 761, 770
 - wzorce, 759
 - zbiory znaków, 768
 - znaki, 764
 - znaki specjalne, 764, 1026
 - wyrażenia stałe, 108, 960
 - wyrażenie warunkowe, 258
 - wysyłanie na wyjście liczb całkowitych, 355
 - wysyłanie na wyjście liczb zmiennoprzecinkowych, 357
 - wyszukiwanie, 653, 669, 671, 689, 690, 697, 1012
 - binarne, 698, 866
 - wzorce, 761
 - wyszukiwanie błędów, 867
 - wyświetlanie obiektów na ekranie, 386
 - wywołanie C z poziomu C++ , 909
 - wywołanie C++ z poziomu C, 909
 - wywołanie funkcji, 261, 272, 674, 968
 - funkcje przeciążone, 969
 - funkcje składowe, 128
 - funkcje wirtualne, 459
 - wywołanie zwrotne, 497, 1052
- X**
- xdigit, 769, 1027
 - Xerox Palo Alto Research, 708
 - XML, 383
 - xor, 956
 - xscale, 490
- Y**
- yscale, 490
- Z**
- zachęta, 78
 - zachłanność na funkcje, 199
 - Zachowaj prostotę, głupku, 714
 - zagnieżdżone bloki, 260
 - zajmowanie pamięci, 617
 - zajmowanie zasobów jest inicjacją, 619, 620
 - zakres, 256, 950, 951
 - globalny, 257, 258, 951
 - instrukcje, 257, 951
 - klasy, 257, 951
 - lokalny, 257, 951
 - przestrzeń nazw, 257, 951
 - zalecenia, 854
 - zależności, 877
 - zamiana dopasowanych ciągów, 1026
 - zamiana gramatyki w kod, 194
 - zamykanie
 - pliki, 327, 927, 1032
 - program, 945
 - zapisywanie
 - dane w pliku, 326, 328
 - sekwencje znaków, 757
 - zapobieganie wyciekiem pamięci, 539
 - zapobieganie wyciekiem zasobów, 817
 - zaprzyjaźnianie, 463
 - zarezerwowane identyfikatory, 950
 - zarządca zasobów, 617
 - zarządzanie, 47
 - zarządzanie pamięcią, 818
 - alokacja, 819
 - alternatywy dla ogólnej pamięci wolnej, 822
 - dealokacja, 819
 - fragmentacja, 819, 820
 - konsolidacja pamięci, 821
 - problemy z pamięcią wolną, 819
 - przewidywalność, 819
 - pula, 822, 823
 - stos, 822, 824
 - usuwanie nieużytków z konsolidacją pamięci, 821
 - zarządzanie zasobami, 541, 617, 878
 - zasada RAII, 586
 - zasady dobrego stylu kodowania, 852
 - zasady kolejności wykonywania działań, 108
 - zasoby, 538, 541, 542, 617, 818, 876
 - fizyczne, 825
 - wyjątki, 617
 - zastępowanie przez makro, 989
 - zawężające konwersje, 96, 98
 - zawężanie zakresu, 156
 - zbiory, 691
 - zastosowanie, 692
 - zbiór bitów, 838
 - zbiór linii, 412
 - zdefiniowany w implementacji, 944
 - zestaw wymagań, 177
 - zestawy znaków, 747

- zgłaszanie wyjątków, 151, 617, 986
- zgodność, 614
- zgodny, 944
- zintegrowane środowisko programistyczne, 34, 70
- ZIP, 1048
- zliczanie powtarzających się słów, 89
- złożone operatory przypisania, 89
- zmiana rozmiaru, 595, 596, 599
 - dodawanie przestrzeni dla nowych elementów, 598
 - ilość dostępnej wolnej przestrzeni, 598
 - przypisywanie, 600
 - push_back(), 599
 - reprezentacja, 597
 - rezerwacja dodatkowych miejsc, 598
- zmiana stylu programowania, 852
- zmiana wielkości znaków, 369
- zmienna iteracyjna, 119
- zmienna kontrolna, 119
- zmiennne, 78, 80, 92, 236
 - alokacja w pamięci, 531
 - definicja, 79, 80
 - deklaracja, 252
 - domyślna inicjacja, 254
 - globalne, 277
 - kontrola typów, 94
 - nazwy, 90
 - przypisanie wartości, 86
 - typ danych, 80
 - wartość początkowa, 80
- znacznik struktury, 912
- znajdowanie elementu
 - o największej wartości, 639
- znak kończący, 341
- znaki, 368, 948, 1024
 - ASCII, 354
 - nieterminalne, 194
 - nowy wiersz, 79
 - specjalne, 949
 - terminalne, 194
- zrównoważone drzewo uporządkowane, 660
- zwalnianie pamięci, 618, 918
- zwracanie pamięci do obszaru wolnego, 538
- zwracanie wartości z funkcji, 263

Ż

- źle działające interfejsy, 826
- źródła błędów, 141
- źródło danych, 322

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



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

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

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

Wykorzystaj wiedzę BJARNE STROUSTRUPA i pisz profesjonalne programy w C++!

Jeśli zależy Ci na tym, aby zdobyć rzetelną wiedzę i perfekcyjne umiejętności programowania z użyciem języka C++, powinieneś uczyć się od wybitnego eksperta i twórcy tego języka — Bjarne Stroustrupa, który zaprojektował i jako pierwszy zaimplementował C++. Podręcznik, który trzymasz w ręku, daje Ci szansę odkrycia wszelkich tajemnic tego języka, obszernie opisanego w międzynarodowym standardzie i obsługującego najważniejsze techniki programistyczne. C++ umożliwia pisanie wydajnego i eleganckiego kodu, a większość technik w nim stosowanych można przenieść do innych języków programowania.

Książka *Programowanie. Teoria i praktyka z wykorzystaniem C++* zawiera szczegółowy opis pojęć i technik programistycznych, a także samego języka C++, oraz przykłady kodu. Znajdziesz tu również omówienia zagadnień zaawansowanych, takich jak przetwarzanie tekstu i testowanie. Z tego podręcznika dowiesz się, na czym polega wywoływanie funkcji przeciążonych i dopasowywanie wyrażeń regularnych. Zobaczysz też, jaki powinien być standard kodowania. Poznasz sposoby projektowania klas graficznych i systemów wbudowanych, tajniki implementacji, wykorzystywania funkcji oraz indywidualizacji operacji wejścia i wyjścia. Korzystając z tego przewodnika, nauczysz się od samego mistrza pisać doskonale, wydajne i łatwe w utrzymaniu programy.

- ❖ *Techniki programistyczne*
- ❖ *Infrastruktura algorytmiczna*
- ❖ *Biblioteka standardowa C++*
- ❖ *Instrukcje sterujące i obsługa błędów*
- ❖ *Implementacja i wykorzystanie funkcji*
- ❖ *Kontrola typów*
- ❖ *Interfejsy klas*
- ❖ *Indywidualizacja operacji wejścia i wyjścia*
- ❖ *Projektowanie klas graficznych*
- ❖ *Wektory i pamięć wolna*
- ❖ *Kontenery i iteratory*
- ❖ *Programowanie systemów wbudowanych*
- ❖ *Makra*



Dr BJARNE STROUSTRUP

zaprojektował i jako pierwszy zaimplementował język C++.

Stroustrup kieruje katedrą informatyki w College of Engineering na Uniwersytecie

Texas A&M, jest członkiem amerykańskiej National Academy of Engineering oraz AT&T. Jest także założycielem i członkiem komisji standaryzacyjnej ISO w sprawach języka C++.

helion.pl
księgarnia
internetowa

Nr katalogowy: 14343



Księgarnia internetowa
<http://helion.pl>



Zamówienia telefoniczne:
0 801 339900



0 601 339900



Helion

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

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

sięgnij po WIĘCEJ



KOD KORZYŚCI

ISBN 978-83-246-7720-7



9 788324 677207

Cena: 129,00 zł