

Siddhartha Rao

Programowanie
w Twoim
zasięgu!

Wydanie VII

C++

Dla każdego

SAMS

Hellon



Tytuł oryginału: Sams Teach Yourself C++ in One Hour a Day (7th Edition)

Tłumaczenie: Robert Górczyński

ISBN: 978-83-246-8077-1

© Helion 2014.

All rights reserved.

Authorized translation from the English language edition: SAMS TEACH YOURSELF C++ IN ONE HOUR A DAY, Seventh Edition; ISBN 0672335670; by Siddharta Rao; published by Pearson Education, Inc, publishing as SAMS Publishing.

Copyright © 2012 by Pearson Education, Inc.

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

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

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.

Wydawnictwo HELION dołożyło wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie bierze jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Wydawnictwo HELION nie ponosi 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)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/cppit7.zip>

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/cppit7>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

Spis treści

O autorze	21
Wstęp	23
Część I. Podstawy	27
Lekcja 1. Zaczynamy	29
Krótka historia języka C++	30
Powiązanie z językiem C	30
Zalety języka C++	30
Ewolucja standardu C++	31
Kto używa programów utworzonych w C++?	31
Tworzenie aplikacji C++	32
Kroki prowadzące do wygenerowania pliku wykonywalnego	32
Analiza błędów i ich usuwanie	33
Zintegrowane środowiska programistyczne	33
Tworzenie pierwszej aplikacji w C++	34
Kompilacja i uruchomienie pierwszej aplikacji w C++	35
Błędy kompilacji	37
Co nowego w C++11?	37
Podsumowanie	38
Pytania i odpowiedzi	38
Warsztaty	39
Lekcja 2. Anatomia programu C++	41
Komponenty programu	42
Dyrektywa preprocesora #include	42
Część główna programu — funkcja main()	43
Wartość zwrotna	44
Koncepcja przestrzeni nazw	45
Komentarze w kodzie C++	47
Funkcje w C++	48

Podstawowe wejście za pomocą <code>std::cin</code> i wyjście za pomocą <code>std::cout</code>	51
Podsumowanie	53
Pytania i odpowiedzi	53
Warsztaty	54
Lekcja 3. Zmienne i stałe	55
Czym jest zmienna?	56
Ogólne omówienie pamięci i adresowania	56
Deklarowanie zmiennych uzyskujących dostęp i używających pamięci	56
Deklarowanie i inicjalizowanie wielu zmiennych tego samego typu	59
Zrozumienie zakresu zmiennej	59
Zmienne globalne	61
Typy zmiennych najczęściej używane w C++	63
Użycie typu <code>bool</code> do przechowywania wartości boolowskich	64
Użycie typu <code>char</code> do przechowywania znaków	65
Koncepcja liczb ze znakiem i bez znaku	65
Liczby całkowite ze znakiem, czyli typy <code>short</code> , <code>int</code> , <code>long</code> i <code>long long</code>	66
Liczby całkowite bez znaku, czyli typy <code>unsigned short</code> , <code>unsigned int</code> , <code>unsigned long</code> i <code>unsigned long long</code>	67
Typy zmiennoprzecinkowe <code>float</code> i <code>double</code>	67
Określanie wielkości zmiennej za pomocą <code>sizeof</code>	68
Użycie <code>typedef</code> do zastąpienia typu zmiennej	72
Czym jest stała?	72
Dosłowne stałe	73
Deklarowanie zmiennych jako stałych przy użyciu <code>const</code>	74
Deklarowanie stałych za pomocą <code>constexpr</code>	75
Stałe typu wyliczeniowego	76
Definiowanie stałych za pomocą dyrektywy <code>#define</code>	78
Nazwy zmiennych i stałych	79
Słowa kluczowe, których nie można używać jako nazw zmiennych lub stałych	80
Podsumowanie	80
Pytania i odpowiedzi	81
Warsztaty	84
Lekcja 4. Tablice i ciągi tekstowe	85
Czym jest tablica?	86
Kiedy trzeba użyć tablicy?	86
Deklarowanie i inicjalizacja tablic statycznych	87

Jak w tablicy przechowywane są dane?	88
Uzyskanie dostępu do danych przechowywanych w tablicy	90
Modyfikacja danych przechowywanych w tablicy	91
Tablice wielowymiarowe	94
Deklarowanie i inicjalizowanie tablic wielowymiarowych	95
Uzyskanie dostępu do elementów tablicy wielowymiarowej	95
Tablice dynamiczne	97
Ciągi tekstowe w stylu C	99
Ciągi tekstowe C++ — użycie klasy <code>std::string</code>	102
Podsumowanie	104
Pytania i odpowiedzi	105
Warsztaty	106
Lekcja 5. Wyrażenia, instrukcje i operatory	109
Polecenia	110
Polecenia złożone, czyli bloki	111
Użycie operatorów	111
Operator przypisania (=)	111
Zrozumienie l-wartości i r-wartości	112
Operatory dodawania (+), odejmowania (-), mnożenia (*), dzielenia (/) i reszty z dzielenia (%)	112
Operatory inkrementacji (++) i dekrementacji (--)	113
Operator postfiksowy czy prefiksowy?	114
Operatory równości (==) i nierówności (!=)	117
Operatory relacji	118
Operatory logiczne NOT, AND, OR i XOR	120
Użycie w C++ operatorów logicznych NOT (!), AND (&&) i OR ()	122
Bitowe operatory NOT (~), AND (&), OR () i XOR (^)	126
Operatory bitowego przesunięcia w prawo (>>) oraz w lewo (<<)	128
Złożone operatory przypisania	130
Użycie operatora <code>sizeof</code> w celu określenia ilości pamięci zajmowanej przez zmienną	132
Pierwszeństwo operatorów	133
Podsumowanie	136
Pytania i odpowiedzi	136
Warsztaty	137

Lekcja 6. Sterowanie przebiegiem działania programu	139
Wykonanie warunkowe za pomocą if-else	140
Programowanie warunkowe z użyciem if-else	141
Warunkowe wykonanie wielu poleceń	143
Zagnieżdżone polecenia if	145
Przetwarzanie warunkowe za pomocą switch-case	149
Wykonywanie warunkowe przy użyciu operatora ?:	153
Wykonywanie kodu w pętlach	154
Bardzo prosta pętla wykonywana przy użyciu polecenia goto	155
Pętla while	157
Pętla do...while	159
Pętla for	161
Zmiana zachowania pętli za pomocą poleceń continue i break	165
Pętle działające w nieskończoność	165
Kontrolowanie pętli działającej w nieskończoność	166
Programowanie zagnieżdżonych pętli	169
Użycie zagnieżdżonych pętli do iteracji tablic wielowymiarowych	171
Użycie zagnieżdżonych pętli do obliczenia liczb ciągu Fibonacciego	173
Podsumowanie	174
Pytania i odpowiedzi	175
Warsztaty	176
Lekcja 7. Funkcje	179
Kiedy należy stosować funkcje?	180
Czym jest prototyp funkcji?	181
Czym jest definicja funkcji?	182
Czym jest wywołanie funkcji i argumenty?	182
Tworzenie funkcji z wieloma parametrami	183
Tworzenie funkcji bez parametrów i bez wartości zwrotnej	185
Parametry funkcji wraz z wartościami domyślnymi	186
Rekurencja, czyli funkcja wywołująca samą siebie	188
Funkcje z wieloma poleceniami return	190
Użycie funkcji do pracy z różnymi formami danych	191
Przeciążanie funkcji	192
Przekazanie funkcji tablicy wartości	194
Przekazywanie argumentów przez referencję	195
Jak wywołania funkcji są obsługiwane przez mikroprocesor?	197
Funkcje typu inline	198
Funkcja lambda	200

Podsumowanie	202
Pytania i odpowiedzi	203
Warsztaty	204
Lekcja 8. Wskaźniki i referencje	207
Czym jest wskaźnik?	208
Deklaracja wskaźnika	208
Określenie adresu zmiennej przy użyciu operatora referencji (&)	209
Użycie wskaźników do przechowywania adresów	210
Uzyskanie dostępu do danych przy użyciu operatora dereferencji (*)	213
Ile pamięci zabiera wskaźnik?	216
Dynamiczna alokacja pamięci	218
Użycie operatorów new i delete w celu dynamicznej alokacji i zwalniania pamięci	218
Efektywne użycie operatorów inkrementacji (++) i dekrementacji (--) na wskaźnikach	222
Użycie słowa kluczowego const ze wskaźnikami	225
Przekazywanie wskaźników funkcjom	227
Podobieństwa pomiędzy tablicami i wskaźnikami	228
Najczęstsze błędy programistyczne popełniane podczas używania wskaźników	231
Wycieki pamięci	232
Kiedy wskaźnik nie prowadzi do poprawnego adresu w pamięci?	232
Zawieszono wskaźniki (nazywane również zabłąkanymi)	234
Najlepsze praktyki podczas pracy ze wskaźnikami	234
Sprawdzenie, czy żądanie alokacji zakończyło się powodzeniem	236
Czym jest referencja?	240
Dlaczego referencje są użyteczne?	241
Użycie słowa kluczowego const w referencjach	243
Przekazywanie funkcji argumentów przez referencję	243
Podsumowanie	245
Pytania i odpowiedzi	245
Warsztaty	247
Część II. Podstawy programowania zorientowanego obiektowo w C++	249
Lekcja 9. Klasy i obiekty	251
Koncepcja klas i obiektów	252
Deklarowanie klasy	252
Tworzenie obiektu klasy	253

Uzyskanie dostępu do elementów składowych przy użyciu operatora kropki	254
Uzyskanie dostępu do elementów składowych przy użyciu operatora wskaźnika	255
Słowa kluczowe public i private	257
Abstrakcja danych dzięki słowu kluczowemu private	259
Konstruktory	261
Deklarowanie i implementowanie konstruktora	261
Kiedy i jak używać konstruktorów?	262
Przeciążanie konstruktorów	264
Klasa bez konstruktora domyślnego	267
Parametry konstruktora wraz z wartościami domyślnymi	269
Konstruktory wraz z listami inicjalizacyjnymi	270
Destruktor	272
Deklarowanie i implementowanie destruktora	272
Kiedy i jak używać destruktora?	273
Konstruktor kopiujący	276
Kopiowanie płytkie i związane z tym problemy	276
Wykonanie głębokiej kopii przy użyciu konstruktora kopiującego	279
Konstruktory przenoszące pomagają w poprawieniu wydajności	284
Różne sposoby użycia konstruktorów i destruktora	286
Klasa, której nie można kopiować	286
Klasa typu Singleton, która pozwala na istnienie tylko jednego egzemplarza	287
Klasy, których egzemplarze nie mogą być tworzone na stosie	290
Wskaźnik this	292
Operator sizeof() dla klasy	293
Jaka jest różnica pomiędzy strukturą i klasą?	297
Deklaracja „przyjaciela” klasy	297
Podsumowanie	300
Pytania i odpowiedzi	300
Warsztaty	302
Lekcja 10. Dziedziczenie	305
Podstawy dziedziczenia	306
Dziedziczenie i pochodzenie	306
Stosowana w języku C++ składnia pochodzenia	308
Specyfikator dostępu protected	311
Inicjalizacja klasy bazowej	
— przekazywanie parametrów klasie bazowej	314

Klasy potomne nadpisują metody klasy bazowej	316
Wywoływanie nadpisanych metod klasy bazowej	319
Wywoływanie metod klasy bazowej z poziomu klas potomnych	319
Klasy potomne ukrywają metody klasy bazowej	321
Kolejność użycia konstruktorów	324
Kolejność użycia destruktorów	324
Dziedziczenie prywatne	327
Dziedziczenie chronione	330
Problem segmentowania	333
Dziedziczenie wielokrotne	334
Podsumowanie	337
Pytania i odpowiedzi	338
Warsztaty	339
Lekcja 11. Polimorfizm	341
Podstawy polimorfizmu	342
Potrzeba stosowania polimorfizmu	342
Zachowanie polimorficzne implementowane przy użyciu funkcji wirtualnych	344
Konieczność stosowania wirtualnych destruktorów	346
Jak działa funkcja wirtualna? Zrozumienie tabeli funkcji wirtualnych	351
Abstrakcyjne klasy bazowe i funkcje czysto wirtualne	355
Użycie dziedziczenia prywatnego do rozwiązania problemu niejednoznaczności semantycznej	358
Wirtualne konstruktory kopiujące?	363
Podsumowanie	367
Pytania i odpowiedzi	367
Warsztaty	368
Lekcja 12. Typy operatorów i ich przeciążanie	371
Czym są operatory w C++?	372
Operatory jednoargumentowe	373
Typy operatorów jednoargumentowych	373
Programowanie jednoargumentowego operatora inkrementacji i dekrementacji	374
Programowanie operatorów konwersji	378
Programowanie operatora dereferencji (*) i operatora wyboru elementu składowego (->)	380
Operatory dwuargumentowe	384
Typy operatorów dwuargumentowych	385
Programowanie operatorów dodawania (a+b) i odejmowania (a-b)	386

Implementowanie operatorów dodawania/przypisania (+=) i odejmowania/przypisania (-=)	389
Przeciążanie operatorów równości (==) i nierówności (!=)	391
Przeciążanie operatorów <, >, <= i >=	394
Przeciążanie kopiującego operatora przypisania (=)	397
Operatory indeksowania	400
Funkcja operator()	404
Użycie konstruktora przenoszącego i przenoszącego operatora przypisania w wysokowydajnym programowaniu	406
Operatory, których nie można ponownie zdefiniować	413
Podsumowanie	414
Pytania i odpowiedzi	414
Warsztaty	415
Lekcja 13. Operatory rzutowania	417
Kiedy trzeba skorzystać z rzutowania?	418
Dlaczego rzutowanie w stylu C nie jest popularne wśród niektórych programistów C++?	419
Operatory rzutowania C++	419
Użycie static_cast	420
Użycie dynamic_cast i identyfikacja typu w czasie działania	421
Użycie reinterpret_cast	425
Użycie const_cast	426
Problemy z operatorami rzutowania C++	427
Podsumowanie	429
Pytania i odpowiedzi	429
Warsztaty	430
Lekcja 14. Wprowadzenie do makr i wzorców	433
Preprocesor i kompilator	434
Użycie dyrektywy #define do definiowania stałych	434
Użycie makr do ochrony przed wielokrotnym dołączaniem	437
Użycie dyrektywy #define do definiowania funkcji	438
Po co te wszystkie nawiasy?	440
Użycie makra assert do sprawdzania wyrażeń	441
Wady i zalety użycia funkcji makro	443
Wprowadzenie do wzorców	444
Składnia deklaracji wzorca	445
Różne rodzaje deklaracji wzorca	446
Funkcje wzorca	446

Wzorce i bezpieczeństwo typów	448
Klasy wzorca	449
Ustanawianie i specjalizacja wzorca	450
Deklarowanie wzorców z wieloma parametrami	451
Deklarowanie wzorców z parametrami domyślnymi	452
Przykład wzorca	452
Klasy wzorców i statyczne elementy składowe	454
Użycie <code>static_assert</code> do przeprowadzania operacji sprawdzania w trakcie kompilacji	456
Użycie wzorców w praktycznym programowaniu C++	457
Podsumowanie	458
Pytania i odpowiedzi	458
Warsztaty	459

Część III. Poznajemy standardową bibliotekę wzorców (STL) 461

Lekcja 15. Wprowadzenie do standardowej biblioteki wzorców 463	463
Kontenery STL	464
Kontenery sekwencyjne	464
Kontenery asocjacyjne	465
Wybór odpowiedniego kontenera	466
Adaptery	469
Iteratory STL	470
Algorytmy STL	471
Oddziaływania między kontenerami i algorytmami za pomocą iteratorów	472
Użycie słowa kluczowego <code>auto</code> pozwalającego kompilatorowi na definicję typu	474
Klasy STL string	475
Podsumowanie	475
Pytania i odpowiedzi	475
Warsztaty	476
Lekcja 16. Klasa string w STL 479	479
Dlaczego potrzebna jest klasa służąca do manipulowania ciągami tekstowymi?	480
Praca z klasą STL string	481
Ustanawianie obiektu STL string i tworzenie kopii	482
Uzyskanie dostępu do obiektu string i jego zawartości	484
Łączenie ciągów tekstowych	487

Wyszukiwanie znaku bądź podciągu tekstowego w obiekcie string	488
Skracanie obiektu STL string	490
Uproszczenie deklaracji iteratora przy użyciu słowa kluczowego auto	493
Odwracanie zawartości ciągu tekstowego	493
Konwersja wielkości znaków obiektu string	494
Implementacja klasy STL string oparta na wzorcach	496
Podsumowanie	496
Pytania i odpowiedzi	497
Warsztaty	497
Lekcja 17. Dynamiczne klasy tablic w STL	499
Charakterystyka klasy std::vector	500
Typowe operacje klasy vector	500
Ustanawianie klasy vector	500
Wstawianie elementów na końcu przy użyciu push_back()	503
Listy inicjalizacyjne	504
Wstawianie elementów w określonym położeniu przy użyciu metody insert()	504
Uzyskanie dostępu do elementów obiektu vector przy użyciu semantyki tablicy	508
Uzyskanie dostępu do elementów obiektu vector przy użyciu semantyki wskaźnika	510
Usuwanie elementów z obiektu vector	511
Zrozumienie koncepcji wielkości i pojemności	513
Klasa STL deque	515
Podsumowanie	518
Pytania i odpowiedzi	519
Warsztaty	520
Lekcja 18. Klasy STL list i forward_list	523
Charakterystyka klasy std::list	524
Podstawowe operacje klasy list	524
Ustanawianie obiektu std::list	524
Wstawianie elementów na początku obiektu list	526
Wstawianie elementów w środku obiektu list	529
Usuwanie elementów w obiekcie list	531
Odwrócenie i sortowanie elementów w obiekcie list	533
Odwracanie elementów	534
Sortowanie elementów	535
Sortowanie i usuwanie elementów listy zawierających obiekty danej klasy	538

Podsumowanie	544
Pytania i odpowiedzi	545
Warsztaty	545
Lekcja 19. Klasy STL set	547
Wprowadzenie do klas STL set	548
Podstawowe operacje klas STL set i multiset	549
Ustanawianie obiektu std::set	549
Wstawianie elementów do obiektów set lub multiset	552
Wyszukiwanie elementów w obiektach STL set lub multiset	554
Usuwanie elementów z obiektów STL set lub multiset	556
Wady i zalety używania obiektów STL set i multiset	562
Podsumowanie	566
Pytania i odpowiedzi	566
Warsztaty	567
Lekcja 20. Klasy STL map	569
Krótkie wprowadzenie do klas STL map	570
Podstawowe operacje klas STL map i multimap	571
Ustanawianie obiektu std::map lub std::multimap	571
Wstawianie elementów do obiektów STL map lub multimap	573
Wyszukiwanie elementów w obiekcie STL map	577
Wyszukiwanie elementów w obiekcie STL multimap	579
Usuwanie elementów z obiektów STL map lub multimap	580
Dostarczanie własnego predykatu sortowania	583
Jak działa tabela hash?	588
Używanie tabel hash w C++11: unordered_map i unordered_multimap	588
Podsumowanie	593
Pytania i odpowiedzi	593
Warsztaty	594
Część IV. Jeszcze więcej STL	597
Lekcja 21. Zrozumienie obiektów funkcji	599
Koncepcja obiektów funkcji i predykatów	600
Typowe aplikacje obiektów funkcji	600
Funkcje jednoargumentowe	600
Predykat jednoargumentowy	606
Funkcje dwuargumentowe	608
Predykat dwuargumentowy	611

Podsumowanie	614
Pytania i odpowiedzi	614
Warsztaty	615
Lekcja 22. Wyrażenia lambda w C++ 11	617
Czym są wyrażenia lambda?	618
W jaki sposób zdefiniować wyrażenie lambda?	619
Wyrażenie lambda dla funkcji jednoargumentowej	619
Wyrażenie lambda dla predykatu jednoargumentowego	621
Wyrażenie lambda wraz ze stanem przy użyciu listy przechwytywania [...]	622
Ogólna składnia wyrażeń lambda	624
Wyrażenie lambda dla funkcji dwuargumentowej	626
Wyrażenie lambda dla predykatu dwuargumentowego	628
Podsumowanie	631
Pytania i odpowiedzi	632
Warsztaty	632
Lekcja 23. Algorytmy STL	635
Co to są algorytmy STL?	636
Klasyfikacja algorytmów STL	636
Algorytmy niezmiennie	636
Algorytmy zmienne	638
Używanie algorytmów STL	640
Znajdowanie elementów o podanej wartości lub warunku	640
Zliczanie elementów o podanej wartości lub warunku	643
Wyszukiwanie elementu lub zakresu w kolekcji	645
Inicjalizacja w kontenerze elementów wraz z określonymi wartościami	648
Użycie <code>std::generate()</code> do inicjalizacji elementów wartościami wygenerowanymi w trakcie działania programu	650
Przetwarzanie elementów w zakresie za pomocą <code>for_each</code>	652
Przeprowadzanie transformacji zakresu za pomocą <code>std::transform</code>	654
Operacje kopiowania i usuwania	657
Zastępowanie wartości oraz zastępowanie elementu na podstawie danego warunku	661
Sortowanie i przeszukiwanie posortowanej kolekcji oraz usuwanie duplikatów	663
Partycjonowanie zakresu	666
Wstawianie elementów do posortowanej kolekcji	669

Podsumowanie	672
Pytania i odpowiedzi	672
Warsztaty	673
Lekcja 24. Kontenery adaptacyjne: stack i queue	675
Cechy charakterystyczne zachowania stosów i kolejek	676
Stosy	676
Kolejki	676
Używanie klasy STL stack	677
Ustanawianie obiektu stack	677
Funkcje składowe klasy stack	679
Wstawianie i usuwanie elementów z góry stosu przy użyciu metod push() i pop()	679
Używanie klasy STL queue	681
Ustanawianie obiektu queue	682
Funkcje składowe klasy queue	683
Wstawianie na końcu i usuwanie na początku kolejki przy użyciu metod push() i pop()	684
Używanie klasy STL priority_queue	686
Ustanawianie obiektu priority_queue	686
Funkcje składowe klasy priority_queue	688
Wstawianie na końcu i usuwanie na początku kolejki priorytetowej przy użyciu metod push() i pop()	689
Podsumowanie	692
Pytania i odpowiedzi	692
Warsztaty	693
Lekcja 25. Praca z opcjami bitowymi za pomocą STL	695
Klasa bitset	696
Ustanowienie klasy std::bitset	696
Używanie klasy std::bitset i jej elementów składowych	698
Operatory std::bitset	698
Metody składowe klasy std::bitset	699
Klasa vector<bool>	702
Ustanowienie klasy vector<bool>	702
Używanie klasy vector<bool>	703
Podsumowanie	704
Pytania i odpowiedzi	705
Warsztaty	705

Część V. Zaawansowane koncepcje C++	707
Lekcja 26. Sprytnie wskaźniki	709
Czym są sprytnie wskaźniki?	710
Na czym polega problem związany z używaniem wskaźników konwencjonalnych?	710
W jaki sposób sprytnie wskaźniki mogą pomóc?	711
W jaki sposób są implementowane sprytnie wskaźniki?	711
Typy sprytnych wskaźników	713
Kopiowanie głębokie	714
Mechanizm kopiowania przy zapisie (COW)	716
Sprytnie wskaźniki zliczania odniesień	716
Sprytnie wskaźniki powiązane z licznikiem odniesień	717
Kopiowanie destrukcyjne	718
Używanie klasy <code>std::unique_ptr</code>	720
Popularne biblioteki sprytnych wskaźników	722
Podsumowanie	723
Pytania i odpowiedzi	723
Warsztaty	724
Lekcja 27. Użycie strumieni w operacjach wejścia-wyjścia	727
Koncepcja strumieni	728
Ważne klasy i obiekty strumieni C++	729
Użycie <code>std::cout</code> do zapisu w konsoli sformatowanych danych	731
Użycie <code>std::cout</code> do zmiany formatu wyświetlanych liczb	731
Wyrównanie tekstu i ustawienie szerokości pola przy użyciu <code>std::cout</code>	734
Użycie <code>std::cin</code> dla danych wejściowych	735
Użycie <code>std::cin</code> w celu umieszczenia danych wejściowych w zmiennych typu POD	735
Użycie <code>std::cin::get</code> w celu umieszczenia danych wejściowych w buforze typu <code>char</code> w stylu C	736
Użycie <code>std::cin</code> w celu umieszczenia danych wejściowych w <code>std::string</code>	738
Użycie <code>std::fstream</code> do obsługi pliku	739
Otwieranie i zamykanie pliku przy użyciu metod <code>open()</code> i <code>close()</code>	740
Tworzenie i zapisywanie tekstu w pliku przy użyciu metody <code>open()</code> i operatora <code><<</code>	741
Odczyt tekstu z pliku przy użyciu metody <code>open()</code> i operatora <code>>></code>	742
Zapis i odczyt pliku binarnego	744
Użycie <code>std::stringstream</code> do konwersji ciągu tekstowego	746

Podsumowanie	748
Pytania i odpowiedzi	748
Warsztaty	749
Quiz	749
Ćwiczenia	749
Lekcja 28. Obsługa wyjątków	751
Czym jest wyjątek?	752
Co powoduje zgłoszenie wyjątku?	753
Implementacja bezpieczeństwa wyjątków przy użyciu try i catch	753
Użycie catch(...) do obsługi wszystkich wyjątków	753
Przechwytywanie wyjątku określonego typu	755
Użycie throw do zgłoszenia wyjątku określonego typu	757
Jak działa obsługa wyjątków?	758
Wyjątki klasy std::exception	761
Własna klasa wyjątku wywodząca się z std::exception	762
Podsumowanie	765
Pytania i odpowiedzi	765
Warsztaty	766
Lekcja 29. Co dalej?	769
Czym wyróżniają się obecnie stosowane procesory?	770
Jak lepiej używać wielu rdzeni?	771
Czym jest wątek?	772
Dlaczego należy tworzyć aplikacje wielowątkowe?	772
Jak wątki wymieniają dane?	773
Użycie muteksu i semaforów do synchronizacji wątków	774
Problemy powodowane przez wielowątkowość	775
Tworzenie doskonałego kodu C++	776
Ucz się C++. Nie poprzestań na tym, czego się tu dowiedziałeś!	778
Dokumentacja w internecie	778
Społeczności, w których możesz uzyskać porady i pomoc	779
Podsumowanie	779
Pytania i odpowiedzi	780
Warsztaty	780

Dodatki	781
Dodatek A. Praca z liczbami: dwójkowo i szesnastkowo	783
System liczb dziesiętnych	784
System liczb dwójkowych	784
Dlaczego w komputerach używany jest system dwójkowy?	785
Czym są bity i bajty?	785
Ile bajtów jest w kilobajcie?	786
System liczb szesnastkowych	786
Dlaczego potrzebujemy systemu szesnastkowego?	787
Konwersja na inną podstawę	787
Ogólny proces konwersji	787
Konwersja liczby dziesiętnej na dwójkową	787
Konwersja liczby dziesiętnej na szesnastkową	788
Dodatek B. Słowa kluczowe C++	789
Dodatek C. Kolejność operatorów	791
Dodatek D. Odpowiedzi	793
Dodatek E. Kody ASCII	839
Tabela ASCII znaków wyświetlanych na ekranie	840
Skorowidz	845

Lekcja 8

Wskaźniki i referencje

Jedną z największych zalet języka C++ jest możliwość tworzenia aplikacji wysokiego poziomu, które będą działały niezależnie od komputera. Język C++ pozwala na dostrojenie wydajności aplikacji na poziomie bitów i bajtów. Zrozumienie sposobu działania wskaźników i referencji to ważny krok na drodze do zdobycia umiejętności tworzenia programów efektywnie wykorzystujących dostępne zasoby systemu.

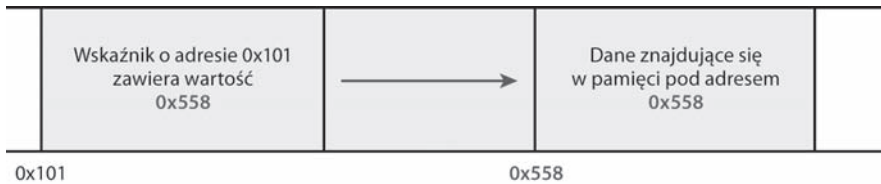
Z tej lekcji dowiesz się:

- ▶ czym są wskaźniki,
- ▶ czym jest pula wolnej pamięci,
- ▶ jak używać operatorów new i delete w celu alokacji i zwalniania pamięci,
- ▶ jak tworzyć stabilne aplikacje za pomocą wskaźników i alokacji dynamicznej,
- ▶ czym są referencje,
- ▶ na czym polegają różnice pomiędzy wskaźnikami i referencjami,
- ▶ kiedy używać wskaźników, a kiedy referencji.

Czym jest wskaźnik?

Ujmując rzecz najprościej, *wskaźnik* to zmienna przechowująca adres w pamięci. Podobnie jak zmienna typu `int` jest przeznaczona do przechowywania wartości w postaci liczby całkowitej, wskaźnik przechowuje adres w pamięci, co pokazano na rysunku 8.1.

RYSUNEK 8.1.
Wizualne przedstawienie wskaźnika



Wskaźnik jest więc zmienną i podobnie jak wszystkie zmienne zabiera pewną ilość pamięci (w przypadku pokazanym na rysunku 8.1 będzie to pamięć pod adresem 0x101). Cechą charakterystyczną wskaźnika jest fakt, że przechowywana w nim wartość (tutaj 0x558) jest interpretowana jako adres w pamięci. Dlatego też wskaźnik jest zmienną specjalną *wskazującą* położenie w pamięci.

Deklaracja wskaźnika

Wskaźnik jest zmienną, więc musi być zadeklarowany. Zwykle deklarujesz wskaźnik, aby prowadzić do wartości określonego typu (np. `int`). Oznacza to, że adres przechowywany we wskaźniku prowadzi do miejsca w pamięci, w którym znajduje się liczba całkowita. Istnieje również możliwość określenia wskaźnika w taki sposób, aby wskazywał blok pamięci (to wskaźnik `void`).

Wskaźnik będący zmienną musi być zadeklarowany w sposób podobny do sposobu deklarowania pozostałych zmiennych:

```
TypWskaźnika * NazwaZmiennejWskaźnika;
```

Tak jak w większości zmiennych, jeśli nie zainicjalizujesz wskaźnika, będzie zawierał losowo wybraną wartość. Ponieważ nie chcesz, aby program uzyskał dostęp do losowo wybranego położenia w pamięci, musisz zainicjalizować wskaźnik z wartością `null`. To `null` jest wartością, którą można wykorzystać w operacji sprawdzania i która jednocześnie nie jest adresem w pamięci:

```
TypWskaźnika * NazwaZmiennejWskaźnika = NULL; // Inicjalizacja wartości wskaźnika.
```

Deklaracja wskaźnika typu liczby całkowitej będzie więc miała postać:

```
int *pInteger = NULL;
```

Wskaźnik, podobnie jak wszystkie poznane dotąd typy danych, zawiera przypadkową wartość dopóty, dopóki nie zostanie zainicjalizowany. Taka przypadkowa wartość jest szczególnie niebezpieczna w przypadku wskaźników, ponieważ wskazuje pewien adres w pamięci. Niezainicjalizowany wskaźnik może spowodować, że program uzyska dostęp do niepoprawnego położenia w pamięci i ulegnie awarii.

Ostrzeżenie
Ostrzeżenie

Określenie adresu zmiennej przy użyciu operatora referencji (&)

Zmienne to narzędzia dostarczane przez język, aby umożliwić pracę z danymi w pamięci. Koncepcja ta została szczegółowo wyjaśniona w lekcji 3., zatytułowanej „Zmienne i stałe”. Wskaźniki również są zmiennymi, ale mają specjalny typ używany jedynie w celu przechowywania adresu w pamięci.

Jeżeli VarName to nazwa zmiennej, wtedy &VarName podaje adres w pamięci, gdzie przechowywana jest wartość wymienionej zmiennej.

Jeśli zatem zadeklarowałeś liczbę całkowitą, używając przedstawionej poniżej doskonale znanej składni:

```
int Age = 30;
```

wówczas &Age będzie adresem w pamięci przechowującym wartość (30) zmiennej. W listingu 8.1 przedstawiono koncepcję adresu w pamięci dla zmiennej w postaci liczby całkowitej, który jest używany do przechowywania wartości zmiennej.

Listing 8.1. Określenie adresu liczb typu int i double

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     int Age = 30;
6:     const double Pi = 3.1416;
7:
8:     // Użycie operatora & do wyszukania adresu w pamięci.
9:     cout << "Liczba całkowita Age jest pod adresem: 0x" << hex << &Age <<
    ↪endl;
```

```

10:     cout << "Liczba Pi typu double jest pod adresem: 0x" << hex << &Pi <<
        ↪endl;
11:
12:     return 0;
13: }

```

Wynik ▼

Liczba całkowita Age jest pod adresem: 0x0045FE00
 Liczba Pi typu double jest pod adresem: 0x0045FDF8

Analiza ▼

Zwróć uwagę na użycie operatora referencji (&) w wierszach 9. i 10. w celu ustalenia adresów zmiennej Age i stałej Pi. Przedrostek 0x został dodany, aby zachować konwencję stosowaną podczas wyświetlania liczb szesnastkowych.

Uwaga

Wiesz już, że ilość pamięci zużywana przez zmienną zależy od jej typu. W listingu 3.4 przedstawionym w lekcji 3. użyto operatora `sizeof()` w celu pokazania, że wielkość liczby całkowitej to 4 bajty (w systemie autora i używanym przez niego kompilatorze). Opierając się na przedstawionych powyżej danych wyjściowych wskazujących na położenie zmiennej Age pod adresem 0x0045FE08 i znając wielkość liczby całkowitej (4 bajty), wiemy, że cztery bajty w położeniu od 0x0045FE00 do 0x0045FE04 należą do liczby całkowitej Age.

Uwaga

Operator referencji (&) jest nazywany również operatorem adresu.

Użycie wskaźników do przechowywania adresów

Dowiedziałeś się już, jak deklarować wskaźniki oraz jak ustalić adres zmiennej. Wiesz także, że wskaźniki są zmiennymi używanymi do przechowywania adresów w pamięci. Najwyższy czas skonsolidować wiedzę i wykorzystać wskaźniki do przechowywania adresów pobranych przy użyciu operatora referencji (&).

Przyjmujemy założenie, że deklaracja zmiennej jest przeprowadzana w znany sposób:

```

// Deklaracja zmiennej.
Typ NazwaZmiennej = WartośćPoczątkowa;

```

W celu przechowywania adresu zmiennej we wskaźniku konieczne jest zadeklarowanie wskaźnika o takim samym typie jak zmienna (Typ) oraz zainicjalizowanie wskaźnika przy użyciu operatora referencji (&) wraz z adresem w pamięci, gdzie znajduje się wartość zmiennej:

```
// Zadeklarowanie wskaźnika takiego samego typu i zainicjalizowanie go z adresem zmiennej
// w pamięci.
Typ* Wskaźnik = &Zmienna;
```

Dlatego też w przypadku zadeklarowania zmiennej w postaci liczby całkowitej i przy użyciu doskonale znanej składni:

```
int Age = 30;
```

możesz zadeklarować wskaźnik typu `int` zawierający rzeczywisty adres w pamięci, gdzie przechowywana jest wartość zmiennej `Age`:

```
int* pInteger = &Age; // Wskaźnik do liczby całkowitej Age.
```

W listingu 8.2 możesz zobaczyć, jak wskaźnik jest używany do przechowywania adresu uzyskanego za pomocą operatora referencji (&).

Listing 8.2. Przykład deklaracji i inicjalizacji wskaźnika

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     int Age = 30;
6:     int* pInteger = &Age; // Wskaźnik typu int, zainicjalizowany z wartością &Age.
7:
8:     // Wyświetlenie wartości wskaźnika.
9:     cout << "Liczba całkowita Age jest pod adresem: 0x" << hex <<
        ↳ pInteger << endl;
10:
11:     return 0;
12: }
```

Wynik ▼

Liczba całkowita Age jest pod adresem: 0x0045FE00

Analiza ▼

Dane wyjściowe powyższego listingu są w zasadzie takie same jak poprzedniego, w obu przypadkach został wyświetlony ten sam komunikat wraz z adresem

w pamięci, gdzie przechowywana jest wartość zmiennej Age. Różnica w stosunku do poprzedniego programu polega na tym, że adres jest najpierw przypisywany wskaźnikowi (wiersz 6.), a następnie wartość wskaźnika (teraz już adres) zostaje wyświetlona przy użyciu polecenia cout (wiersz 9.).

Uwaga
Uwaga

Dane wyjściowe, które otrzymasz, mogą się różnić od przedstawionych w książce. Adres zmiennej może być inny po każdym uruchomieniu programu w tym samym komputerze.

Skoro już wiesz, jak przechowywać adres w zmiennej wskaźnika, bardzo łatwo możesz wyobrazić sobie, że tej samej zmiennej wskaźnika można przypisać inny adres w pamięci. W ten sposób wskaźnik będzie prowadził do innej wartości, co przedstawiono w listingu 8.3.

Listing 8.3. Przypisanie wskaźnikowi adresu innej zmiennej

```

0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     int Age = 30;
6:
7:     int* pInteger = &Age;
8:     cout << "pInteger prowadzi teraz do zmiennej Age" << endl;
9:
10:    // Wyświetlenie wartości wskaźnika.
11:    cout << "pInteger = 0x" << hex << pInteger << endl;
12:
13:    int DogsAge = 9;
14:    pInteger = &DogsAge;
15:    cout << "pInteger prowadzi teraz do zmiennej DogsAge" << endl;
16:
17:    cout << "pInteger = 0x" << hex << pInteger << endl;
18:
19:    return 0;
20: }
```

Wynik ▼

```

pInteger prowadzi teraz do zmiennej Age
pInteger = 0x002EFB34
pInteger prowadzi teraz do zmiennej DogsAge
pInteger = 0x002EFB1C
```


Analiza ▼

Program pokazuje, że wskaźnik (`pInteger`) prowadzący do jednej liczby całkowitej może prowadzić także do zupełnie innej liczby całkowitej. W wierszu 7. nastąpiła jego inicjalizacja wraz z wartością `&Age`, a więc zawierał adres zmiennej `Age`. W wierszu 14. temu samemu wskaźnikowi przypisano wartość `&DogsAge`, co oznacza, że prowadzi do innego położenia w pamięci, które zawiera wartość zmiennej `DogsAge`. Dane wyjściowe pokazują, że wartość wskaźnika (tzn. adres w pamięci) uległa zmianie. Najpierw prowadziła do miejsca w pamięci (`0x002EFB34`) przechowującego liczbę całkowitą `Age`, a później do innego miejsca (`0x002EFB1C`) przechowującego liczbę całkowitą `DogsAge`.

Uzyskanie dostępu do danych przy użyciu operatora dereferencji (*)

Masz wskaźnik do danych zawierający poprawny adres. W jaki sposób można uzyskać dostęp do wspomnianego położenia, tzn. jak pobrać lub zmienić dane we wskazanym położeniu w pamięci? Odpowiedzią jest użycie operatora dereferencji (*) nazywanego także operatorem wyłuskania. Jeżeli masz prawidłowy wskaźnik `pData`, wtedy użycie `*pData` pozwala na uzyskanie dostępu do wartości przechowywanej pod adresem wskazywanym przez wskaźnik. Praktyczne zastosowanie operatora dereferencji przedstawiono w listingu 8.4.

Listing 8.4. Przykład użycia operatora dereferencji (*) w celu uzyskania dostępu do wartości wskazywanej przez wskaźnik

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     int Age = 30;
6:     int DogsAge = 9;
7:
8:     cout << "Liczba całkowita Age = " << Age << endl;
9:     cout << "Liczba całkowita DogsAge = " << DogsAge << endl;
10:
11:     int* pInteger = &Age;
12:     cout << "pInteger prowadzi do zmiennej Age" << endl;
13:
14:     // Wyświetlenie wartości wskaźnika.
```

```
15: cout << "pInteger = 0x" << hex << pInteger << endl;
16:
17: // Wyświetlenie wartości we wskazanym położeniu.
18: cout << "*pInteger = " << dec << *pInteger << endl;
19:
20: pInteger = &DogsAge;
21: cout << "pInteger prowadzi teraz do zmiennej DogsAge" << endl;
22:
23: cout << "pInteger = 0x" << hex << pInteger << endl;
24: cout << "*pInteger = " << dec << *pInteger << endl;
25:
26: return 0;
27: }
```

Wynik ▼

```
Liczba całkowita Age = 30
Liczba całkowita DogsAge = 9
pInteger prowadzi do zmiennej Age
pInteger = 0x0025F788
*pInteger = 30
pInteger prowadzi teraz do zmiennej DogsAge
pInteger = 0x0025F77C
*pInteger = 9
```

Analiza ▼

Poza zmianą adresu przechowywanego przez wskaźnik, co również miało miejsce w poprzednim przykładzie (listing 8.3), w powyższym listingu użyto operatora dereferencji (*) wraz ze zmienną wskaźnika pInteger w celu wyświetlenia różnych wartości przechowywanych w dwóch adresach w pamięci. Zwróć uwagę na wiersze 18. i 24.

W obu tych wierszach dostęp do liczby całkowitej wskazywanej przez wskaźnik pInteger następuje za pomocą operatora dereferencji. Po zmianie adresu we wskaźniku pInteger (patrz wiersz 20.) ten sam wskaźnik prowadzi teraz do wartości zmiennej DogsAge, co powoduje wyświetlenie w danych wyjściowych liczby 9.

Podczas użycia operatora dereferencji adres przechowywany we wskaźniku jest wykorzystywany przez aplikację do pobrania czterech bajtów z pamięci zarezerwowanej dla danej liczby całkowitej (ponieważ to wskaźnik do liczby całkowitej o wielkości 4 bajtów, co potwierdza wynik operacji `sizeof(int)`). Dlatego też zapewnienie poprawności adresu przechowywanego przez wskaźnik

ma znaczenie krytyczne. Przez inicjalizację wskaźnika z wartością &Age (wiersz 11.) gwarantujemy, że wskaźnik zawiera prawidłowy adres w pamięci. Jeżeli nie zainicjalizujesz wskaźnika, może on zawierać dowolną wartość, która istniała w danym położeniu w pamięci w chwili rezerwacji tej pamięci dla wskaźnika. Odwołanie do takiego wskaźnika najczęściej oznacza wystąpienie błędu *Access Violation*, czyli próbę uzyskania przez aplikację dostępu do adresu w pamięci, do którego nie ma uprawnień.

Operator dereferencji jest nazywany również operatorem dostępu pośredniego.

Uwaga
Uwaga

W poprzednim przykładzie wskaźnik był używany w celu odczytu (pobrania) wartości ze wskazanego miejsca w pamięci. W listingu 8.5 pokazano, co się stanie, jeśli wskaźnik *pInteger zostanie użyty jako l-wartość, tzn. kod przypisze go zmiennej, zamiast uzyskać do niego dostęp.

Listing 8.5. Operowanie danymi za pomocą wskaźnika i operatora dereferencji

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     int DogsAge = 30;
6:     cout << "Zainicjalizowana zmienna DogsAge = " << DogsAge << endl;
7:
8:     int* pAge = &DogsAge;
9:     cout << "Wskaźnik pAge prowadzi do zmiennej DogsAge" << endl;
10:
11:     cout << "Podaj wiek psa: ";
12:
13:     // Dane wejściowe zostają umieszczone w pamięci wskazywanej przez wskaźnik pAge.
14:     cin >> *pAge;
15:
16:     // Wyświetlenie adresu, gdzie przechowywana jest wartość.
17:     cout << "Dane wejściowe wskazywane przez pAge są pod adresem 0x" <<
    ↪hex << pAge << endl;
18:
19:     cout << "Liczba całkowita DogsAge = " << dec << DogsAge << endl;
20:
21:     return 0;
22: }
```

Wynik ▼

Zainicjalizowana zmienna DogsAge = 30
Wskaźnik pAge prowadzi do zmiennej DogsAge
Podaj wiek psa: 10
Dane wejściowe wskazywane przez pAge są pod adresem 0x0025FA18
Liczba całkowita DogsAge = 10

Analiza ▼

W omawianym listingu kluczowy krok to wiersz 14., w którym liczba całkowita podana przez użytkownika zostaje zapisana w położeniu w pamięci wskazywanym przez wskaźnik pAge. Zauważ, że pomimo umieszczenia danych wejściowych w położeniu wskazywanym przez pAge, polecenie w wierszu 19. wyświetlające wartość zmiennej DogsAge pokazuje wartość wskazaną przez wskaźnik. Wynika to z faktu, że wskaźnik pAge prowadzi do zmiennej DogsAge, na co wskazuje polecenie inicjalizujące w wierszu 8. Wszelkie zmiany w położeniach w pamięci przechowujących zmienną DogsAge i wskazywanych przez wskaźnik pAge są odzwierciedlane w obu elementach.

Ile pamięci zabiera wskaźnik?

Dowiedziałeś się, że wskaźnik to po prostu rodzaj zmiennej przechowującej adres położenia w pamięci. Dlatego też, niezależnie od wskazywanego typu, zawartością wskaźnika jest adres, czyli liczba. Wielkość adresu to liczba bajtów wymaganych do przechowywania liczby w postaci stałej w danym systemie. Wynik działania operatora sizeof() względem wskaźnika będzie zależał od kompilatora i systemu operacyjnego, dla którego program został skompilowany. Natomiast nie zależy od natury danych, do których prowadzi wskaźnik, co potwierdza program przedstawiony w listingu 8.6.

Listing 8.6. Potwierdzenie, że wskaźniki prowadzące do różnych typów danych mają tę samą wielkość

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     int Age = 30;
6:     double Pi = 3.1416;
7:     char SayYes = 'y';
```

```
8:
9: // Inicjalizacja wskaźników wraz z adresami zmiennych.
10: int* pInt = &Age;
11: double* pDouble = &Pi;
12: char* pChar = &SayYes;
13:
14: cout << "sizeof podstawowych typów -" << endl;
15: cout << "sizeof(int) = " << sizeof(int) << endl;
16: cout << "sizeof(double) = " << sizeof(double) << endl;
17: cout << "sizeof(char) = " << sizeof(char) << endl;
18:
19: cout << "sizeof wskaźników do podstawowych typów -" << endl;
20: cout << "sizeof(pInt) = " << sizeof(pInt) << endl;
21: cout << "sizeof(pDouble) = " << sizeof(pDouble) << endl;
22: cout << "sizeof(pChar) = " << sizeof(pChar) << endl;
23:
24: return 0;
25: }
```

Wynik ▼

```
sizeof podstawowych typów -
sizeof(int) = 4
sizeof(double) = 8
sizeof(char) = 1
sizeof wskaźników do podstawowych typów -
sizeof(pInt) = 4
sizeof(pDouble) = 4
sizeof(pChar) = 4
```

Analiza ▼

Dane wyjściowe jasno pokazują, że nawet gdy wartość `sizeof(char)` wynosi 1 bajt, a `sizeof(double)` wynosi 8 bajtów, to wielkość wskaźnika (`sizeof(pointer)`) jest stała i wynosi 4 bajty. Po prostu ilość pamięci wymagana do przechowywania adresów w pamięci jest taka sama, niezależnie od tego, czy wskaźnik prowadzi do 1, czy do 8 bajtów.

Dane wyjściowe listingu 8.6 pokazują, że wielkość wskaźnika wynosi 4 bajty. Jednak w Twoim systemie operacyjnym wielkość ta może być inna. Przedstawione w książce dane wyjściowe zostały wygenerowane przez kod skompilowany przez 32-bitowy kompilator. Jeżeli używasz 64-bitowego kompilatora i uruchomisz program w 64-bitowym systemie operacyjnym, możesz zobaczyć, że wielkość zmiennej wskaźnika wynosi 64 bity, czyli 8 bajtów.

Uwaga
Uwaga

Dynamiczna alokacja pamięci

Kiedy stworzysz program zawierający deklarację tablicy poniższego typu:

```
int Numbers[100]; // Tablica statyczna stu liczb całkowitych.
```

program boryka się z dwoma problemami. Oto one.

1. Ograniczasz pojemność programu, który będzie mógł przechowywać maksymalnie sto liczb całkowitych.
2. Zmniejszasz wydajność systemu, w przypadku gdy tablica zarezerwowana dla stu liczb całkowitych będzie przechowywała tylko jedną.

Wymienione problemy istnieją z powodu alokacji pamięci dla tablicy — dla podanej wcześniej deklaracji tablicy kompilator rezerwuje statyczną ilość pamięci.

Aby utworzyć aplikację optymalnie wykorzystującą zasoby pamięci na podstawie potrzeb i działań użytkownika, należy zastosować dynamiczną alokację pamięci. W ten sposób będzie można zaalokować większą ilość pamięci, gdy będzie potrzebna, oraz zwalniać pamięć, kiedy nie będziemy z niej korzystać. Język C++ oferuje dwa operatory — `new` i `delete` — pomagające w lepszym zarządzaniu pamięcią przez aplikację. Wskaźniki będące zmiennymi używanymi do przechowywania adresów w pamięci odgrywają krytyczną rolę w efektywnym, dynamicznym zarządzaniu pamięcią.

Użycie operatorów `new` i `delete` w celu dynamicznej alokacji i zwalniania pamięci

Operator `new` służy do alokacji nowych bloków pamięci. Najczęściej używana forma operatora `new` powoduje zwrot wskaźnika do żądanej pamięci, o ile operacja zakończyła się powodzeniem, w przeciwnym razie następuje zgłoszenie wyjątku. Podczas użycia operatora `new` konieczne jest wskazanie typu danych, dla których następuje alokacja pamięci:

```
Typ* Wskaźnik = new Typ; // Żądanie pamięci dla jednego elementu.
```

Istnieje również możliwość wskazania liczby elementów, dla których ma być zaalokowana pamięć (w przypadku alokacji pamięci dla więcej niż tylko jednego elementu):

```
Typ* Wskaźnik = new Typ[LiczbaElementów]; // Żądanie pamięci dla podanej liczby // elementów.
```

Dlatego też, jeśli musisz zaalokować pamięć dla liczb całkowitych, możesz skorzystać z poniższych składni:

```
int* pNumber = new int; // Pobranie wskaźnika do liczby całkowitej.
int* pNumbers = new int[10]; // Pobranie wskaźnika do bloku dziesięciu liczb całkowitych.
```

Zauważ, że `new` oznacza żądanie pamięci. Nie ma gwarancji, że wykonanie polecenia żądania pamięci zakończy się powodzeniem — zależy to od stanu systemu oraz dostępności zasobów pamięci.

Uwaga
Uwaga

Dla każdej przeprowadzonej operacji alokacji pamięci za pomocą operatora `new` konieczne jest przeprowadzenie operacji zwolnienia tej pamięci przy użyciu operatora `delete`:

```
Typ* Wskaźnik = new Typ;
delete Wskaźnik; // Zwolnienie pamięci zaalokowanej wcześniej dla jednego egzemplarza
                // wskazanego typu.
```

Ta sama reguła obowiązuje w przypadku żądania alokacji pamięci dla wielu elementów:

```
Typ* Wskaźnik = new Typ[LiczbaElementów];
delete[] Wskaźnik; // Zwolnienie zaalokowanego powyżej bloku pamięci.
```

Zwróć uwagę na użycie `delete[]` w czasie alokacji bloku pamięci przy użyciu operatora `new[...]` i `delete` po alokacji tylko jednego elementu za pomocą operatora `new`.

Uwaga
Uwaga

Jeżeli nie zwolnisz zaalokowanej wcześniej pamięci, której już nie używasz, pozostanie zarezerwowana dla aplikacji. To z kolei zmniejszy ilość pamięci systemowej dostępnej dla innych programów i prawdopodobnie zmniejszy szybkość działania Twojej aplikacji. Taka sytuacja nosi nazwę *wycieku pamięci*; należy jej unikać za wszelką cenę.

Kod przedstawiony w listingu 8.7 pokazuje dynamiczną alokację i zwalnianie pamięci.

Listing 8.7. Wykorzystanie operatora dereferencji w celu uzyskania dostępu do pamięci zaalokowanej przez operator `new` oraz jej zwolnienie za pomocą operatora `delete`

```
0: #include <iostream>
1: using namespace std;
2:
```

```

3: int main()
4: {
5:     // Żądanie alokacji pamięci dla elementu typu int.
6:     int* pAge = new int;
7:
8:     // Użycie zaalokowanej pamięci do przechowywania liczby.
9:     cout << "Podaj wiek psa: ";
10:    cin >> *pAge;
11:
12:    // Użycie operatora dereferencji * w celu uzyskania dostępu do przechowywanej wartości.
13:    cout << "Wartość zmiennej Age wynosi " << *pAge << " i jest
    ↳przechowywana pod adresem 0x" << hex << pAge << endl;
14:
15:    delete pAge; // Zwolnienie pamięci.
16:
17:    return 0;
18: }

```

Wynik ▼

Podaj wiek psa: 9

Wartość zmiennej Age wynosi 9 i jest przechowywana pod adresem 0x00338120

Analiza ▼

W wierszu 6. pokazano przykład użycia operatora `new` w celu alokacji pamięci dla liczby całkowitej. W zarezerwowanej pamięci będą przechowywane podane przez użytkownika dane wejściowe, czyli wiek psa. Zauważ, że operator `new` zwraca wskaźnik i to jest powodem przypisania mu wartości. Wiek podany przez użytkownika jest przechowywany w nowo zaalokowanej pamięci, odbywa się to w wierszu 10. za pomocą polecenia `cin` i operatora dereferencji (*). W wierszu 13. kod wyświetla przechowywaną wartość (ponownie posługując się operatorem dereferencji), a także adres w pamięci, pod którym znajduje się przechowywana wartość. Zwróć uwagę, że adres znajdujący się we wskaźniku `pAge` (wiersz 13.) to ten sam adres, który został zwrócony przez operator `new` w wierszu 6. i od tamtej chwili nie uległ zmianie.

Ostrzeżenie

Operator `delete` nie może być wywoływany względem dowolnego adresu znajdującego się we wskaźniku, to musi być adres zwrócony wcześniej przez operator `new` i tylko ten, który nie był wcześniej zwolniony przez inne wywołanie `delete`.

Mimo iż wskaźniki przedstawione w listingu 8.6 zawierają poprawne adresy, jednak nie powinny być zwalniane za pomocą operatora `delete`, ponieważ wspomniane adresy nie zostały otrzymane na skutek wywołania operatora `new`.

Pamiętaj, że po użyciu `new[...]` w celu alokacji bloku pamięci dla wielu elementów zwolnienie pamięci musi nastąpić przez wywołanie `delete[]`, jak to przedstawiono w listingu 8.8.

Listing 8.8. Alokacja pamięci przy użyciu operatora `new[...]` i jej zwolnienie za pomocą operatora `delete[]`

```
0: #include <iostream>
1: #include <string>
2: using namespace std;
3:
4: int main()
5: {
6:     cout << "Podaj imię: ";
7:     string Name;
8:     cin >> Name;
9:
10:    // Dodanie 1 do rezerwowanej pamięci (na znak null).
11:    int CharsToAllocate = Name.length() + 1;
12:
13:    // Żądanie pamięci potrzebnej do przechowywania kopii danych wejściowych.
14:    char* CopyOfName = new char [CharsToAllocate];
15:
16:    // Funkcja strcpy() kopiuje dane z ciągu tekstowego zakończonego znakiem null.
17:    strcpy(CopyOfName, Name.c_str());
18:
19:    // Wyświetlenie skopiowanego ciągu tekstowego.
20:    cout << "Dynamicznie zaalokowany bufor zawiera: " << CopyOfName <<
    ↪endl;
21:
22:    // Po zakończeniu pracy z buforem należy go usunąć.
23:    delete[] CopyOfName;
24:
25:    return 0;
26: }
```

Wynik ▼

Podaj imię: Robert
Dynamicznie zaalokowany bufor zawiera: Robert

Analiza ▼

Najbardziej interesujące nas wiersze w powyższym programie to te, które zawierają operatory `new` i `delete[]`, czyli (odpowiednio) wiersze 11. i 23. Różnica w stosunku do poprzedniego przykładu (patrz listing 8.7) polega na alokacji bloku pamięci wystarczającego do przechowywania wielu elementów — kod w listingu 8.7 alokował pamięć tylko dla jednego elementu. Alokacji pamięci dla tablicy elementów musi odpowiadać użycie operatora `delete[]` w celu zwolnienia pamięci po zakończeniu pracy z tablicą. Liczba znaków, dla których trzeba zaalokować pamięć, jest obliczana w wierszu 11. Liczba ta jest o jeden większa od liczby znaków wprowadzonych przez użytkownika, ponieważ trzeba uwzględnić znak `null`, bardzo ważny w ciągach tekstowych typu `C`. Konieczność umieszczenia znaku `null` na końcu ciągu tekstowego została wyjaśniona w lekcji 4., zatytułowanej „Tablice i ciągi tekstowe”. Rzeczywista operacja kopiowania odbywa się w wierszu 17., jest przeprowadzana przez funkcję `strcpy()` używającą z kolei funkcji `c_str()` dostarczanej przez klasę `std::string`. Znaki są kopiowane do bufora typu `char` o nazwie `CopyOfName`.

Uwaga

Operatory `new` i `delete` alokują pamięć pochodzącą z puli wolnej pamięci. Pula wolnej pamięci to rodzaj abstrakcji w postaci puli pamięci, którą aplikacja może zaalokować (tzn. zarezerwować) oraz dealokować (tzn. zwalniać).

Efektywne użycie operatorów inkrementacji (`++`) i dekrementacji (`--`) na wskaźnikach

Wskaźnik zawiera adres w pamięci. Przykładowo w przedstawionym wcześniej listingu 8.3 wskaźnik prowadzący do liczby całkowitej zawiera wartość `0x002EFB34`, czyli adres w pamięci, gdzie jest przechowywana wspomniana liczba całkowita. Sama liczba ma wielkość czterech bajtów, a tym samym zabiera cztery miejsca w pamięci od `0x002EFB34` do `0x002EFB37`. Inkrementacja wskaźnika za pomocą operatora `++` *nie spowoduje*, że wskaźnik będzie prowadził do `0x002EFB35`. Takie działanie, które ma na celu wskazanie środka liczby całkowitej, jest dosłownie bezcelowe.

Operacja inkrementacji lub dekrementacji wskaźnika jest interpretowana przez kompilator jako potrzeba wskazania kolejnej wartości w bloku pamięci, przy przyjęciu założenia, że będzie ona takiego samego typu, a *nie następnym bajtem* (o ile wartość nie ma długości jednego bajta, np. `char`).

Dlatego też inkrementacja wskaźnika, takiego jak `pInteger`, użytego w listingu 8.3 powoduje inkrementację o cztery bajty, co odpowiada wynikowi wywołania `sizeof(int)`. Użycie operatora `++` względem wskaźnika oznacza poinformowanie kompilatora, że wskaźnik ma prowadzić do kolejnej liczby całkowitej. Po operacji inkrementacji wskaźnik będzie prowadził do adresu `0x002EFB38`. Podobnie dodanie wartości 2 do wskaźnika spowoduje jego przesunięcie o dwie liczby całkowite do przodu, czyli o osiem bajtów. Dalej w tej lekcji zobaczysz korelację pomiędzy tym zachowaniem wskaźników a indeksami używanymi w tablicach.

Dekrementacja wskaźników przy użyciu operatora `--` daje taki sam efekt, tzn. wartość adresu zawartego we wskaźniku zostaje zmniejszona o wynik operacji `sizeof` przeprowadzonej na typie danych wskazywanych przez wskaźnik.

Co się stanie po inkrementacji lub dekrementacji wskaźnika?

Adres znajdujący się we wskaźniku jest inkrementowany lub dekrementowany o wartość zwrotną działania operatora `sizeof` względem typu danych wskazywanego przez wskaźnik (to niekoniecznie będzie bajt). Dzięki temu kompilator gwarantuje, że wskaźnik nigdy nie będzie prowadził do środka lub na koniec danych umieszczonych pod danym adresem w pamięci — wskaźnik zawsze prowadzi na początek danych.

Jeżeli wskaźnik zostałby zadeklarowany w postaci:

```
Typ* pTyp = Adres;
```

wówczas `++pTyp` oznacza, że `pTyp` zawiera (a więc wskazuje) wartość `Adres + sizeof(Typ)`.

W kodzie przedstawionym w listingu 8.9 znajdziesz wyjaśnienie efektu inkrementacji wskaźnika i dodawania do niego wartości przesunięcia.

Listing 8.9. Alokacja dynamiczna w zależności od potrzeb, inkrementacja wskaźników za pomocą wartości przesunięcia i operatora `++`

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     cout << "Ile liczb całkowitych chcesz podać? ";
6:     int InputNums = 0;
7:     cin >> InputNums;
8:
9:     int* pNumbers = new int [InputNums]; // Alokacja żądanych liczb całkowitych.
```

```
10: int* pCopy = pNumbers;
11:
12: cout<<"Udało się zaalokować pamięć dla "<<InputNums<< " liczb
    ↳całkowitych"<<endl;
13: for(int Index = 0; Index < InputNums; ++Index)
14: {
15:     cout << "Podaj liczbę "<< Index << ": ";
16:     cin >> *(pNumbers + Index);
17: }
18:
19: cout << "Wyświetlenie wszystkich podanych liczb: " << endl;
20: for(int Index = 0; Index < InputNums; ++Index)
21:     cout << *(pCopy++) << " ";
22:
23:     cout << endl;
24:
25:     // Po zakończeniu pracy ze wskaźnikiem trzeba zwolnić pamięć.
26:     delete[] pNumbers;
27:
28:     return 0;
29: }
```

Wynik ▼

```
Ile liczb całkowitych chcesz podać? 2
Udało się zaalokować pamięć dla 2 liczb całkowitych
Podaj liczbę 0: 789
Podaj liczbę 1: 575
Wyświetlenie wszystkich podanych liczb:
789 575
```

Następne uruchomienie programu:

```
Ile liczb całkowitych chcesz podać? 5
Udało się zaalokować pamięć dla 5 liczb całkowitych
Podaj liczbę 0: 789
Podaj liczbę 1: 12
Podaj liczbę 2: -65
Podaj liczbę 3: 285
Podaj liczbę 4: -101
Wyświetlenie wszystkich podanych liczb:
789 12 -65 285 -101
```

Analiza ▼

Przed alokacją pamięci w wierszu 9. program prosi użytkownika o podanie ilości liczb całkowitych, które mają być przechowywane w systemie. Zwróć uwagę na wykonanie kopii adresu w wierszu 10. do późniejszego użycia

podczas zwalniania bloku pamięci w wierszu 26. przy użyciu operatora `delete`. W tym programie pokazano zalety użycia wskaźników i dynamicznej alokacji pamięci w porównaniu do zastosowania tablicy statycznej. Omawiana aplikacja zużywa mniejszą ilość pamięci, gdy użytkownik przechowuje małą ilość liczb całkowitych, oraz większą, kiedy trzeba przechować dużą ilość liczb całkowitych. W żadnym z wymienionych przypadków nie są marnowane zasoby systemowe. Z powodu zastosowania dynamicznej alokacji pamięci nie ma żadnej sztucznej górnej granicy ilości liczb całkowitych, które mogą być przechowywane — jedynym ograniczeniem jest dostępność zasobów systemowych. W wierszach od 13. do 17. znajduje się pętla `for` pozwalająca użytkownikowi na wprowadzenie liczb, które następnie są przechowywane w kolejnych miejscach w pamięci przy użyciu wyrażenia zdefiniowanego w wierszu 16. Do wskaźnika została dodana liczone od zera wartość przesunięcia (`Index`). Kompilator tworzy więc program wstawiający wartość podaną przez użytkownika w odpowiednim miejscu w pamięci bez nadpisywania poprzedniej. Innymi słowy, `(pNumber + Index)` to wyrażenie, które zwraca wskaźnik prowadzący do liczby całkowitej w rozpoczynającym się od zera indeksie położenia w pamięci. W takim przypadku wartość 1 indeksu wskazuje drugą liczbę całkowitą. Z kolei operator dereferencji w `*(pNumber + Index)` to wyrażenie używane przez polecenie `cin` w celu uzyskania dostępu do wartości we wspomnianym indeksie rozpoczynającym się od zera. Pętla `for` w wierszach 21. i 22. jest używana do wyświetlania wartości przechowywanych w poprzedniej pętli. Polecenie `for` wykorzystuje wiele inicjalizatorów, tworzy kopię w `pCopy` i inkrementuje tę kopię w wierszu 21., aby wyświetlić wartość.

Powodem utworzenia kopii w wierszu 10. jest fakt, że pętla z wykorzystaniem operatora inkrementacji modyfikuje `(++)` używany wskaźnik. Początkowy wskaźnik zwrócony przez operator `new` musi być zachowany nietknięty, aby w wierszu 26. można było wywołać odpowiednie polecenie `delete[]` wraz z dokładną wartością podaną przez operator `new`, a nie inną, przypadkową wartość.

Użycie słowa kluczowego `const` ze wskaźnikami

W lekcji 3. dowiedziałeś się, że deklaracja zmiennej jako `const` powoduje przypisanie tego rodzaju zmiennej wartości na stałe, tzn. wartości, która nie może być zmieniana w trakcie cyklu życiowego zmiennej. Tak więc wartość takiej zmiennej nie może być zmodyfikowana lub użyta jako l-wartość.

Wskaźniki również są zmiennymi, a tym samym słowo kluczowe `const` ma dla nich znaczenie. Jednak wskaźnik to specjalny rodzaj zmiennej zawierającej adres w pamięci i używanej do modyfikacji bloku danych w pamięci. Dlatego też dla wskaźników i stałych mamy do dyspozycji następujące rozwiązania.

- ▶ Dane wskazywane przez wskaźnik są stałą i nie mogą być zmodyfikowane, adres zawarty we wskaźniku może ulec zmianie, tzn. wskaźnik może prowadzić do zupełnie innego miejsca w pamięci.

```
int HoursInDay = 24;
const int* pInteger = &HoursInDay; // Nie można użyć pInteger do zmiany.
HoursInDay
int MonthsInYear = 12;
pInteger = &MonthsInYear; // OK!
*pInteger = 13; // Błąd kompilacji: nie można zmienić danych.
int* pAnotherPointerToInt = pInteger; // Błąd kompilacji: stalej nie można
// przepisać niestalej.
```

- ▶ Adres zawarty we wskaźniku jest stałą i nie może być zmodyfikowany, ale zmienione mogą być dane, do których prowadzi wskaźnik.

```
int DaysInMonth = 30;
// Wskaźnik pInteger nie może prowadzić do niczego innego.
int* const pDaysInMonth = &DaysInMonth;
*pDaysInMonth = 31; // OK! Wartość może być zmieniona.
int DaysInLunarMonth = 28;
pDaysInMonth = &DaysInLunarMonth; // Błąd kompilacji: nie można zmienić
// adresu!
```

- ▶ Zarówno adres znajdujący się we wskaźniku, jak i wartość, do której on prowadzi, są stałymi i nie mogą być zmodyfikowane (to najbardziej restrykcyjny wariant).

```
int HoursInDay = 24;
// Wskaźnik może prowadzić jedynie do HoursInDay.
const int* const pHoursInDay = &HoursInDay;
*pHoursInDay = 25; // Błąd kompilacji: nie można zmienić wartości wskazywanej
// przez wskaźnik.
int DaysInMonth = 30;
pHoursInDay = &DaysInMonth; // Błąd kompilacji: nie można zmienić wartości
// wskaźnika.
```

Wymienione różne formy `const` są szczególnie użyteczne podczas przekazywania wskaźników funkcjom. Parametry funkcji muszą być zadeklarowane w sposób zapewniający obsługę maksymalnie najwyższego (restrykcyjnego) poziomu zachowania stałości, aby zagwarantować, że funkcja (jeśli nie ma takiej potrzeby) nie będzie modyfikowała wartości,

do której prowadzi wskaźnik. Takie rozwiązanie pomaga w zachowaniu przejrzystości funkcji, zwłaszcza gdy zmiany w funkcji są wprowadzane na przestrzeni dłuższego czasu lub zmieniają się pracujący nad nią programiści.

Przekazywanie wskaźników funkcjom

Wskaźniki to efektywny sposób przekazania funkcji adresu w pamięci, zawierającego wartość, lub zwrócenia wyniku wygenerowanego przez funkcję. Podczas używania wskaźników z funkcjami bardzo ważne staje się zapewnienie, że funkcja wywołująca może modyfikować jedynie te parametry, które chcesz zmienić, a nie inne. Przykładowo funkcja obliczająca pole okręgu na podstawie danego promienia przekazywanego przez wskaźnik nie powinna mieć możliwości modyfikacji wspomnianego promienia. W takiej sytuacji użycie słowa kluczowego `const` we wskaźniku pozwala na efektywną kontrolę tego, co funkcja może modyfikować i czego nie może. Przykład rozwiązania przedstawiono w listingu 8.10.

Listing 8.10. Użycie słowa kluczowego `new` podczas obliczania pola okręgu, gdy promień i π są dostarczane w postaci stałych

```
0: #include <iostream>
1: using namespace std;
2:
3: void CalcArea(const double* const pPi, // Stała wskaźnika do stałej danych.
4:              const double* const pRadius, // Nic nie można zmienić.
5:              double* const pArea) //Zmiana wskazywanej wartości, a nie adresu.
6: {
7:     // Przed użyciem wskaźników należy je sprawdzić!
8:     if (pPi && pRadius && pArea)
9:         *pArea = (*pPi) * (*pRadius) * (*pRadius);
10: }
11:
12: int main()
13: {
14:     const double Pi = 3.1416;
15:
16:     cout << "Podaj promień okręgu: ";
17:     double Radius = 0;
18:     cin >> Radius;
19:
20:     double Area = 0;
21:     CalcArea (&Pi, &Radius, &Area);
22:
23:     cout << "Pole wynosi = " << Area << endl;
```

```
24:  
25:     return 0;  
26: }
```

Wynik ▼

Podaj promień okręgu: 10.5
Pole wynosi = 346.361

Analiza ▼

W wierszach od 3. do 5. pokazano dwie formy `const`, gdzie zarówno wskaźnik `pRadius`, jak i `pPi` zostały dostarczone w postaci „stałego wskaźnika do stałych danych”. Tak więc nie ma możliwości modyfikacji adresu znajdującego się we wskaźniku oraz danych, do których ten wskaźnik prowadzi. Wskaźnik `pArea` to parametr przeznaczony do przechowywania danych wyjściowych, wartość wskaźnika (adres) nie może być modyfikowana, ale dane, do których prowadzi, mogą być zmienione. W wierszu 7. widać, jak parametry funkcji w postaci wskaźników są sprawdzane pod kątem poprawności, zanim zostaną użyte. Nie chcesz przecież, aby funkcja obliczyła pole, jeśli wywołujący przypadkowo przekaże wskaźnik `null` jako dowolny z trzech wymienionych parametrów, ponieważ to doprowadzi do awarii aplikacji.

Podobieństwa pomiędzy tablicami i wskaźnikami

Czy nie uważasz, że przykład przedstawiony w listingu 8.9, gdzie wskaźnik był inkrementowany za pomocą rozpoczynającego się od zera indeksu w celu uzyskania dostępu do kolejnej liczby całkowitej w pamięci, ma wiele wspólnego ze sposobem indeksowania tablic? Kiedy deklarujesz tablicę liczb całkowitych przy użyciu poniższego polecenia:

```
int MyNumbers[5];
```

kompilator alokuje ściśle określoną i stałą ilość pamięci do przechowywania pięciu liczb całkowitych i daje wskaźnik prowadzący do pierwszego elementu tablicy. Wskaźnik jest identyfikowany przez nazwę przypisaną zmiennej tablicy. Innymi słowy, `MyNumber` to wskaźnik prowadzący do pierwszego elementu — `MyNumber[0]`. Przedstawioną korelację zaprezentowano w listingu 8.11.

Listing 8.11. Pokazanie, że zmienna w postaci tablicy jest wskaźnikiem do pierwszego elementu

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     // Tablica statyczna pięciu liczb całkowitych.
6:     int MyNumbers[5];
7:
8:     // Tablica zostaje przypisana wskaźnikowi typu int.
9:     int* pNumbers = MyNumbers;
10:
11:    // Wyświetlenie adresu przechowywanego przez wskaźnik.
12:    cout << "pNumbers = 0x" << hex << pNumbers << endl;
13:
14:    // Adres pierwszego elementu tablicy.
15:    cout << "&MyNumbers[0] = 0x" << hex << &MyNumbers[0] << endl;
16:
17:    return 0;
18: }
```

Wynik ▼

```
pNumbers = 0x004BFE8C
&MyNumbers[0] = 0x004BFE8C
```

Analiza ▼

W powyższym prostym programie pokazano, że zmienna tablicy może być przypisana wskaźnikowi tego samego typu (wiersz 9.). W zasadzie otrzymujemy potwierdzenie, że tablica jest zbliżona do wskaźnika. W wierszach od 12. do 15. pokazano, że adres przechowywany we wskaźniku jest taki sam jak adres w pamięci dla umieszczenia pierwszego elementu tablicy (czyli o wartości indeksu 0). Ten program jasno pokazuje, że tablica jest wskaźnikiem do pierwszego elementu znajdującego się w tablicy.

Dostęp do drugiego elementu tablicy uzyskasz za pomocą wyrażenia `MyNumbers[1]`, ale możesz również wykorzystać wskaźnik `pNumbers` i składnię `*(pNumbers + 1)`. Trzeci element tablicy statycznej jest dostępny przy użyciu wyrażenia `MyNumbers[2]`, natomiast w tablicy dynamicznej można w tym celu użyć składni `*(pNumbers + 2)`.

Ponieważ zmienne tablic to w zasadzie wskaźniki, powinno być możliwe wykorzystanie operatora dereferencji (*) używanego we wskaźnikach podczas pracy z tablicami. Podobnie powinno być możliwe użycie operatora tablicy ([]) do pracy ze wskaźnikami, co przedstawiono w listingu 8.12.

Listing 8.12. Uzyskanie dostępu do elementów tablicy za pomocą operatora dereferencji (*) i użycie operatora tablicy ([]) wraz ze wskaźnikiem

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     const int ARRAY_LEN = 5;
6:
7:     // Zainicjalizowana tablica statyczna pięciu liczb całkowitych.
8:     int MyNumbers[ARRAY_LEN] = {24, -1, 365, -999, 2011};
9:
10:    // Wskaźnik zainicjalizowany z pierwszym elementem tablicy.
11:    int* pNumbers = MyNumbers;
12:
13:    cout << "Wyświetlenie tablicy za pomocą składni wskaźnika,
14:    ↪operatora*" << endl;
15:    for (int Index = 0; Index < ARRAY_LEN; ++Index)
16:        cout << "Element " << Index << " = " << *(MyNumbers + Index) <<
17:        ↪endl;
18:
19:    cout << "Wyświetlenie tablicy za pomocą wskaźnika wraz ze składnią
20:    ↪tablicy, operator[]" << endl;
21:    for (int Index = 0; Index < ARRAY_LEN; ++Index)
22:        cout << "Element " << Index << " = " << pNumbers[Index] << endl;
23:
24:    return 0;
25: }
```

Wynik ▼

Wyświetlenie tablicy za pomocą składni wskaźnika, operatora*

Element 0 = 24

Element 1 = -1

Element 2 = 365

Element 3 = -999

Element 4 = 2011

Wyświetlenie tablicy za pomocą wskaźnika wraz ze składnią tablicy, operator[]

Element 0 = 24

Element 1 = -1

```
Element 2 = 365  
Element 3 = -999  
Element 4 = 2011
```

Analiza ▼

Aplikacja deklaruje tablicę statyczną pięciu liczb całkowitych, która jest zainicjalizowana wraz z pięcioma wartościami (patrz wiersz 8.). Program wyświetla na ekranie zawartość tablicy, korzysta przy tym z dwóch alternatywnych rozwiązań. Pierwsze polega na użyciu zmiennej tablicy wraz z operatorem dereferencji (patrz wiersz 15.), natomiast drugie wykorzystuje zmienną wskaźnika wraz z operatorem tablicy (patrz wiersz 19.).

Działanie programu pokazuje więc, że zarówno tablica `MyNumbers`, jak i wskaźnik `pNumber` w rzeczywistości działają jak wskaźnik. Innymi słowy, deklaracja tablicy jest podobna do wskaźnika, który będzie utworzony w celu przeprowadzania operacji w ramach na stałe zdefiniowanego obszaru pamięci. Zwróć uwagę na możliwość przypisania tablicy wskaźnikowi (wiersz 11.), ale też na brak możliwości przypisania wskaźnika do tablicy — wynika to z natury tablicy statycznej, która nie może być użyta jako l-wartość.

Musisz koniecznie zapamiętać, że wskaźniki zaalokowane dynamicznie za pomocą operatora `new` muszą być zwalniane przy użyciu operatora `delete` nawet wtedy, kiedy skorzystałeś ze składni podobnej do składni tablicy statycznej.

Jeżeli zapomnisz o tym, w aplikacji wystąpi wyciek pamięci, a tego należy unikać za wszelką cenę.

Ostrzeżenie
Ostrzeżenie

Najczęstsze błędy programistyczne popełniane podczas używania wskaźników

Język C++ pozwala na dynamiczną alokację pamięci, co sprawia, że zużycie pamięci przez aplikację jest optymalne. W przeciwieństwie do nowszych języków programowania, takich jak C# i Java, opartych na środowisku uruchomieniowym, C++ nie oferuje mechanizmu automatycznego usuwania nieużytków odpowiedzialnego za zwalnianie pamięci zaalokowanej przez aplikację, ale nieużywanej. Ponieważ temat wskaźników jest trudny, programista ma — niestety — wiele możliwości popełnienia błędów.

Wycieki pamięci

Jest to prawdopodobnie jeden z najczęściej spotykanych problemów z aplikacjami utworzonymi w C++: im dłużej program działa, tym większą ilość pamięci zużywa, a sam system działa coraz wolniej. Tego rodzaju sytuacja występuje zwykle wtedy, kiedy programista nie zadbał o to, by w aplikacji pamięć dynamicznie zaalokowana przy użyciu operatora `new` była zwalniana przez odpowiednie wywołanie `delete`, gdy dany blok pamięci nie jest dłużej potrzebny.

Obowiązkiem programisty jest zagwarantowanie, że cała zaalokowana przez aplikację pamięć zostanie przez nią zwolniona. Poniżej przedstawiono sytuację, która nigdy nie powinna mieć miejsca.

```
int* pNumbers = new int [5]; // Początkowa alokacja.
// Użycie wskaźnika pNumbers.
...
// Zapomnienie o konieczności użycia polecenia delete[] pNumbers; w celu zwolnienia pamięci.
...
// Przeprowadzenie innej alokacji i nadpisanie wskaźnika.
pNumbers = new int [10]; // Wyciek poprzednio zaalokowanej pamięci.
```

Kiedy wskaźnik nie prowadzi do poprawnego adresu w pamięci?

Kiedy odwołujesz się do wskaźnika za pomocą operatora dereferencji w celu uzyskania dostępu do wartości wskazywanej przez ten wskaźnik, musisz mieć 100% pewności, że wskaźnik zawiera prawidłowy adres w pamięci. W przeciwnym razie program może ulec awarii lub zachowywać się niezgodnie z oczekiwaniami. Wydaje się logiczne, że nieprawidłowe wskaźniki to również częsta przyczyna awarii aplikacji. Wskaźnik może być nieprawidłowy z wielu różnych powodów, ale wszystkie są powiązane z kiepskim zarządzaniem pamięcią. Typową sytuacją, w której wskaźnik może stać się nieprawidłowy, przedstawiono w listingu 8.13.

Listing 8.13. Przykład złego programowania z użyciem niepoprawnych wskaźników

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     // Niezainicjalizowany wskaźnik (błąd).
6:     int* pTemperature;
7:
```

```
8:  cout << "Czy jest pogodnie (t/n)?" << endl;
9:  char userInput = 't';
10:  cin >> userInput;
11:
12:  if (userInput == 't')
13:  {
14:      pTemperature = new int;
15:      *pTemperature = 30;
16:  }
17:
18:  // Wskaźnik pTemperature zawiera nieprawidłową wartość, jeśli użytkownik nacisnął
   ↪ klawisz 'n'.
19:  cout << "Temperatura wynosi: " << *pTemperature;
20:
21:  // Wywołanie operatora delete nawet wtedy, gdy nie został użyty operator new.
22:  delete pTemperature;
23:
24:  return 0;
25: }
```

Wynik ▼

```
Czy jest pogodnie (t/n)? t
Temperatura wynosi: 30
```

Następne uruchomienie programu:

```
Czy jest pogodnie (t/n)? n
<AWARIA PROGRAMU!>
```

Analiza ▼

W powyższym programie występuje wiele problemów, niektóre z nich zostały już wskazane w kodzie. Zwróć uwagę na alokację pamięci i przypisanie jej wskaźnikowi (wiersz 14.), co następuje warunkowo po naciśnięciu klawisza *t* przez użytkownika. W przypadku naciśnięcia innego dowolnego klawisza blok `if` nie zostanie wykonany i wskaźnik `pTemperature` pozostanie nieprawidłowy. Dlatego też naciśnięcie klawisza *n* przez użytkownika podczas drugiego uruchomienia programu prowadzi do awarii aplikacji. Powód awarii to nieprawidłowy adres w pamięci przechowywany przez wskaźnik `pTemperature`, do którego odwołanie następuje w wierszu 19.

Równie niebezpieczne jest przeprowadzone w wierszu 22. wywołanie operatora `delete` względem wskaźnika, dla którego nie zaalokowano pamięci przy użyciu operatora `new`. Jeżeli masz kopię wskaźnika, wywołanie `delete`

wystarczy przeprowadzić tylko na jednej kopii (staraj się również unikać posiadania zbyt wielu kopii wskaźnika).

Lepsza (tzn. bezpieczniejsza i stabilniejsza) wersja programu przedstawionego w listingu 8.13 będzie miała zainicjalizowane wskaźniki używane po wcześniejszym sprawdzeniu poprawności oraz zwalniane jednokrotnie i tylko wtedy, gdy są poprawne.

Zawieszane wskaźniki (nazywane również zabłąkanymi)

Zauważ, że każdy prawidłowy wskaźnik staje się nieprawidłowy po jego zwolnieniu za pomocą operatora `delete`. Jeżeli wskaźnik `pTemperature` miałby być użyty po wierszu 22., nawet w sytuacji, gdy użytkownik nacisnął klawisz `t` i wskaźnik zachował ważność do tej chwili, po wywołaniu `delete` staje się nieprawidłowy i nie powinien być używany.

Aby uniknąć tego rodzaju problemu, wielu programistów przypisuje wskaźnikom wartość `null` podczas inicjalizacji lub po ich zwolnieniu, a przed użyciem wskaźnika za pomocą operatora dereferencji sprawdza jego poprawność.

Najlepsze praktyki podczas pracy ze wskaźnikami

Poniżej wymieniono pewne podstawowe reguły, których zastosowanie podczas używania wskaźników w aplikacji powinno ułatwić pracę.

TAK	NIE
Zawsze przeprowadzaj inicjalizację zmiennych wskaźników, ponieważ w przeciwnym razie będą zawierały zupełnie przypadkowe wartości. Wspomniane wartości są interpretowane jako adresy w pamięci, ale aplikacja nie ma uprawnień do uzyskania do nich dostępu. Jeżeli nie możesz zainicjalizować wskaźnika z poprawnym adresem zwróconym przez operator <code>new</code> lub inną prawidłową zmienną, zainicjalizuj go z wartością <code>null</code> .	<p>Nie próbuj uzyskać dostępu do bloku pamięci, używając wskaźnika po jego zwolnieniu za pomocą operatora <code>delete</code>.</p> <p>Nie wywołuj więcej niż tylko jednokrotnie operatora <code>delete</code> względem adresu w pamięci.</p>

TAK	NIE
<p>Przed użyciem wskaźnika sprawdź, czy nie ma przypisanej wartości null. Tego rodzaju sprawdzenie gwarantuje, że wskaźniki nieposiadające przypisanych poprawnych adresów w poleceniach znajdujących się po ich deklaracji (gdzie zostały zainicjalizowane wraz z wartościami null) nie będą mogły być użyte (przykład takiego rozwiązania pokazano w listingu 8.13).</p> <p>Upewnij się o zaprogramowaniu wskaźników w taki sposób, że będą używane jedynie po potwierdzeniu ich poprawności. W przeciwnym przypadku może dojść do awarii aplikacji.</p> <p>Pamiętaj o użyciu operatora delete w celu zwolnienia pamięci zaalokowanej przez operator new. W przeciwnym razie aplikacja będzie miała wyciek pamięci, który może doprowadzić do zmniejszenia wydajności działania systemu.</p>	<p>Nie twórz wycieku pamięci w aplikacji na skutek zapomnienia o konieczności użycia operatora delete po zakończeniu pracy z zaalokowanym blokiem pamięci.</p>

Po zapoznaniu się z najlepszymi praktykami podczas pracy ze wskaźnikami nadeszła doskonała chwila na poprawienie wyjątkowo nieudanego kodu przedstawionego w listingu 8.13. Poprawiona wersja kodu znajduje się w listingu 8.14.

Listing 8.14. Bezpieczniejsze programowanie wskaźników, poprawiona wersja listingu 8.13

```

0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     cout << "Czy jest pogodnie (t/n)? ";
6:     char userInput = 't';
7:     cin >> userInput;
8:
9:     if (userInput == 't')
10:    {
11:        // Zainicjalizowany wskaźnik (to dobrze).
12:        int* pTemperature = new int;
13:        *pTemperature = 30;
14:
15:        cout << "Temperatura wynosi: " << *pTemperature << endl;

```

```
16:
17:         // Użycie operatora delete po zakończeniu pracy ze wskaźnikiem.
18:         delete pTemperature;
19:     }
20:
21:     return 0;
22: }
```

Wynik ▼

Czy jest pogodnie (t/n)? t
Temperatura wynosi: 30

Następne uruchomienie programu:

Czy jest pogodnie (t/n)? n

(Zakończenie działania bez awarii).

Analiza ▼

Podstawowa różnica polega na tym, że wskaźnik jest tworzony wtedy, gdy naprawdę jest potrzebny, czyli po naciśnięciu klawisza *t* przez użytkownika, i zainicjalizowany w chwili tworzenia (patrz wiersz 12.). Zwolnienie pamięci następuje w tym samym bloku, a tym samym nie występuje sytuacja, w której wskaźnik został użyty (w odwołaniu lub w wywołaniu `delete`), kiedy nie miał przypisanego prawidłowego adresu w pamięci.

Sprawdzenie, czy żądanie alokacji zakończyło się powodzeniem

Działanie operatora `new` kończy się powodzeniem, o ile użytkownik nie zażąda alokacji ogromnej ilości pamięci lub system nie znajduje się w stanie krytycznym i ma niewielką ilość dostępnych zasobów. Istnieją aplikacje żądające dużych ilości pamięci (np. bazy danych) i dlatego ważne jest, aby nie przyjmować założenia, że alokacja pamięci na pewno zakończy się niepowodzeniem. Język C++ oferuje dwie metody zagwarantowania poprawności wskaźnika. Metoda domyślna używa wyjątków, zakończona niepowodzeniem operacja alokacji spowoduje zgłoszenie wyjątku w postaci obiektu `std::bad_alloc`. Wynikiem zgłoszenia tego wyjątku jest przerwanie działania aplikacji, tak więc jeśli nie przygotowałeś *procedury obsługi wyjątków*, aplikacja zakończy działanie wraz z komunikatem błędu informującym o nieobsłużonym wyjątku.

W lekcji 28., zatytułowanej „Obsługa wyjątków”, temat związany z obsługą wyjątków zostanie dokładnie omówiony. W listingu 8.15 znajdziesz przykład rozwiązania, w którym procedura obsługi wyjątków jest używana do sprawdzenia, czy alokacja pamięci zakończyła się niepowodzeniem.

Listing 8.15. Obsługa wyjątków, eleganckie zakończenie pracy, gdy wywołanie operatora `new` zakończy się niepowodzeniem

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     try
6:     {
7:         // Żądanie alokacji dużej ilości pamięci.
8:         int* pAge = new int [536870911];
9:
10:        // Użycie zaalokowanej pamięci.
11:
12:        delete[] pAge;
13:    }
14:    catch (bad_alloc)
15:    {
16:        cout << "Alokacja pamięci zakończona niepowodzeniem. Zamknięcie
17:        ↪ programu" << endl;
18:    }
19:    return 0;
20: }
```

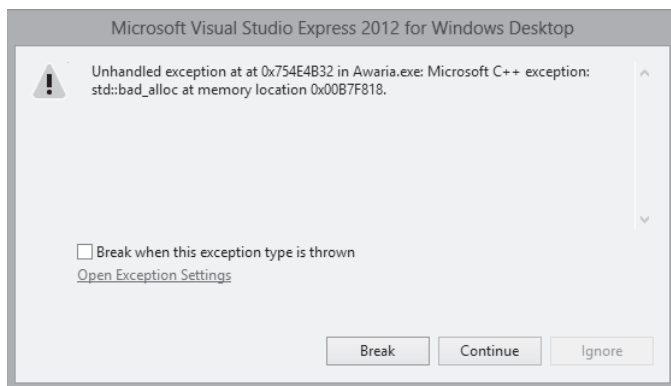
Wynik ▼

Alokacja pamięci zakończona niepowodzeniem. Zamknięcie programu

Analiza ▼

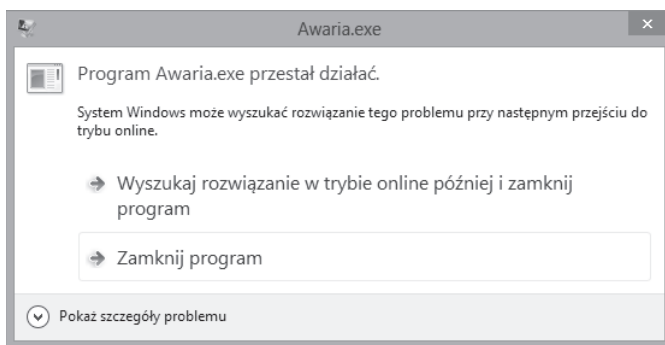
Program ten może być wykonany na różne sposoby w konkretnych komputerach. Moje środowisko pracy nie pozwala na spełnienie żądania alokacji pamięci dla 536870911 liczb całkowitych. Gdyby nie przygotowana procedura obsługi wyjątków (blok `catch` w wierszach od 14. do 17.), program zakończyłby niespodziewanie działanie. Uruchomienie programu w trybie debugowania w Visual Studio powoduje wygenerowanie danych wyjściowych pokazanych na rysunku 8.2.

RYSUNEK 8.2.
Awaria programu na skutek braku w listingu 8.15 procedury obsługi wyjątku (program uruchomiony w trybie debugowania w kompilatorze MSVC)



Tryb debugowania powoduje wstawienie do środowiska programistycznego procedur obsługi wyjątków, a skutkiem ich działania jest komunikat pokazany na rysunku 8.2. Z kolei po uruchomieniu programu w trybie Release system operacyjny (tutaj Windows) powoduje zakończenie działania aplikacji w sposób pokazany na rysunku 8.3.

RYSUNEK 8.3.
Awaria programu na skutek braku w listingu 8.15 procedury obsługi wyjątku (program uruchomiony w trybie Release)



Kiedy aplikacja ulega awarii w sposób pokazany na rysunku 8.3, jej działanie zostaje zakończone przez system operacyjny. W takim przypadku brak zaimplementowanej procedury obsługi wyjątków uniemożliwia nawet wyświetlenie jakiegoś sensownego komunikatu użytkownikowi.

Implementacja w kodzie programu procedury obsługi wyjątków daje możliwość eleganckiego zakończenia pracy aplikacji i poinformowania użytkownika o wystąpieniu problemu, zamiast pozwolenia systemowi operacyjnemu na wyświetlenie komunikatu o awarii programu.

Istnieje wariant operatora `new` o nazwie `new(nothrow)`, który w przypadku nieudanej alokacji, zamiast zgłosić wyjątek, powoduje zwrócenie wskaźnikowi wartości `null`. Przed użyciem wskaźnika można sprawdzić jego poprawność. Wersja programu wraz z operatorem `new(nothrow)` została przedstawiona w listingu 8.16.

Listing 8.16. Użycie operatora `new(nothrow)`, który zwraca wartość `null`, gdy alokacja zakończy się niepowodzeniem

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     // Żądanie alokacji dużej ilości pamięci, wersja, która nie zgłasza wyjątku.
6:     int* pAge = new(nothrow) int [0xffffffff];
7:
8:     if (pAge) // Sprawdzenie pAge != NULL.
9:     {
10:        // Użycie zaalokowanej pamięci.
11:        delete[] pAge;
12:    }
13:    else
14:        cout << "Alokacja pamięci zakończona niepowodzeniem. Zamknięcie
        ↪ programu" << endl;
15:
16:    return 0;
17: }
```

Wynik ▼

Alokacja pamięci zakończona niepowodzeniem. Zamknięcie programu

Analiza ▼

Program jest taki sam jak w poprzednim przykładzie (patrz listing 8.15), ale używa operatora `new(nothrow)`, który w przypadku nieudanej alokacji pamięci zwraca wartość `null`, zamiast zgłosić wyjątek `std::bad_alloc`. Obie przedstawione formy są dobre, wybór odpowiedniej należy do Ciebie.

Czym jest referencja?

Referencja jest aliasem dla zmiennej. Po zadeklarowaniu referencji konieczne jest jej zainicjalizowanie ze zmienną. Dlatego też zmienna referencji to po prostu inny sposób uzyskania dostępu do danych we wskazanej zmiennej. Referencję deklaruje się przy użyciu operatora referencji (&), jak to przedstawiono w poniższym fragmencie kodu:

```
TypZmiennej NazwaZmiennej = Wartość;  
TypZmiennej& ZmiennaReferencyjna = NazwaZmiennej;
```

Aby dokładnie zrozumieć, jak zadeklarować referencje i używać ich w kodzie, zapoznaj się z listingiem 8.17.

Listing 8.17. Pokazanie, że referencje są aliasami dla przypisanych wartości

```
0: #include <iostream>  
1: using namespace std;  
2:  
3: int main()  
4: {  
5:     int Original = 30;  
6:     cout << "Zmienna Original = " << Original << endl;  
7:     cout << "Zmienna Original jest pod adresem: " << hex << &Original <<  
    << endl;  
8:  
9:     int& Ref = Original;  
10:    cout << "Ref jest pod adresem: " << hex << &Ref << endl;  
11:  
12:    int& Ref2 = Ref;  
13:    cout << "Ref2 jest pod adresem: " << hex << &Ref2 << endl;  
14:    cout << "Ref2 pobiera wartość, Ref2 = " << dec << Ref2 << endl;  
15:  
16:    return 0;  
17: }
```

Wynik ▼

```
Zmienna Original = 30  
Zmienna Original jest pod adresem: 0044FB5C  
Ref jest pod adresem: 0044FB5C  
Ref2 jest pod adresem: 0044FB5C  
Ref2 pobiera wartość, Ref2 = 30
```

Analiza ▼

Dane wyjściowe programu pokazują, że referencja, niezależnie od tego, czy została zainicjalizowana z oryginalną zmienną (patrz wiersz 9.), czy do innej referencji (patrz wiersz 12.), zawsze odwołuje się do tego samego położenia w pamięci, w którym początkowo się znajdowała. Dlatego też referencje są prawdziwymi aliasami, po prostu innymi nazwami zmiennej `Original`. Wyświetlona w wierszu 14. wartość `Ref2` jest taka sama jak zmiennej `Original` (wyświetlona w wierszu 6.), ponieważ `Ref2` jest aliasem do `Original` i prowadzi do tego samego miejsca w pamięci.

Dlaczego referencje są użyteczne?

Referencje pozwalają na pracę z miejscem w pamięci, z którym zostały zainicjalizowane. Z tego powodu referencje są szczególnie użyteczne podczas programowania funkcji. Jak się dowiedziałeś w lekcji 7., zatytułowanej „Funkcje”, typowa funkcja jest deklarowana w przedstawiony poniżej sposób:

```
TypZwrotny DowolnaOperacja(Typ Parametr);  
Funkcja DowolnaOperacja() jest wywoływana następująco:  
TypZwrotny Wynik = DowolnaOperacja(argument); // Wywołanie funkcji.
```

Powyższe polecenie powoduje skopiowanie argumentu do Parametru, który następnie będzie użyty przez funkcję `DowolnaOperacja()`. Krok kopiowania może być całkiem sporym obciążeniem, jeśli dany argument zużywa dużą ilość pamięci. Podobnie gdy funkcja `DowolnaOperacja()` zwraca wartość, jest ona kopiowana z powrotem do zmiennej `Wynik`. Byłoby idealnie, gdyby można było pominąć krok kopiowania i umożliwić funkcji bezpośrednią pracę z danymi na stosie programu wywołującego. Dzięki referencjom takie rozwiązanie jest możliwe.

Wersja funkcji bez kroku kopiowania przedstawia się następująco:

```
TypZwrotny DowolnaOperacja(Typ& Parametr); // Zwróć uwagę na referencję &.
```

Funkcję tę można wywołać w poniższy sposób:

```
TypZwrotny Wynik = DowolnaOperacja(argument);
```

Ponieważ argument jest przekazywany przez referencję, `Parametr` nie jest kopią argumentu, a raczej aliasem do niego, podobnie jak w przypadku `Ref` w listingu 8.17. Ponadto funkcja akceptująca parametr jako referencję może opcjonalnie zwracać wartość, używając parametru referencji. Zapoznaj się

z listingiem 8.18, aby przekonać się, jak funkcja może używać referencji zamiast wartości zwrótej.

Listing 8.18. Funkcja obliczająca kwadrat zwrócony później w parametrze przez referencję

```
0: #include <iostream>
1: using namespace std;
2:
3: void ReturnSquare(int& Number)
4: {
5:     Number *= Number;
6: }
7:
8:
9: int main()
10: {
11:     cout << "Podaj liczbę, którą chcesz podnieść do kwadratu: ";
12:     int Number = 0;
13:     cin >> Number;
14:
15:     ReturnSquare(Number);
16:     cout << "Kwadrat podanej liczby wynosi: " << Number << endl;
17:
18:     return 0;
19: }
```

Wynik ▼

Podaj liczbę, którą chcesz podnieść do kwadratu: 5
Kwadrat podanej liczby wynosi: 25

Analiza ▼

Funkcja obliczająca kwadrat podanej liczby znajduje się w wierszach od 3. do 6. Zwróć uwagę, że dane wejściowe akceptuje w postaci parametru przekazywanego przez referencję, a wynik zwraca w ten sam sposób. Jeśli zapomnisz oznaczyć parametr `Number` jako referencję (&), wynik nie zostanie przekazany wywołującej funkcji `main()`, ponieważ funkcja `ReturnSquare()` będzie przeprowadzała operację na lokalnej kopii `Number` niszczonej w chwili zakończenia działania funkcji. Dzięki użyciu referencji masz pewność, że funkcja `ReturnSquare()` działa względem tego samego adresu w pamięci, który zaalokowano dla `Number` w funkcji `main()`. Wynik operacji jest więc dostępny w funkcji `main()` nawet po zakończeniu działania funkcji `ReturnSquare()`.

W omówionym programie został zmodyfikowany parametr danych wejściowych zawierający liczbę podaną przez użytkownika. Jeżeli potrzebujesz obu wartości, czyli liczby początkowej i podniesionej do kwadratu, konieczne jest przygotowanie funkcji akceptującej dwa parametry: z liczbą początkową oraz z liczbą podniesioną do kwadratu.

Użycie słowa kluczowego `const` w referencjach

Być może wystąpi konieczność posiadania referencji, jaka nie będzie mogła zmienić wartości początkowej zmiennej, której jest aliasem. Użycie słowa kluczowego `const` podczas deklarowania tego rodzaju referencji to jedno z możliwych rozwiązań:

```
int Original = 30;
const int& ConstRef = Original;
ConstRef = 40; //Niedozwolone: ConstRef nie może zmienić wartości Original.
int& Ref2 = ConstRef; //Niedozwolone: Ref2 nie jest stałą.
const int& ConstRef2 = ConstRef; //OK.
```

Przekazywanie funkcji argumentów przez referencję

Jedną z największych zalet referencji jest to, że umożliwia wywoływanej funkcji pracę z parametrami, które nie muszą być kopiowane z funkcji wywołującej, co znacznie poprawia wydajność działania wywoływanej funkcji. Jednak wywoływana funkcja działa, używając parametrów bezpośrednio znajdujących się na stosie funkcji wywołującej. Bardzo często ważne jest zagwarantowanie, że wywołana funkcja nie będzie mogła zmienić wartości zmiennej w funkcji wywołującej. Referencje zdefiniowane z użyciem słowa kluczowego `const` doskonale się przydadzą w takich sytuacjach, co zostało zademonstrowane w listingu 8.19. Parametr referencji `const` nie może być używany jako l-wartość, więc próba przypisania mu wartości spowoduje wygenerowanie błędu w trakcie kompilacji.

Listing 8.19. Użycie referencji `const` w celu zagwarantowania, że wywołująca funkcja nie będzie mogła zmodyfikować wartości podanej przez referencję

```
0: #include <iostream>
1: using namespace std;
2:
```

```
3: void CalculateSquare(const int& Number, int& Result) //Zwróć uwagę na "const".
4: {
5:     Result = Number*Number;
6: }
7:
8: int main()
9: {
10:     cout << "Podaj liczbę, którą chcesz podnieść do kwadratu: ";
11:     int Number = 0;
12:     cin >> Number;
13:
14:     int Square = 0;
15:     CalculateSquare(Number, Square);
16:     cout << Number << "^2 = " << Square << endl;
17:
18:     return 0;
19: }
```

Wynik ▼

Podaj liczbę, którą chcesz podnieść do kwadratu: 27
 $27^2 = 729$

Analiza ▼

W przeciwieństwie do poprzedniego programu, w którym zmienna zawierająca liczbę początkową przechowywała także wynik działania funkcji, powyższy program używa dwóch zmiennych: jednej przeznaczonej dla liczby początkowej podanej przez użytkownika i drugiej dla wyniku przeprowadzonej operacji. Aby zagwarantować, że liczba podana przez użytkownika nie będzie mogła być zmieniona, użyto referencji `const` zdefiniowanej za pomocą słowa kluczowego `const` (patrz wiersz 3.). W ten sposób parametr `Number` jest automatycznie parametrem danych wejściowych, a jego wartość nie może być zmodyfikowana.

Jako eksperyment możesz wprowadzić modyfikację w wierszu 5., aby zwrócić wynik z użyciem logiki takiej samej jak w listingu 8.18:

```
Number *= Number;
```

Bez wątpliwości w trakcie kompilacji nastąpi wygenerowanie błędu informującego o braku możliwości zmiany wartości `const`. Dlatego też referencja `const` to użyteczne narzędzie udostępniane przez język C++ wskazujące, że dany parametr jest parametrem danych wejściowych. Jednocześnie masz gwarancję, że wartość przekazywana przez referencję nie będzie mogła być modyfikowana przez

wywołaną funkcję. To może wydawać się oczywiste, ale w środowisku pracy obejmującym wielu programistów, gdzie jedna osoba tworzy pierwszą wersję funkcji, a inna poprawia ją i usprawnia, użycie referencji `const` wprowadza dużą różnicę w jakości tworzonego kodu.

Podsumowanie

W tej lekcji poznałeś wskaźniki i referencje. Dowiedziałeś się, jak wskaźniki mogą być wykorzystywane do uzyskania dostępu i operowania pamięcią oraz jakie to przydatne narzędzie pomagające w trakcie dynamicznej alokacji pamięci. Poznałeś operatory `new` i `delete` używane do alokacji pamięci dla elementu. Ponadto poznałeś ich odpowiedniki `new[...]` i `delete[]` pomagające w alokacji pamięci dla tablicy danych. W lekcji przedstawiono również pułapki czyhające na programistów w trakcie pracy ze wskaźnikami i dynamiczną alokacją pamięci. Przekonałeś się też, jak ważne jest zwalnianie dynamicznie alokowanej pamięci, aby uniknąć tzw. wycieków pamięci. Referencje są aliasami i jednocześnie oferującą potężne możliwości alternatywą dla używania wskaźników podczas przekazywania argumentów funkcjom — referencje gwarantują zachowanie ich poprawności. Dowiedziałeś się, jak we właściwy sposób stosować `const` podczas pracy ze wskaźnikami i referencjami oraz jak deklarować funkcje na najwyższym możliwym poziomie zachowania stałości parametrów.

Pytania i odpowiedzi

Pytanie: Dlaczego stosować dynamiczną alokację pamięci, skoro mogę zdefiniować tablicę statyczną i nie przejmować się zwalnianiem pamięci?

Odpowiedź: Tablice statyczne mają stałą wielkość i nigdy nie zapewniają możliwości skalowania w górę, jeśli aplikacja będzie wymagała większej ilości pamięci, ani nie optymalizują użycia pamięci, gdy aplikacja korzysta z mniejszej ilości danych. W takich sytuacjach stosowanie dynamicznej alokacji pamięci powoduje istotną różnicę.

Pytanie: Mam dwa przedstawione poniżej wskaźniki:

```
int* pNumber = new int;  
int* pCopy = pNumber;
```

Czy dobrym rozwiązaniem będzie wywołanie operatora `delete` względem obu, aby zagwarantować całkowite zwolnienie pamięci?

Odpowiedź: To jest błędne rozwiązanie. Można tylko jednokrotnie wywołać operator `delete` względem danego adresu w pamięci zwróconego przez operator `new`. Ponadto warto unikać sytuacji, w której dwa wskaźniki prowadzą do tego samego adresu w pamięci, ponieważ wywołanie operatora `delete` spowoduje unieważnienie obu wskaźników. Nie powinieneś także tworzyć programu w taki sposób, że nie masz pewności dotyczącej poprawności używanych wskaźników.

Pytanie: Kiedy powinienem używać operatora `new(nothrow)`?

Odpowiedź: Jeśli nie chcesz zajmować się obsługą wyjątku `std::bad_alloc`, wtedy możesz użyć wersji `new(nothrow)` operatora `new`, ponieważ w przypadku nieudanej operacji alokacji pamięci zwraca on wartość `null`.

Pytanie: Potrzebuję wywołać funkcję obliczającą pole. Do dyspozycji mam dwie przedstawione poniżej metody:

```
void CalculateArea (const double* const pRadius, double* const pArea);
void CalculateArea (const double& radius, double& area);
```

Który z powyższych wariantów powinienem wybrać?

Odpowiedź: Drugi z wymienionych, użycie referencji jako referencji nie jest błędem, podczas gdy w przypadku wskaźników już tak. Ponadto druga metoda jest prostsza.

Pytanie: Mam dwa wskaźniki:

```
int Number = 30;
const int* pNumber = &Number;
```

Rozumiemwzględem na deklarację `const`. Czy mogę przypisać `pNumber` do wskaźnika typu innego niż `const`, a następnie użyć go do modyfikacji wartości znajdującej się w `Number`?

Odpowiedź: Nie, nie ma możliwości zmiany wskaźnika `const`:

```
int* pAnother = pNumber; // Wskaźnika const nie można przypisać do nonconst.
```

Pytanie: Dlaczego mam trudzić się przekazywaniem funkcji wartości przez referencję?

Odpowiedź: Nie musisz tego robić, o ile takie rozwiązanie nie ma dużego wpływu na kod. Jeśli jednak parametry funkcji akceptują obiekty, które mogą być całkiem duże (wielkość wyrażona w bajtach), ich przekazanie przez wartość stanie się naprawdę kosztowną operacją. Wywołanie funkcji może być znacznie efektywniejsze po użyciu referencji. Pamiętaj o częstym użyciu `const`, z wyjątkiem funkcji, które muszą w zmiennej przechowywać wynik działania.

Pytanie: Jaka jest różnica pomiędzy dwoma poniższymi deklaracjami?

```
int MyNumbers[100];  
int* MyArrays[100];
```

Odpowiedź: `MyNumbers` to tablica liczb całkowitych, tzn. `MyNumbers` jest wskaźnikiem do miejsca w pamięci przechowującego sto liczb całkowitych i wskazuje pierwszą z nich (indeks 0). To jest statyczna alternatywa dla poniższego fragmentu kodu:

```
int* MyNumbers = new int [100]; // Tablica zaalokowana dynamicznie.  
// Użycie MyNumbers.  
delete MyNumbers;
```

Z drugiej strony, `MyArrays` to tablica stu liczb całkowitych, każdy wskaźnik może prowadzić do liczby całkowitej lub tablicy liczb całkowitych.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Dlaczego nie można przypisać referencji `const` do referencji innej niż `const`?
2. Czy `new` i `delete` to funkcje?
3. Jaka jest natura wartości przechowywanej w zmiennej wskaźnika?
4. Jakiego operatora użyjesz w celu uzyskania dostępu do danych wskazywanych przez wskaźnik?

Ćwiczenia

1. Co zostanie wyświetlone po wykonaniu przedstawionych poniżej poleceń?

```
0: int Number = 3;
1: int* pNum1 = &Number;
2: *pNum1 = 20;
3: int* pNum2 = pNum1;
4: Number *= 2;
5: cout << *pNum2;
```

2. Jakie są podobieństwa i różnice pomiędzy przedstawionymi poniżej trzema przeciążonymi funkcjami?

```
int DoSomething(int Num1, int Num2);
int DoSomething(int& Num1, int& Num2);
int DoSomething(int* pNum1, int* pNum2);
```

3. Jak zmienisz deklarację pNum1 w ćwiczeniu 1. (wiersz 1.), aby przypisanie w wierszu 3. stało się nieprawidłowe? (Podpowiedź: musisz upewnić się, że pNum1 nie może zmienić wskazywanych danych).

4. **Łowcy błędów:** Co jest nie tak z poniższym fragmentem kodu?

```
#include <iostream>
using namespace std;
int main()
{
    int *pNumber = new int;
    pNumber = 9;
    cout << "Wartość wskaźnika pNumber: " << *pNumber;
    delete pNumber;
    return 0;
}
```

5. **Łowcy błędów:** Co jest nie tak z poniższym fragmentem kodu?

```
#include <iostream>
using namespace std;
int main()
{
    int pNumber = new int;
    int* pNumberCopy = pNumber;
    *pNumberCopy = 30;
    cout << *pNumber;
    delete pNumberCopy;
    delete pNumber;
    return 0;
}
```

6. Jakie będą dane wyjściowe powyższego programu po jego poprawieniu?

Skorowidz

A

- ABC, Abstract Base Class, 358
- abstrakcja danych, 259
- abstrakcyjna klasa bazowa, 355, 358, 366
- adaptery, 469
- adres zmiennej, 209, 210
- algorytm
 - binary_search, 663
 - count_if, 643
 - find, 471, 643
 - find_if, 471, 622
 - for_each, 603, 652
 - generate, 650
 - partition, 666
 - remove_if, 471
 - reverse, 471, 494
 - search_n, 645
 - sort, 663
 - stable_partition, 667
 - transform, 471, 494, 611, 655
 - unique, 663
- algorytmy
 - inicjalizacyjne, 638
 - kopiujące, 638
 - modyfikujące, 638
 - niezmiennne, 636, 637
 - partycjonujące, 639
 - porównania, 637
 - sortujące, 639
 - std, 471, 494, 603, 643, 666
 - STL, 471, 614, 635
 - usuujące, 638
 - zastępujące, 639
 - zmiennne, 638–640
- alias, 241
- alokacja
 - bufora, 280
 - dynamiczna, 223
 - pamięci, 218–221, 224, 237, 245
 - pamięci dla tablicy elementów, 222

- analiza
 - błędów, 33
 - operatorów logicznych, 122
 - znaku null, 100
- aplikacje
 - jednowątkowe, 771
 - wielowątkowe, 772
- argumenty funkcji, 182
- ASCII, 65, 839
- asercja w trakcie kompilacji, 456
- awaria programu, 278

B

- bajt, 786
- bezpieczeństwo typów, 419, 448
- biblioteka STL, 200, 404, 457, 461–595
- biblioteki sprytnych wskaźników, 722
- binarny predykat sortowania, 538
- bit, 785
 - najbardziej znaczący, 66
 - najmniej znaczący, 66
 - znaku, 66
- blok, 111
 - catch, 753
 - try, 753
- błąd
 - kompilacji, 37, 71, 226, 360, 448, 526
 - nieprawidłowej alokacji pamięci, 755
 - słupka w płocie, 93
 - syntaktyczny, 39
 - typu Access Violation, 752
 - typu przepełnienie bufora, 104

C

- C++11, 30, 31, 37, 201
- ciąg Fibonacciego, 173, 188
- ciągi tekstowe, 99, 480
- COW, Copy on Write, 716
- czas wyszukiwania elementu, 562

D

debugowanie, 237, 755
definicja
 funkcji, 182
 klonu funkcji, 364
 wyrażenia lambda, 619
definiowanie
 funkcji, 438
 kolejki, 515
 stałych, 434
deklarowanie
 destruktora, 272
 dynamicznej tablicy, 71
 funkcji, 49
 iteratora, 474, 493, 594
 klasy, 252
 konstruktora, 261
 konstruktora kopiującego, 279
 konstruktora przenoszącego, 407
 operatora, 372
 sprytnego wskaźnika, 382
 stałych, 74, 78, 81
 superklasy, 308
 tablic, 87, 90
 tablic wielowymiarowych, 95
 using namespace, 46
 wskaźnika, 208, 211
 wzorca, 445, 496
 zmiennych, 56, 59, 210
dekrementacja wskaźnika, 222
dereferencja, 213
destruktor, 272–275, 281
 klasy bazowej, 351
 prywatny, 290
 pusty, 275
 wirtualny, 346, 349, 366
długość ciągu tekstowego, 102–105
domyślny predykat sortowania, 551, 583
dostęp do
 atrybutów, 258
 atrybutów klasy bazowej, 316
 danych, 213
 danych prywatnych, 298, 299
 elementów
 obiektu vector, 508, 510
 składowych, 254, 301
 tablicy, 71, 90, 93, 229, 509
 tablicy wielowymiarowej, 95
 konstruktora, 289
 obiektu, 344
 obiektu string, 484, 485
 pamięci, 219
dynamiczna alokacja pamięci, 218, 245

dynamiczne

 alokowanie egzemplarza klasy, 380
 alokowanie buforów, 280
 klasy tablic, 499
dyrektywa preprocesora, 42
 #define, 78, 434, 438
 #endif, 438
 #ifdef, 438
 #include, 42, 53
działanie
 obsługi wyjątków, 758
 operatora new, 219, 237
dziedziczenie, 305, 308
 chronione, 330–333, 337
 prywatne, 327, 333
 publiczne, 307, 311
 wielokrotne, 308, 334, 358

E

edytor tekstu, 34
egzemplarz klasy, 300
egzemplarz klasy bazowej, 361
elementy składowe, 257–260
elementy składowe statyczne, 454

F

FIFO, First-In-First-Out, 676
flaga boolowska, 311
funkcja
 append(), 487
 Area(), 181
 back(), 684
 c_str(), 484
 cbegin(), 528
 cend(), 528
 Circumference(), 181
 Clone(), 364
 copy(), 657
 copy_backward(), 658, 673
 copy_if(), 658
 count(), 553, 558, 654, 699
 count_if(), 645
 DisplayComparison(), 448
 DisplayContents(), 528, 553
 DisplayNums(), 202
 DisplayVector(), 507, 513
 empty(), 684, 689
 end(), 556
 erase(), 490, 531, 556, 580
 fill(), 648

fill_n(), 648
 find(), 488, 554, 561–564, 577, 641
 find_if(), 642
 flip(), 699, 704
 for_each(), 652, 654
 front(), 684, 691
 FuncDisplayElement(), 601
 generate(), 650
 generate_n(), 650
 getline(), 738
 GetMax(), 446
 GetPi(), 199
 HashFunction(), 588
 insert(), 504, 529, 573
 load_factor(), 563, 589
 lower_bound(), 669
 main(), 43, 49, 180
 max_bucket_count(), 563, 589
 max_load_factor(), 563, 565
 operator(), 404, 601–611
 plus(), 657
 pop(), 679, 684, 689
 pop_back(), 511
 pop_front(), 516, 518
 push(), 679, 684, 689
 push_back(), 98, 503, 518, 527, 704
 push_front(), 516, 527, 542
 remove(), 543, 658
 remove_if(), 658, 660
 replace(), 661
 replace_if(), 661
 reserve(), 513, 519
 reset(), 699
 reverse(), 534
 search(), 646
 search_n(), 645, 648
 set(), 699
 size(), 99, 504, 512, 563, 684, 699
 sizeof(), 100
 sort(), 534
 SortPredicate_Descending(), 537
 strcat(), 102
 strcpy(), 102, 275
 strlen(), 101, 275
 tolower(), 657
 top(), 680, 689
 transform(), 655
 upper_bound(), 669
 funkcje, 48, 179
 argumenty, 182
 bez parametrów, 185
 bez wartości zwrotnej, 185
 czysto wirtualne, 355

definicja, 182
 dwuargumentowe, 600, 608
 jednoargumentowe, 600
 klasy
 bitset, 699
 priority_queue, 688
 queue, 683
 stack, 679
 kopiowania, 657
 lambda, 200
 prototyp, 181
 przeciążanie, 192, 203
 przekazywanie argumentów, 243
 rekurencyjne, 188, 189
 tablica wartości, 194
 typu inline, 198
 używanie referencji, 242
 wirtualne, 344, 351–354, 367
 wywołanie, 182
 wzorca, 446
 z wieloma parametrami, 183
 z wieloma poleceniami return, 190
 funktor, 587, 599

H

hermetyzacja, 371
 hierarchia
 dziedziczenia, 308, 337, 362, 429
 dziedziczenia prywatnego, 331

I

IDE, Integrated Development Environment, 33
 identyfikacja
 typu, 422, 424
 typu w czasie działania, 421
 implementacja
 bezpieczeństwa wyjątków, 753
 destruktora, 272
 funkcji, 182
 klasy STL string, 496
 konstruktora, 261
 operatora
 dereferencji, 382
 dodawania, 389
 dwuargumentowego, 388
 indeksowania, 401
 konwersji, 378
 porównania, 394
 wyboru, 382

- implementacja
 - sprytnych wskaźników, 711
 - tabeli hash, 589
 - indeksy tablicy, 88
 - informacja
 - o stanie, 604, 623
 - o znaku, 70
 - inicjalizacja
 - ciągu tekstowego, 482
 - elementów, 650
 - klasy bazowej, 314
 - kolekcji, 650
 - listy, 525
 - tablicy, 87, 105
 - tablic wielowymiarowych, 95
 - wskaźnika, 208, 211
 - zmiennej, 59, 82
 - zmiennych składowych klasy, 262
 - inkrementacja
 - iteratora, 473
 - wskaźnika, 222, 223
 - instrukcja CALL, 198
 - iteracja tablic wielowymiarowych, 171
 - iterador, 594
 - danych wejściowych, 470
 - danych wyjściowych, 470
 - dwukierunkowy, 471
 - poruszający się tylko do przodu, 470
 - STL, 470
 - swobodnego dostępu, 471
- J**
- język C++, 30
- K**
- klasa, 252, 449
 - auto_ptr, 718
 - basic_string<T>, 475
 - bitset, 127, 696
 - CompareStringNoCase, 631
 - Date, 380
 - deque, 515
 - exception, 761
 - forward_list, 523, 542
 - fstream, 739, 748
 - hash_set, 563
 - ifstream, 746, 748
 - list, 524
 - map, 569, 571, 587
 - multimap, 571
 - multiset, 547
 - ofstream, 745, 748
 - priority_queue, 686
 - queue, 681
 - set, 547
 - smart_pointer, 713
 - sprytnego wskaźnika, 383, 384
 - stack, 677, 679
 - string, 102, 475–486, 496
 - stringstream, 746, 749
 - thread, 772
 - typu Singleton, 287
 - unique_ptr, 720
 - unordered_multiset, 562
 - unordered_set, 563
 - vector, 97, 202, 464, 500
 - tworzenie egzemplarzy, 501
 - wstawianie elementów, 503, 504
 - vector<bool>, 702
 - wstring, 475, 481, 496
 - klasy
 - bazowe, 306–309, 314
 - bazowe abstrakcyjne, 356
 - potomne, 306–309, 321, 338
 - STL, 523, 542, 569
 - strumieni, 729
 - tablic, 499
 - wzorca, 449, 452, 454, 456
 - zaprzyjaźnione, 298, 300
 - kod spaghetti, 157
 - kody ASCII, 839
 - kolejka, 676, 681
 - FIFO, 470
 - LIFO, 470, 676
 - priorytetowa, 686, 689
 - kolejność
 - niszczenia obiektów, 759
 - sortowania, 563
 - tworzenia obiektów, 761
 - tworzenia zmiennych, 326
 - użycia destruktorów, 324
 - użycia konstruktorów, 324
 - wykonywania operacji, 773
 - kolekcja posortowana, 669
 - komentarze, 47, 53
 - kompilacja, 32, 35
 - kompilacja klasy, 255
 - kompilator, 37, 434, 780
 - kompilator g++, 36
 - komunikat
 - błędu, 329, 456
 - ostrzeżenia, 38
 - koniec wiersza, 44

konstrukcja
 if-else, 126, 140–142, 147, 154
 if-else-if, 149, 152
 switch-case, 149, 150, 152

konstruktor, 261
 domyślny, 267, 271, 301
 klasy pochodnej, 314
 kopiujący, 276, 279–285, 301, 363, 408
 przeciążony, 264, 267, 269
 przenoszący, 284, 406, 408
 z listami inicjalizacyjnymi, 270

kontenery
 adaptacyjne, 675
 asocjacyjne
 std::map, 465
 std::multiset, 466
 std::multimap, 466
 std::set, 465
 std::unordered_set, 465

sekwencyjne
 std::deque, 464
 std::forward_list, 464
 std::list, 464
 std::vector, 464

specjalne
 std::priority_queue, 470
 std::queue, 470
 std::stack, 470

konwersja
 liczb, 787
 konwersja znaków, 494, 629

kopiowanie
 destrukcyjne, 718
 głębokie, 279, 714
 obiektu string, 482
 płytkie, 276, 278

kopiujący operator przypisania, 284, 398
 kryterium sortowania, 583
 kubek, bucket, 588

L

liczba
 egzemplarzy klasy bazowej, 359, 361
 elementów obiektu, 553

liczby całkowite
 bez znaku, 67
 ze znakiem, 66

licznik, 161
 licznik odniesień, 717
 LIFO, Last-In-First-Out, 470, 676
 linker, 32

lista
 dwukierunkowa, 524
 jednokierunkowa, 524, 542

listy
 inicjalizacyjne, 270, 271, 504
 najlepszych praktyk, 776
 przechwytywania, 622, 631

l-wartość, 112, 135

Ł

łączenie
 ciągów tekstowych, 373, 487
 trzech ciągów tekstowych, 406

M

makra, 433, 434
 wady, 443
 zalety, 443

makro assert(), 442

manipulator
 setfill(), 734, 735
 setw(), 734

manipulatory
 liczb zmiennoprzecinkowych, 730
 podstaw liczb, 730
 wyjścia, 730, 731

manipulowanie ciągami tekstowymi, 480

mechanizm
 kopiowania przy zapisie, 716
 licznika odniesień, 717
 RTTI, 429

metody
 czysto wirtualne, 355
 klasy bazowej, 319
 prywatne, 257
 publiczne, 257–259, 288
 wirtualne, 345
 zagwarantowania poprawności
 wskaźnika, 236

mikroprocesor, 197
 modyfikowanie tablicy, 91
 muteksy, 774

N

nadpisywanie metod, 316–321, 345

narzędzie
 Microsoft Visual C++ Express, 35
 Visual Studio Express 2012, 34

nawiasy
 klamrowe, 61
 kwadratowe, 619
 ostre, 43
 nazwy zmiennych i stałych, 72, 79
 niejednoznaczność semantyczna, 358, 362

O

obiekt, 252
 bitset, 696
 Date, 380
 deque
 funkcja pop_front(), 516
 funkcja push_back(), 518
 funkcja push_front(), 516, 518
 forward_list, 542
 list
 funkcja erase(), 531
 funkcja insert(), 529
 funkcja push_back(), 527
 funkcja push_front(), 527
 funkcja reverse(), 534
 funkcja sort(), 534, 535
 ustanawianie, 525
 map, 570
 funkcja erase(), 580
 funkcja find(), 577
 funkcja insert(), 573
 multimap, 570
 multiset, 549
 priority_queue, 686, 691
 funkcja pop(), 689
 funkcja push(), 689
 funkcja top(), 689
 predykat greater <int>, 691
 queue, 682, 685
 funkcja back(), 685
 funkcja front(), 685
 set, 549
 funkcja erase(), 556, 559
 funkcja find(), 554
 funkcja insert(), 552
 stack, 677
 std::bitset, 127
 std::sort, 202
 std::string, 273, 379, 484
 string
 algorytm std::reverse, 493
 algorytm std::transform, 494
 funkcja append(), 487
 funkcja erase(), 490
 funkcja find(), 488
 kopiowanie, 482
 ustanawianie, 482
 wyszukiwanie podciągu, 488
 wyszukiwanie znaku, 489
 stringstream, 747
 unique_ptr, 720, 721
 unordered_map, 563, 588
 unordered_multimap, 588
 vector, 501, 502
 funkcja capacity(), 513
 funkcja insert(), 504
 funkcja pop_back(), 511
 funkcja push_back(), 503, 504
 funkcja reserve(), 513
 funkcja size(), 513
 operator indeksowania, 509
 pojemność, capacity, 513
 semantyka tablicy, 508
 semantyka wskaźnika, 510
 wielkość, size, 513
 obiekty
 funkcji, 587, 599
 globalne klas strumieni, 729
 obsługa wyjątków, 237, 246, 751–766
 ochrona przed wielokrotnym dołączaniem, 437
 odczyt
 danych ze strumienia, 728
 pliku binarnego, 744
 tekstu z pliku, 742
 odwracanie
 ciągu tekstowego, 493
 elementów w obiekcie list, 534
 kolejności elementów, 673
 określanie
 adresu zmiennej, 209
 typu w trakcie działania programu, 355
 opcje bitowe, 695
 operacje
 klas map i multimap, 571
 klas set i multiset, 549
 klasy forward_list, 542
 klasy list, 524
 klasy vector, 500
 na ciągach tekstowych, 475
 na flagach bitowych, 128
 wejścia-wyjścia, 727
 operator, 792
 ?, 153
 adresu, 210
 AND, 120–126
 const_cast, 426
 dekrementacji, 113, 375
 delete, 218, 220, 231, 246, 351
 delete[], 222, 276

dereferencji, 213, 215, 219, 225, 380
 dodawania, 112, 373
 dodawania/przypisania, 389
 dodawania dwuargumentowy, 386
 dostępu pośredniego, 215
 dynamic_cast, 421, 429
 dzielenia, 112
 indeksowania, 403, 464, 509, 574
 inkrementacji, 113, 372
 jednoargumentowy dekrementacji, 375
 jednoargumentowy inkrementacji, 375
 kopiujący przypisania, 398, 408
 kropki, 254
 mnożenia, 112
 mnożenie/przypisanie, 391
 new, 218, 236
 new(nothrow), 239
 nierówności, 117, 392, 394
 NOT, 120–126
 odejmowania, 112
 odejmowania dwuargumentowy, 387, 389
 OR, 120–126
 przenoszący przypisania, 406–408
 przesunięcia, 128
 przypisania, 111, 397
 referencji, 209, 240
 reinterpret_cast, 425, 429
 reszty z dzielenia, 112
 równości, 117, 391
 sizeof, 68, 71, 80, 132, 223
 sizeof dla klasy, 293
 static_cast, 420
 tablicy, 230
 warunkowy, 153
 wskaźnika, 255
 wyboru elementu składowego, 380, 382
 wyboru zakresu, 262, 321
 wyłuskania, 213
 XOR, 120, 126

operatory, 371, 372
 arytmetyczne, 112
 bitowe, 126, 128
 bitowego przesunięcia, 128, 129
 dwuargumentowe, 384–386
 indeksowania, 400
 jednoargumentowe, 373, 374
 konwersji, 378
 logiczne, 120, 122, 125
 negacji, 126
 niezmiennicze, 413
 obsługiwane przez klasę bitset, 698, 699
 porównania, 394
 postfiksowe, 114

prefiksowe, 114
 przypisania, 130
 relacji, 118
 rzutowania, 417, 419, 427
 złożone przypisania, 130
 optymalizacja zużycia pamięci, 97
 otwieranie pliku, 740, 741

P

pamięć RAM, 56
 para klucz-wartość, 570, 588
 parametry
 funkcji, 184, 186
 konstruktora, 269
 z wartościami domyślnymi, 271
 partycjonowanie zakresu, 666
 pętla, 154
 do...while, 157, 166, 174
 for, 71, 157, 161–168, 175
 while, 157, 158, 166, 175
 pętle
 nieskończone, 165
 zagnieżdżone, 170–173
 pierwszeństwo operatorów, 133, 441
 pliki
 .cpp, 32, 437
 .obj, 32
 nagłówkowe, 43, 437
 wykonywalne, 32
 płytka kopia, 278
 pobieranie danych, 51
 pochodzenie, 306, 308
 POD, Plain Old Data, 777
 podklasa, 309, 338
 pojemność obiektu vector, 513
 polecenia
 warunkowe, 139
 zagnieżdżone if, 145
 złożone, 111
 polecenie, 110
 break, 165–167, 175
 cin, 51, 102
 continue, 165, 167, 169
 cout, 44, 51
 delete[], 225
 endl, 35, 51
 for, 161
 goto, 155, 157
 return, 155, 175, 190
 using std::cout, 47
 using std::endl, 47
 using namespace, 45

- polimorfizm, 341
- poprawienie wydajności, 284
- postfiksowe operatory inkrementacji, 377
- predykat, 600, 606
 - dwuargumentowy, 404, 600, 614
 - jednoargumentowy, 404, 606
 - sortowania, 583
- prefiksowe operatory inkrementacji, 374
- preprocesor, 42, 434
- priorytety operatorów, 792
- procedura obsługi wyjątku, 759
- procesor, 770
- procesor wielordzeniowy, 770
- programowanie
 - jednoargumentowego operatora, 374
 - operatora dereferencji, 380
 - operatora wyboru, 380
 - operatorów dodawania, 386
 - operatorów konwersji, 378
 - operatorów odejmowania, 386
 - warunkowe, 141
 - zagnieżdżonych pętli, 169
 - zorientowane obiektowo, 249
- prototyp funkcji, 181
- prywatna kopia konstruktora, 286
- prywatne elementy składowe, 257, 260, 261
- prywatny
 - konstruktor domyślny, 288
 - operator przypisania, 286
- przechowywanie
 - adresów, 210
 - danych, 102
 - liczb, 57, 116
 - wartości boolowskich, 64
 - znaków, 65
- przechwytywanie wyjątku, 755
- przeciążanie
 - funkcji, 192, 203
 - konstruktora, 264
 - kopiującego operatora przypisania, 397
 - operatorów, 371
 - operatorów porównania, 394
 - operatorów równości, 391
- przekazywanie
 - argumentów
 - przez referencję, 195, 243
 - przez wartość, 276, 278
 - obiektu funkcji, 277
 - parametrów klasie bazowej, 314
 - wskaźników funkcjom, 226, 227
- przenoszący operator przypisania, 285
- przepełnienie, 116
- przerywanie działania pętli, 166

- przestrzeń nazw std, 45, 46, 730
- przeszukiwanie kolekcji, 663
- przetwarzanie elementów w zakresie, 652
- przetwarzanie warunkowe, 123, 125, 149
- przypisanie
 - wartości atrybutowi, 258
 - wskaźnikowi adresu zmiennej, 212

R

- RAM, Random Access Memory, 56
- referencja, 207, 209, 240–243
- referencja const, 243
- rekurencja, 188
- relacja
 - klas, 335
 - typu jest-czymś, 307, 337
- rodzaje deklaracji wzorca, 446
- rozmiar zmiennej, 68
- RTTI, Run Time Type Identification, 355, 422, 429
- r-wartość, 112, 135
- rzutowanie, 418
 - dynamiczne, 422
 - jawne, 421
 - niejawne, 421
 - statyczne, 421
 - w dół, 420, 429
 - w górę, 420, 429

S

- segmentowanie, slicing, 334
- semafory, 774
- semantyka
 - tablicy, 508, 574
 - wskaźnika, 510
- skalowalność, 97
- składnia wyrażeń lambda, 624
- słowo kluczowe, 81, 789
 - auto, 70, 474, 493
 - class, 252, 293, 371
 - const, 73, 225, 403
 - constexpr, 73, 75
 - do, 161
 - enum, 73, 76
 - friend, 297, 299
 - private, 255
 - protected, 308, 311
 - inline, 199, 200
 - mutable, 625
 - new, 227
 - public, 255, 586

- static, 287
- struct, 297
- template, 445
- typedef, 72
- typename, 445
- using, 46
- virtual, 345, 360–363
- sortowanie, 583, 663
 - elementów, 562
 - elementów listy, 533, 536, 538, 541
 - kontenerów, 466
- specjalizacja, 311
- specjalizacja wzorca, 450
- specyfikator dostępu, 308, 330
- sprawdzanie
 - w trakcie kompilacji, 456
 - wyrażeń, 441
- sprytne wskaźniki, smart pointer, 380, 384, 709–725
- stałe, 72
- stałe typu wyliczeniowego, 76
- standard C++11, 201
- standardowa biblioteka wzorców, 463
- statyczne elementy składowe, 454
- sterta, 290
- STL, Standard Template Library, 404, 457
- stos, 198, 290, 676
- struktura, 297
 - DisplayElement<T>, 601, 603
 - std::less<T>, 551
- strumień, 44, 727
- strumień
 - pliku, 741
 - std::cout, 731
 - std::cin, 735
 - std::fstream, 739
 - std::stringstream, 746
- superklasa, 309
- synchronizacja wątków, 773
- system liczb
 - dwójkowych, 784
 - dziesiętnych, 784
 - szesnastkowych, 786

Ś

- ścieżka dostępu
 - do katalogu, 54
 - względna, 43

T

- tabela
 - funkcji wirtualnych, VFT, 352
 - hash, 565, 587–589
- tablica, 85, 228
 - vector, 465
 - wskaźników, 366
- tablice
 - dynamiczne, 71, 97, 104, 247, 499
 - statyczne, 87, 143, 231, 245
 - wielowymiarowe, 94
- transformacja zakresu, 654
- tryby otwierania pliku, 741
- tworzenie
 - aplikacji, 32, 34
 - egzemplarzy klasy vector, 501
 - egzemplarzy na stercie, 290
 - obiektów set, 550
 - obiektów w wolnej pamięci, 291
 - obiektu bitset, 697
 - obiektu klasy, 253
 - tablicy dynamicznej, 97
 - wątku, 772
- typ
 - bool, 64
 - char, 65
 - double, 67
 - float, 67
 - int, 66
 - long, 66
 - long long, 66
 - short, 66
 - unsigned int, 67, 83, 136
 - unsigned long, 67
 - unsigned long long, 67
 - unsigned short, 65, 116, 136
 - void, 50, 185
 - wyliczeniowy, 76, 77
- typy
 - liczb całkowitych, 66, 67
 - operatorów, 371
 - dwuargumentowych, 385
 - jednoargumentowych, 373
 - sprytnych wskaźników, 713
 - wskaźnika, 211
 - zmiennoprzecinkowe, 67
 - zmiennych, 63, 64

U

- udostępnianie atrybutów, 312
- ukrywanie metod, 321
- uniemożliwienie kopiowania obiektów, 287
- ustanawianie
 - klasy
 - bitset, 696
 - vector, 500
 - vector<bool>, 702
 - obiekту
 - list, 524
 - map, 571
 - multimap, 572
 - priority_queue, 686
 - queue, 682
 - set, 549
 - stack, 677, 678
 - string, 482
 - wzorca, 450
- usuwanie
 - elementów, 531, 560
 - listy, 532, 538
 - z kolejki, 516, 684
 - z kolejki priorytetowej, 689
 - z obiektów, 511, 556, 580
 - ze stosu, 679
 - znaków, 491, 492
- użycie
 - algorytmów STL, 640
 - algorytmu
 - count_if(), 644
 - for_each(), 619
 - partition, 667
 - stable_partition, 668
 - std::count(), 644
 - std::sort, 664
 - ciągów tekstowych, 105
 - destruktora, 273, 286, 324
 - dyrektywy #define, 434, 438
 - dziedziczenia
 - prywatnego, 358
 - wielokrotnego, 335
 - funkcji
 - capacity(), 514
 - dwuargumentowej, 609
 - erase(), 491, 557
 - fill(), 648
 - find(), 490, 555–559, 641
 - for_each(), 652
 - generate(), 650
 - insert(), 504, 574
 - klasy priority_queue, 689
 - kopiowania, 659
 - lambda, 201, 603
 - lower_bound(), 669
 - makro, 443
 - pop_back(), 511
 - push_back(e), 503
 - rekurencyjnej, 189
 - replace(), 661
 - size(), 514
 - string::find, 488
 - transform(), 655
 - typu inline, 199
 - upper_bound(), 671
 - informacji o stanie, 605
 - iteratorów, 473
 - klasy
 - bitset, 698
 - ciągu tekstowego, 480
 - priority_queue, 686
 - queue, 681
 - stack, 677
 - std::string, 102
 - unique_ptr, 720
 - vector<bool>, 703
 - konstruktora, 262, 286, 324
 - kopiującego, 279
 - przenoszącego, 406
 - makra
 - definiującego stałe, 435
 - do ochrony przed dołączaniem, 437
 - do sprawdzania wyrażeń, 441
 - obiekту
 - funkcji, 604, 612
 - list, 538–541
 - map, 592
 - set, 562
 - unique_ptr, 721
 - unordered_map, 592
 - operatorów, 111
 - const_cast, 426
 - dereferencji, 213, 214
 - dodawania/przypisania, 389
 - dynamic_cast, 421
 - indeksowania, 403
 - new, 219, 237
 - new(nothrow), 239
 - przesunięcia, 128
 - przypisania, 130, 406
 - reinterpret_cast, 425
 - sizeof, 68, 132
 - static_cast, 420
 - warunkowego, 153
 - wyboru zakresu, 323
 - operatorów
 - bitowych, 126

dekrementacji, 222
 inkrementacji, 222
 logicznych, 122, 125
 negacji, 126
 relacji, 118
 pętli
 do...while, 160
 for, 97, 161
 while, 157
 polecenia continue, 167
 polecenia throw, 757
 poleceń try i catch, 754
 predykatu, 606, 611
 przeciążonej funkcji, 192
 referencji, 240, 243
 słowa kluczowego
 auto, 70, 474
 const, 225, 243, 403
 do, 161
 friend, 299
 inline, 200
 new, 227
 protected, 312
 typedef, 72
 using, 323
 sprytnego wskaźnika, 380, 714, 718
 stałej typu wyliczeniowego, 149
 stałych, 92
 static_assert, 456
 strumienia, 727
 std::cout, 731, 734
 std::cin, 735, 738
 std::fstream, 739
 std::stringstream, 746
 tabeli hash, 588
 typu bool, 64
 typu wyliczeniowego, 77
 wartości zwrotnej, 50
 wielu parametrów, 188
 wielu poleceń return, 190
 wielu rdzeni, 771
 wirtualnego destruktora, 349
 wskaźników, 234, 710
 wyrażenia lambda, 620–623
 wzorca, 446, 452, 457
 zagnieżdżonych pętli, 169, 171, 173
 zagnieżdżonych poleceń if, 145
 zmiennych, 57, 62, 454

V

VFT, Virtual Function Table, 352

W

wartość
 domyślna, 186
 null, 208, 275
 wskaźnika, 213
 zwrotna, 44
 warunkowe wykonanie kodu, 141–154
 wątki
 komunikacja, 773
 stan wyścigu, 775
 synchronizacja, 774
 zakleszczenie, 775
 wejście, 51
 wektor, 98
 wielkość
 adresu, 216
 ciągu tekstowego, 275, 282
 liczby całkowitej, 210
 obiektu vector, 513
 tablicy, 92
 wskaźnika, 216, 217
 zmiennej, 68
 wielowątkowość, 775, 780
 wirtualne konstruktory kopiujące, 363
 własna klasa wyjątku, 762
 własny predykat sortowania, 583
 właściwości klas kontenerów, 467–469
 wskazywanie bloku pamięci, 208
 wskaźnik, 207, 216
 do liczby całkowitej, 211
 do pierwszego elementu tablicy, 228
 this, 292
 VFT, 354
 void, 208
 wskaźniki
 nieprawidłowe, 232
 sprytnie, 380, 384
 zawieszane, 234
 wstawianie
 elementów, 503–505, 526–529
 do kolejki, 516, 684
 do kolejki priorytetowej, 689
 do kolekcji, 669
 do obiektów map, 573
 do obiektów set, 552
 do obiektu list, 529
 do zakresu, 673
 na stos, 679
 tekstu, 738
 znaków do bufora, 737
 wybór typu kontenera, 466, 592

- wyciek pamięci, 232
- wydajność, 284
- wyjątek, 236, 751–766
 - bad_cast, 761
 - ios_base::failure, 761
 - std::bad_alloc, 755, 761
 - std::exception, 761, 764
- wyjście, 51
- wykonywanie kodu w pętlach, 154
- wyrażenia lambda, 617–633
 - dla funkcji
 - dwuargumentowej, 626
 - jednoargumentowej, 619
 - dla predykatu
 - dwuargumentowego, 628
 - jednoargumentowego, 621
 - wraz ze stanem, 622
- wyszukiwanie
 - elementów, 472, 645
 - w obiekcie map, 577
 - w obiekcie multimap, 579
 - w obiektach set, 554
 - pary klucz-wartość, 577
 - podciągu tekstowego, 488
 - zakresu, 646
 - znaku, 488
- wyświetlanie
 - adresu zmiennej, 212
 - ciągu tekstowego, 103
 - danych w konsoli, 728
 - danych wejściowych, 52
 - tekstu, 110
 - danych na ekranie, 51
 - elementów kontenera, 619
 - kolekcji, 601
 - zawartości kolekcji, 652
 - zawartości obiektu, 620
- wywoływanie
 - funkcji, 182, 195
 - metod, 342
 - metod klasy bazowej, 319
 - nadpisanych metod, 319
 - operatora new, 237
- wznawianie działania pętli, 165
- wzorce, 433, 444
 - funkcje, 446
 - rodzaje deklaracji, 446
 - specjalizacja, 450
 - ustanawianie, 450
 - z parametrami domyślnymi, 452
 - z wieloma parametrami, 451

Z

- zagnieżdżone polecenia, 145
- zakres
 - funkcji, 184
 - zmiennej, 59
- zamykanie pliku, 740
- zapis
 - danych w strumieniu, 729
 - pliku binarnego, 744
 - struktury w pliku binarnym, 744
 - strumienia w zmiennej, 729
 - tekstu w pliku, 741
- zarządzanie alokacją pamięci, 273
- zastępowanie
 - elementu, 661
 - tekstu, 434, 436
 - typu zmiennej, 72
 - wartości, 661
- zastosowanie nawiasów, 441
- zgłaszanie własnego wyjątku, 757
- zintegrowane środowisko programistyczne, IDE, 33
- zliczanie
 - elementów, 643
 - odniesień, 716
- zmiennie, 55
 - globalne, 61, 83
 - lokalne, 61, 184
- znajdowanie elementów, 640
- znak
 - #, 42
 - końca ciągu tekstowego, 99
 - liczby, 65
 - nowego wiersza, 44
 - null, 99, 195, 222
 - tyldy, 272
 - ukośnika, 110
- znaki
 - dwukropków, 262
 - formatujące, 34
- zwalnianie pamięci, 218, 232, 246
 - automatyczne, 275
 - destruktor, 273
 - operator delete, 219

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>

Poznaj i wykorzystaj potencjał języka C++!

Pomimo swojego wieku C++ wciąż zajmuje wysoką lokatę na liście najpopularniejszych języków programowania. Przez lata dzięki wydajności oraz dostępności zdobył sobie licznych wielbicieli i wciąż jest niezastąpiony w wielu zadaniach, choć w branży przewagę nad nim mają język Java i platforma .NET. Jeżeli jednak oczekujesz najwyższej wydajności i przewidywalnego czasu wykonania zadania, język C++ może okazać się jedynym słusznym wyborem.

Z tą książką opanujesz język C++ bez trudu, poświęcając na to tylko godzinę dziennie! W trakcie lektury zgłębisz jego tajniki, poznasz zalety oraz wady. Każda kolejna godzina to coraz bardziej zaawansowana, a przy tym ciekawsza dawka wiedzy. Opanowanie materiału zawartego w podręczniku pozwoli Ci na pisanie programów o różnym stopniu złożoności oraz swobodne poruszanie się w świecie języka C++. W trakcie lektury opanujesz składnię i elementy języka — tablice, instrukcje warunkowe, pętle czy stałe nie będą miały przed Tobą żadnych tajemnic. Z kolejnych rozdziałów dowiesz się, jak wykorzystywać wskaźniki i dyrektywy kompilatora. Ponadto zaznajomisz się z zasadami programowania obiektowego oraz poznasz nowości zawarte w wersji 11. Książka ta jest doskonałym źródłem wiedzy dla każdego adepta języka C++!

Dzięki tej książce:

- poznasz składnię języka C++
- opanujesz nowości wersji 11 języka C++
- poznasz zasady programowania obiektowego
- napiszesz program o dowolnym stopniu skomplikowania

helion.pl
księgarnia
internetowa

Nr katalogowy: 16326

Księgarnia internetowa:
<http://helion.pl>

Zamówienia telefoniczne:
0 801 339900
0 601 339900

SAMS



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>



ISBN 978-83-246-8077-1



9 788324 680771

Cena: 99,00 zł