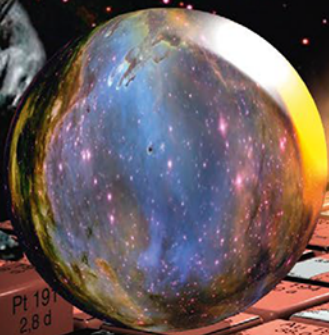


Jerzy Grębosz

OPUS MAGNUM

C++

Programowanie w języku C++



WYDANIE III
POPRAWIONE

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

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

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

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki i opracowanie graficzne książki: Jerzy Grębosz

Zdjęcie Mgławicy Orzeł w grafice na okładce oraz zdjęcia łazika Curiosity i powierzchni Marsa – wykorzystane w tytułach rozdziałów – dzięki uprzejmości NASA.

Wydanie trzecie, poprawione

ISBN: 978-83-289-1131-4

Copyright © Jerzy Grębosz 2024
Printed in Poland

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 230 98 63

e-mail: helion@helion.pl

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

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<https://helion.pl/user/opinie/ocppk3>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Kody źródłowe wybranych przykładów dostępne są pod adresem:

<https://ftp.helion.pl/przyklady/ocppk3.zip>

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

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

Spis treści trzech

Tom 1

0	Proszę tego nie czytać!.....	1
0.1	Zaprzyjaźnijmy się!	1
1	Startujemy!.....	8
1.1	Pierwszy program.....	8
1.2	Drugi program	13
1.3	Ćwiczenia	18
2	Instrukcje sterujące.....	20
2.1	Prawda – fałsz, czyli o warunkach	20
2.1.1	Wyrażenie logiczne	20
2.1.2	Zmienna logiczna <i>bool</i> w roli warunku.....	21
2.1.3	Stare dobre sposoby z dawnego C++	21
2.2	Instrukcja warunkowa <i>if</i>	22
2.3	Pętla <i>while</i>	26
2.4	Pętla <i>do...while</i>	27
2.5	Pętla <i>for</i>	28
2.6	Instrukcja <i>switch</i>	31
2.7	Co wybrać: <i>switch</i> czy <i>if...else</i> ?	33
2.8	Instrukcja <i>break</i>	36
2.9	Instrukcja <i>goto</i>	37
2.10	Instrukcja <i>continue</i>	39
2.11	Klamry w instrukcjach sterujących.....	40
2.12	Ćwiczenia	41
3	Typy	44
3.1	Deklaracje typu.....	44
3.2	Systematyka typów z języka C++	45
3.3	Typy fundamentalne.....	46
3.3.1	Typy przeznaczone do pracy z liczbami całkowitymi.....	46
3.3.2	Typy do przechowywania znaków alfanumerycznych.....	47
3.3.3	Typy reprezentujące liczby zmiennoprzecinkowe	47

3.3.4	bool – typ do reprezentacji obiektów logicznych.....	48
3.3.5	Kwestia dokładności	49
3.3.6	Jak poznać limity (ograniczenia) typów wbudowanych.....	51
3.4	Typy o precyzyjnie żądanej szerokości	55
3.5	Inicjalizacja, czyli nadanie wartości w momencie narodzin.....	59
3.6	Definiowanie obiektów „w biegu”	60
3.7	Stałe dosłowne.....	62
3.7.1	Stałe dosłowne typu <i>bool</i>	63
3.7.2	Stałe będące liczbami całkowitymi	63
3.7.3	Stała reprezentująca liczby zmiennoprzecinkowe.....	66
3.7.4	Stała dosłowna <i>nullptr</i> – dla wskaźników	67
3.7.5	Stałe znakowe	68
3.7.6	Stałe tekstowe, napisy, albo po prostu stringi	71
3.7.7	Surowe stałe tekstowe (napisy, stringi).....	73
3.8	Typy złożone	76
3.9	Typ <i>void</i>	77
3.10	Zakres ważności nazwy obiektu a czas życia obiektu	78
3.10.1	Zakres: lokalny	78
3.10.2	Zakres: instrukcja	79
3.10.3	Zakres: blok funkcji	79
3.10.4	Zakres: obszar pliku	80
3.10.5	Zakres: obszar klasy	80
3.10.6	Zakres określony przez przestrzeń nazw	80
3.11	Zasłanianie nazw	85
3.12	Specyfikator (przydomek) <i>const</i>	87
3.13	Specyfikator (przydomek) <i>constexpr</i>	88
3.14	Obiekty <i>register</i>	92
3.15	Specyfikator <i>volatile</i>	93
3.16	<i>using</i> oraz <i>typedef</i> – tworzenie dodatkowej nazwy typu	94
3.17	Typy wyliczeniowe <i>enum</i>	97
3.17.1	Dawne zwykłe <i>enum</i> a nowe zakresowe <i>enum class</i>	103
3.17.2	 Kilka uwag dla wtajemniczonych.....	106
3.18	<i>auto</i> , czyli automatyczne rozpoznawanie typu definiowanego obiektu	106
3.19	<i>decltype</i> – operator do określania typu zadanego wyrażenia.....	110
3.20	Inicjalizacja z pustą klamrą { }, czyli wartością domniemaną	111
3.21	Przydomek <i>alignas</i> – adresy równe i równiejsze	114
3.22	Ćwiczenia	115

4 Operatory 120

4.1	Operatory arytmetyczne	120
4.1.1	Operator %, czyli reszta z dzielenia (modulo)	121
4.1.2	Jednoargumentowe operatory + i –.....	122
4.1.3	Operatory inkrementacji i dekrementacji	122
4.1.4	Operator przypisania =	124
4.2	Operatory logiczne	125
4.2.1	Operatory relacji	125
4.2.2	Operatory sumy logicznej oraz iloczynu logicznego &&.....	126
4.2.3	Wykrzyknik !, czyli operator negacji	128
4.3	Operatory bitowe	128
4.3.1	Przesunięcie w lewo <<	129
4.3.2	Przesunięcie w prawo >>	130
4.3.3	Bitowe operatory sumy, iloczynu, negacji, różnicy symetrycznej	131
4.4	Różnica między operatorami logicznymi a operatorami bitowymi	131
4.5	Pozostałe operatory przypisania	133
4.6	Operator uzyskiwania adresu (operator &).....	135

4.7	Wyrażenie warunkowe	135
4.8	Operator <i>sizeof</i>	137
4.9	Operator <i>noexcept</i>	138
4.10	Deklaracja <i>static_assert</i>	138
4.11	Operator <i>alignof</i> informujący o najkorzystniejszym wyrównaniu adresu	141
4.12	Operatory rzutowania	142
4.12.1	Rzutowanie według tradycyjnych (niezalecanych) sposobów	142
4.12.2	Rzutowanie za pomocą nowych operatorów rzutowania	144
4.12.3	Operator <i>static_cast</i>	144
4.12.4	Operator <i>const_cast</i>	147
4.12.5	Operator <i>dynamic_cast</i>	148
4.12.6	Operator <i>reinterpret_cast</i>	148
4.13	Operator: przecinek	149
4.14	Priorytety operatorów	150
4.15	Łączność operatorów	152
4.16	Ćwiczenia	153

5 Typ *string* i typ *vector* – pierwsza wzmianka157

5.1	Typ <i>std::string</i> do pracy z tekstami	157
5.2	Typ <i>vector</i> – długi rząd obiektów	162
5.3	Zakresowe <i>for</i>	170
5.4	Ćwiczenia	173

6 Funkcje175

6.1	Definicja funkcji i jej wywołanie	175
6.2	Deklaracja funkcji	176
6.3	Funkcja często wywołuje inną funkcję	178
6.4	Zwracanie przez funkcję rezultatu	178
6.4.1	Obiekt tworzony za pomocą <i>auto</i> , a inicjalizowany rezultatem funkcji	180
6.4.2	O zwracaniu (lub niezwracaniu) rezultatu przez funkcję <i>main</i>	181
6.5	Nowy, alternatywny sposób deklaracji funkcji	182
6.6	Stos	184
6.7	Przesyłanie argumentów do funkcji przez wartość	185
6.8	Przesyłanie argumentów przez referencje	186
6.9	Pożyteczne określenia: l-wartość i r-wartość	189
6.10	Referencje do l-wartości i referencje do r-wartości jako argumenty funkcji	191
6.10.1	Który sposób przesyłania argumentu do funkcji wybrać?	198
6.11	Kiedy deklaracja funkcji nie jest konieczna?	199
6.12	Argumenty domniemane	200
6.12.1	Ciekawostki na temat argumentów domniemanych	203
6.13	Nienazwany argument	208
6.14	Funkcje <i>inline</i> (w linii)	209
6.15	Przypomnienie o zakresie ważności nazw deklarowanych wewnątrz funkcji	213
6.16	Wybór zakresu ważności nazwy i czasu życia obiektu	213
6.16.1	Obiekty globalne	213
6.16.2	Obiekty automatyczne	214
6.16.3	Obiekty lokalne statyczne	215
6.17	Funkcje w programie składającym się z kilku plików	219
6.17.1	Nazwy statyczne globalne	223
6.18	Funkcja zwracająca rezultat będący referencją l-wartości	224
6.19	Funkcje rekurencyjne	229
6.20	Funkcje biblioteczne	238
6.21	Funkcje <i>constexpr</i>	241
6.21.1	Wymogi, które musi spełniać funkcja <i>constexpr</i> (w standardzie C++11)	243

6.21.2	Przykład pokazujący aspekty funkcji <i>constexpr</i>	244
6.21.3	Argumenty funkcji <i>constexpr</i> będące referencjami	253
6.22	Definiowanie referencji przy użyciu słowa <i>auto</i>	254
6.22.1	Gdy inicjalizatorem jest wywołanie funkcji zwracającej referencję.....	261
6.23	Ćwiczenia	264

7 Preprocesor270

7.1	Dyrektywa pusta <i>#</i>	270
7.2	Dyrektywa <i>#define</i>	270
7.3	Dyrektywa <i>#undef</i>	272
7.4	Makrodefinicje	273
7.5	Sklejacz nazw argumentów, czyli operator <i>##</i>	275
7.6	Parametr aktualny makrodefinicji – w postaci tekstu	276
7.7	Dyrektywy kompilacji warunkowej	276
7.8	Dyrektywa <i>#error</i>	280
7.9	Dyrektywa <i>#line</i>	281
7.10	Wstawianie treści innych plików do tekstu kompilowanego właśnie pliku	281
7.11	Dyrektywy zależne od implementacji	283
7.12	Nazwy predefiniowane	283
7.13	Ćwiczenia	286

8 Tablice289

8.1	Co to jest tablica	289
8.2	Elementy tablicy	290
8.3	Inicjalizacja tablic.....	292
8.4	Przekazywanie tablicy do funkcji.....	293
8.5	Przykład z tablicą elementów typu <i>enum</i>	297
8.6	Tablice znakowe.....	299
8.7	Ćwiczenia	307

9 Tablice wielowymiarowe.....312

9.1	Tablica tablic	312
9.2	Przykład programu pracującego z tablicą dwuwymiarową	314
9.3	Gdzie w pamięci jest dany element tablicy.....	316
9.4	Typ wyrażeń związanych z tablicą wielowymiarową.....	316
9.5	Przesyłanie tablic wielowymiarowych do funkcji	318
9.6	Ćwiczenia	320

10 Wektory wielowymiarowe.....322

10.1	Najpierw przypomnienie istotnych tu cech klasy <i>vector</i>	322
10.2	Jak za pomocą klasy <i>vector</i> budować tablice wielowymiarowe	323
10.3	Funkcja pokazująca zawartość wektora dwuwymiarowego	324
10.4	Definicja dwuwymiarowego wektora – pustego	326
10.5	Definicja wektora dwuwymiarowego z listą inicjalizatorów	327
10.6	Wektor dwuwymiarowy o żądanych rozmiarach, choć bez inicjalizacji	328
10.7	Zmiana rozmiaru wektora 2D funkcją <i>resize</i>	329
10.8	Zmiany rozmiaru wektora 2D funkcjami <i>push_back</i> , <i>pop_back</i>	330
10.9	Zmniejszanie rozmiaru wektora dwuwymiarowego funkcją <i>pop_back</i>	333
10.10	Funkcje mogące modyfikować treść wektora 2D.....	333
10.11	Wysłanie rzędu wektora 2D do funkcji pracującej z wektorem 1D.....	335
10.12	Całość przykładu definiującego wektory dwuwymiarowe	336
10.13	Po co są dwuwymiarowe wektory nieprostokątne.....	336

10.14	Wektory trójwymiarowe.....	338
10.15	Sposoby definicji wektora 3D o ustalonych rozmiarach	341
10.16	Nadawanie pustemu wektorowi 3D wymaganych rozmiarów.....	345
10.16.1	Zmiana rozmiarów wektora 3D funkcjami <i>resize</i>	345
10.16.2	Zmiana rozmiarów wektora 3D funkcjami <i>push_back</i>	347
10.17	Trójwymiarowe wektory 3D – nieprostokątne.....	348
10.18	Ćwiczenia	352

11 Wskaźniki – wiadomości wstępne354

11.1	Wskaźniki mogą bardzo ułatwić życie	354
11.2	Definiowanie wskaźników	356
11.3	Praca ze wskaźnikiem.....	357
11.4	Definiowanie wskaźnika z użyciem <i>auto</i>	360
11.5	Wyrażenie <i>*wskaźnik</i> jest l-wartością	361
11.6	Operator rzutowania <i>reinterpret_cast</i> a wskaźniki.....	361
11.7	Wskaźniki typu <i>void*</i>	364
11.8	Strzał na ośle – wskaźnik zawsze na coś wskazuje.....	366
11.8.1	Wskaźnik wolno porównać z adresem zero – <i>nullptr</i>	368
11.9	Ćwiczenia	368

12 Cztery domeny zastosowania wskaźników370

12.1	Zastosowanie wskaźników wobec tablic.....	370
12.1.1	Ćwiczenia z mechaniki ruchu wskaźnika.....	370
12.1.2	Użycie wskaźnika w pracy z tablicą.....	374
12.1.3	Arytmetyka wskaźników.....	378
12.1.4	Porównywanie wskaźników.....	380
12.2	Zastosowanie wskaźników w argumentach funkcji.....	381
12.2.1	Jeszcze raz o przesyłaniu tablic do funkcji.....	385
12.2.2	Odbieranie tablicy jako wskaźnika.....	385
12.2.3	Argument formalny będący wskaźnikiem do obiektu <i>const</i>	387
12.3	Zastosowanie wskaźników przy dostępie do konkretnych komórek pamięci.....	390
12.4	Rezerwacja obszarów pamięci.....	391
12.4.1	Operatory <i>new</i> i <i>delete</i> albo Oratorium Stworzenie Świata.....	392
12.4.2	Operator <i>new</i> a słowo kluczowe <i>auto</i>	396
12.4.3	Inicjalizacja obiektu tworzonego operatorem <i>new</i>	396
12.4.4	Operatorem <i>new</i> możemy także tworzyć obiekty stałe.....	397
12.4.5	Dynamiczna alokacja tablicy.....	398
12.4.6	Tablice wielowymiarowe tworzone operatorem <i>new</i>	399
12.4.7	Umiejscawiający operator <i>new</i>	402
12.4.8	„Przychodzimy, odchodzimy – cichuteńko, na...”	407
12.4.9	Zapas pamięci to nie studnia bez dna	409
12.4.10	Nowy sposób powiadomienia: rzucenie wyjątku <i>std::bad_alloc</i>	410
12.4.11	Funkcja <i>set_new_handler</i>	412
12.5	Ćwiczenia	414

13 Wskaźniki – runda trzecia.....418

13.1	Stałe wskaźniki.....	418
13.2	Stałe wskaźniki a wskaźniki do stałych.....	419
13.2.1	Wierzch i głębia	420
13.3	Definiowanie wskaźnika z użyciem <i>auto</i>	421
13.3.1	Symbol zastępczy <i>auto</i> a opuszczanie gwiazdki przy definiowaniu wskaźnika.....	424
13.4	Sposoby ustawiania wskaźników	426
13.5	Parada kłamek, czyli o rzutowaniu <i>const_cast</i>	428

13.6	Tablice wskaźników	432
13.7	Wariacje na temat C-stringów	434
13.8	Argumenty z linii wywołania programu	441
13.9	Cwiczenia	444

14 Wskaźniki do funkcji446

14.1	Wskaźnik, który może wskazywać na funkcję	446
14.2	Cwiczenia z definiowania wskaźników do funkcji	449
14.3	Wskaźnik do funkcji jako argument innej funkcji	455
14.4	Tablica wskaźników do funkcji	459
14.5	Użycie deklaracji <i>using</i> i <i>typedef</i> w świecie wskaźników	464
14.5.1	Alias przydatny w argumencie funkcji	464
14.5.2	Alias przydatny w definicji tablicy wskaźników do funkcji	465
14.6	Użycie <i>auto</i> lub <i>decltype</i> do automatycznego rozpoznania potrzebnego typu	466
14.7	Cwiczenia	468

15 Przeladowanie nazwy funkcji470

15.1	Co oznacza przeladowanie	470
15.2	Przeladowanie od kuchni	473
15.3	Jak możemy przeladowywać, a jak się nie da?	473
15.4	Czy przeladowanie nazw funkcji jest techniką orientowaną obiektowo?	476
15.5	Linkowanie z modułami z innych języków	477
15.6	Przeladowanie a zakres ważności deklaracji funkcji	478
15.7	Rozważania o identyczności lub odmienności typów argumentów	480
15.7.1	Przeladowanie a typy tworzone z <i>using</i> lub <i>typedef</i> oraz typy <i>enum</i>	481
15.7.2	Tablica a wskaźnik	481
15.7.3	Pewne szczegóły o tablicach wielowymiarowych	482
15.7.4	Przeladowanie a referencja	484
15.7.5	Identyczność typów: <i>T</i> , <i>const T</i> , <i>volatile T</i>	485
15.7.6	Przeladowanie a typy: <i>T*</i> , <i>volatile T*</i> , <i>const T*</i>	486
15.7.7	Przeladowanie a typy: <i>T&</i> , <i>volatile T&</i> , <i>const T&</i>	487
15.8	Adres funkcji przeladowanej	488
15.8.1	Zwrot rezultatu będącego adresem funkcji przeladowanej	490
15.9	Kulisy dopasowywania argumentów do funkcji przeladowanych	492
15.10	Etapy dopasowania	493
15.10.1	Etap 1. Dopasowanie dokładne, bo konwersja niepotrzebna	493
15.10.2	Etap 1a. Dopasowanie dokładne, bo z tzw. trywialną konwersją	494
15.10.3	Etap 2. Dopasowanie z awansem (z promocją)	495
15.10.4	Etap 3. Próba dopasowania za pomocą konwersji standardowych	497
15.10.5	Etap 4. Dopasowanie z użyciem konwersji zdefiniowanych przez użytkownika	499
15.10.6	Etap 5. Dopasowanie do funkcji z wielokropkiem	499
15.11	Wskaźników nie dopasowuje się inaczej niż dosłownie	499
15.12	Dopasowywanie wywołań z kilkoma argumentami	500
15.13	Cwiczenia	501

16 Klasy504

16.1	Typy definiowane przez użytkownika	504
16.2	Składniki klasy	506
16.3	Składnik będący obiektem	507
16.4	Kapsułowanie	508
16.5	Ukrywanie informacji	509
16.6	Klasa a obiekt	512
16.7	Wartości wstępne w składnikach nowych obiektów. Inicjalizacja „w klasie”	514
16.8	Funkcje składowe	517

16.8.1	Posługiwanie się funkcjami składowymi	517
16.8.2	Definiowanie funkcji składowych	518
16.9	Jak to właściwie jest? (<i>this</i>)	523
16.10	Odwołanie się do publicznych danych składowych obiektu	525
16.11	Zastępowanie nazw	526
16.11.1	Nie sięgaj z klasy do obiektów globalnych	529
16.12	Przeładowanie i zastąpienie równocześnie	530
16.13	Nowa klasa? Osobny plik!	530
16.13.1	Poznajmy praktyczną realizację wieloplikowego programu	533
16.13.2	Zasada umieszczania dyrektywy <i>using namespace</i> w plikach	545
16.14	Przesyłanie do funkcji argumentów będących obiektami	545
16.14.1	Przesyłanie obiektu przez wartość	545
16.14.2	Przesyłanie przez referencję	547
16.15	Konstruktor – pierwsza wzmianka	548
16.16	Destruktor – pierwsza wzmianka	553
16.17	Składnik statyczny	557
16.17.1	Do czego może się przydać składnik statyczny w klasie?	566
16.18	Styczna funkcja składowa	566
16.18.1	Deklaracja składnika statycznego mająca inicjalizację „w klasie”	571
16.19	Funkcje składowe typu <i>const</i> oraz <i>volatile</i>	577
16.19.1	Przeładowanie a funkcje składowe <i>const</i> i <i>volatile</i>	581
16.20	Struktura	581
16.21	Klasa będąca agregatem. Klasa bez konstruktora	582
16.22	Funkcje składowe z przydomkiem <i>constexpr</i>	584
16.23	Specyfikator <i>mutable</i>	591
16.24	Bardziej rozbudowany przykład zastosowania klasy	592
16.25	Ćwiczenia	603

Tom 2

17	Biblioteczna klasa <i>std::string</i>	609
17.1	Rozwiązanie przechowywania tekstów musiało się znaleźć	609
17.2	Klasa <i>std::string</i> to przecież nasz stary znajomy	611
17.3	Definiowanie obiektów klasy <i>string</i>	612
17.4	Użycie operatorów <i>=</i> , <i>+</i> , <i>+=</i> w pracy ze stringami	617
17.5	Pojemność, rozmiar i długość stringu	618
17.5.1	Blizniacze funkcje <i>size()</i> i <i>length()</i>	618
17.5.2	Funkcja składowa <i>empty</i>	619
17.5.3	Funkcja składowa <i>max_size</i>	619
17.5.4	Funkcja składowa <i>capacity</i>	619
17.5.5	Funkcje składowe <i>reserve</i> i <i>shrink_to_fit</i>	621
17.5.6	<i>resize</i> – zmiana długości stringu „na siłę”	622
17.5.7	Funkcja składowa <i>clear</i>	624
17.6	Użycie operatora <i>[]</i> oraz funkcji <i>at</i>	624
17.6.1	Działanie operatora <i>[]</i>	625
17.6.2	Działanie funkcji składowej <i>at</i>	626
17.6.3	Przebieganie po wszystkich literach stringu zakresowym <i>for</i>	629
17.7	Funkcje składowe <i>front</i> i <i>back</i>	629
17.8	Jak umieścić w tekście liczbę?	630
17.9	Jak wczytać liczbę ze stringu?	632

17.10	Praca z fragmentem stringu, czyli z substryniem	635
17.11	Funkcja składowa <i>substr</i>	636
17.12	Szukanie zadanego substrynu w obiekcie klasy <i>string</i> – funkcje <i>find</i>	637
17.13	Szukanie rozpoczynane od końca stringu	640
17.14	Szukanie w stringu jednego ze znaków z zadanego zestawu	641
17.15	Usuwanie znaków ze stringu – <i>erase</i> i <i>pop_back</i>	643
17.16	Wstawianie znaków do istniejącego stringu – funkcje <i>insert</i>	644
17.17	Zamiana części znaków na inne znaki – <i>replace</i>	646
17.18	Zagłądanie do wnętrza obiektu klasy <i>string</i> funkcją <i>data</i>	649
17.19	Zawartość obiektu klasy <i>string</i> a C-string	650
17.20	W porządku alfabetycznym, czyli porównywanie stringów	653
17.20.1	Porównywanie stringów za pomocą funkcji <i>compare</i>	654
17.20.2	Porównywanie stringów przy użyciu operatorów <i>==</i> , <i>!=</i> , <i><</i> , <i>></i> , <i><=</i> , <i>>=</i>	658
17.21	Zamiana treści stringu na małe lub wielkie litery	659
17.22	Kopiowanie treści obiektu klasy <i>string</i> do tablicy znakowej – funkcja <i>copy</i>	661
17.23	Wzajemna zamiana treści dwóch obiektów klasy <i>string</i> – funkcja <i>swap</i>	662
17.24	Wczytywanie z klawiatury stringu o nieznannej wcześniej długości – <i>getline</i>	663
17.24.1	Pułapka, czyli jak <i>getline</i> może Cię zaskoczyć	666
17.25	Iteratory stringu	670
17.25.1	Iterator do obiektu stałego	674
17.25.2	Funkcje składowe klasy <i>string</i> pracujące z iteratorami	675
17.26	Klasa <i>string</i> korzysta z techniki przenoszenia	680
17.27	Bryk, czyli „pamięć zewnętrzna” programisty	681
17.28	Ćwiczenia	689

18 Deklaracje przyjaźni696

18.1	Przyjaciele w życiu i w C++	696
18.2	Przykład: dwie klasy deklarują przyjaźń z tą samą funkcją	698
18.3	W przyjaźni trzeba pamiętać o kilku sprawach	700
18.4	Obdarzenie przyjaźnią funkcji składowej innej klasy	703
18.5	Klasy zaprzyjaźnione	705
18.6	Konwencja umieszczania deklaracji przyjaźni w klasie	707
18.7	Kilka otrzeźwiających słów na zakończenie	707
18.8	Ćwiczenia	708

19 Obsługa sytuacji wyjątkowych710

19.1	Jak dać znać, że coś się nie udało?	710
19.2	Pierwszy prosty przykład	712
19.3	Kolejność bloków <i>catch</i> ma znaczenie	714
19.4	Który blok <i>catch</i> nadaje się do złapania lecącego wyjątku?	715
19.5	Bloki <i>try</i> mogą być zagnieżdżane	717
19.6	Obsługa wyjątków w praktycznym programie	720
19.7	Specyfikator <i>noexcept</i> i operator <i>noexcept</i>	731
19.8	Ćwiczenia	734

20 Klasa-składnik oraz klasa lokalna736

20.1	Klasa-składnik, czyli gdy w klasie jest zagnieżdżona definicja innej klasy	736
20.2	Prawdziwy przykład zagnieżdżenia definicji klasy	743
20.3	Lokalna definicja klasy	754
20.4	Lokalne nazwy typów	757
20.5	Ćwiczenia	758

21	Konstruktory i destruktory	760
21.1	Konstruktor	760
21.1.1	Przykład programu zawierającego klasę z konstruktorami	761
21.2	Specyfikator (przydomek) <i>explicit</i>	772
21.3	Kiedy i jak wywoływany jest konstruktor	773
21.3.1	Konstruowanie obiektów lokalnych	773
21.3.2	Konstruowanie obiektów globalnych	774
21.3.3	Konstrukcja obiektów tworzonych operatorem <i>new</i>	774
21.3.4	Jawne wywołanie konstruktora	775
21.3.5	Dalsze sytuacje, gdy pracuje konstruktor	778
21.4	Destruktor	778
21.4.1	Jawne wywołanie destruktora (ogromnie rzadka sytuacja)	780
21.5	Nie rzucanie wyjątków z destruktorów	780
21.6	Konstruktor domniemany	782
21.7	Funkcje składowe z przypiskami = <i>default</i> i = <i>delete</i>	783
21.8	Konstruktorowa lista inicjalizacyjna składników klasy	785
21.8.1	Dla wtajemniczonych: wyjątki rzucone z konstruktorowej listy inicjalizacyjnej	792
21.9	Konstruktor delegujący	796
21.10	Pomocnicza klasa <i>std::initializer_list</i> – lista inicjalizatorów	803
21.10.1	Zastosowania niekonstruktorowe	803
21.10.2	Konfuzja: lista inicjalizatorów a lista inicjalizacyjna	812
21.10.3	Konstruktor z argumentem będącym klamrową listą inicjalizatorów	813
21.11	Konstrukcja obiektu, którego składnikiem jest obiekt innej klasy	818
21.12	Konstruktory niepubliczne?	825
21.13	Konstruktory <i>constexpr</i> mogą wytwarzać obiekty <i>constexpr</i>	827
21.14	Ćwiczenia	837
22	Konstruktory: kopiujący i przenoszący	840
22.1	Konstruktor kopiujący (albo inicjalizator kopiujący)	840
22.2	Przykład klasy z konstruktorem kopiującym	841
22.3	Kompilatorowi wolno pominąć niepotrzebne kopiowanie	846
22.4	Dlaczego przez referencję?	848
22.5	Konstruktor kopiujący gwarantujący nietykalność	849
22.6	Współodpowiedzialność	850
22.7	Konstruktor kopiujący generowany automatycznie	850
22.8	Kiedy powinniśmy sami zdefiniować konstruktor kopiujący?	851
22.9	Referencja do r-wartości daje zezwolenie na recykling	858
22.10	Funkcja <i>std::move</i> , która nie przenosi, a tylko rzutuje	861
22.11	Odebrana r-wartość staje się w ciele funkcji l-wartością	863
22.12	Konstruktor przenoszący (inicjalizator przenoszący)	865
22.12.1	Konstruktor przenoszący generowany przez kompilator	870
22.12.2	Inne konstruktory generowane automatycznie	870
22.12.3	Zwrot obiektu lokalnego przez wartość? Nie używamy przenoszenia!	871
22.13	Tak zwana „semantyka przenoszenia”	872
22.14	Nowe pojęcia dla ambitnych: gl-wartość, x-wartość i pr-wartość	872
22.15	<i>decltype</i> – operator rozpoznawania typu bardzo wyszukanych wyrażeń	875
22.16	Ćwiczenia	880
23	Tablice obiektów	882
23.1	Definiowanie tablic obiektów i praca z nimi	882
23.2	Tablica obiektów definiowana operatorem <i>new</i>	883
23.3	Inicjalizacja tablic obiektów	885
23.3.1	Inicjalizacja tablicy, której obiekty są agregatami	885

23.3.2	Inicjalizacja tablic, których elementy nie są agregatami	888
23.4	Wektory obiektów	892
23.4.1	Wektor, którego elementami są obiekty klasy będącej agregatem	894
23.4.2	Wektor, którego elementami są obiekty klasy niebędącej agregatem	896
23.5	Ćwiczenia	897

24 Wskaźnik do składników klasy898

24.1	Wskaźniki zwykłe – repetytorium	898
24.2	Wskaźnik do pokazywania na składnik-daną	899
24.2.1	Przykład zastosowania wskaźników do składników klasy	903
24.3	Wskaźnik do funkcji składowej	910
24.3.1	Przykład zastosowania wskaźników do funkcji składowych	912
24.4	Tablica wskaźników do danych składowych klasy	919
24.5	Tablica wskaźników do funkcji składowych klasy	920
24.5.1	Przykład tablicy/wektora wskaźników do funkcji składowych	921
24.6	Wskaźniki do składników statycznych są zwykłe	924
24.7	Ćwiczenia	925

25 Konwersje definiowane przez użytkownika927

25.1	Sformułowanie problemu	927
25.2	Konstruktory konwertujące	929
25.2.1	Kiedy jawnie, kiedy niejawnie	930
25.2.2	Przykład konwersji konstruktorem	935
25.3	Funkcja konwertująca – operator konwersji	937
25.3.1	Na co funkcja konwertująca zamieniać nie może	943
25.4	Który wariant konwersji wybrać?	944
25.5	Sytuacje, w których zachodzi konwersja	946
25.6	Zapis jawnego wywołania konwersji typów	947
25.6.1	Advocatus zapisu przypominającego: „wywołanie funkcji”	947
25.6.2	Advocatus zapisu: „rzutowanie”	948
25.7	Nie całkiem pasujące argumenty, czyli konwersje kompilatora przy dopasowaniu	948
25.8	Kilka rad dotyczących konwersji	953
25.9	Ćwiczenia	954

26 Przeładowanie operatorów956

26.1	Co to znaczy przeładować operator?	956
26.2	Przeładowanie operatorów – definicja i trochę teorii	958
26.3	Moje zabawki	962
26.4	Funkcja operatorowa jako funkcja składowa	963
26.5	Funkcja operatorowa nie musi być przyjacielem klasy	966
26.6	Operatory predefiniowane	966
26.7	Ile operandów ma mieć ten operator?	967
26.8	Operatory jednooperandowe	967
26.9	Operatory dwuoperandowe	970
26.9.1	Przykład na przeładowanie operatora dwuoperandowego	970
26.9.2	Przemienność	972
26.9.3	Choć operatory inne, to nazwę mają tę samą	973
26.10	Przykład zupełnie niematematyczny	973
26.11	Operatory postinkrementacji i postdekrementacji – koniec z niesprawiedliwością	983
26.12	Praktyczne rady dotyczące przeładowania	985
26.13	Pojedynek: operator jako funkcja składowa czy globalna?	987
26.14	Zasłona spada, czyli tajemnica operatora <<	988
26.15	Stałe dosłownie definiowane przez użytkownika	994
26.15.1	Przykład: stałe dosłownie użytkownika odbierane jako gotowane	998

26.15.2	Przykład: stałe dosłowne użytkownika odbierane na surowo	1007
26.16	Ćwiczenia	1010

27 Przeladowanie: =, [], (), ->.....1014

27.1	Cztery operatory, które muszą być niestatycznymi funkcjami składowymi.....	1014
27.2	Operator przypisania = (wersja kopiująca)	1014
27.2.1	Przykład na przeladowanie (kopiującego) operatora przypisania	1016
27.2.2	Przypisanie „kaskadowe”	1023
27.2.3	Po co i jak zabezpieczamy się przed przypisaniem $a = a$	1025
27.2.4	Jak opowiedzieć potocznie o konieczności istnienia operatora przypisania?.....	1026
27.2.5	Kiedy kopiujący operator przypisania nie jest generowany automatycznie	1028
27.3	Przenoszący operator przypisania =	1028
27.4	Specjalne funkcje składowe i nierealna prosta zasada.....	1037
27.5	Operator [].....	1038
27.6	Operator ().....	1042
27.7	Operator ->	1048
27.7.1	„Sprytny wskaźnik” wykorzystuje przeladowanie właśnie tego operatora	1050
27.8	Ćwiczenia	1057

Tom 3

28 Przeladowanie operatorów *new* i *delete* na użytek klasy.....1059

28.1	Po co przeladowujemy operatory <i>new</i> i <i>new[]</i>	1059
28.2	Funkcja <i>operator new</i> i <i>operator new[]</i> w klasie K	1060
28.3	Jak się deklaruje operatory <i>new</i> i <i>delete</i> w klasie?.....	1063
28.4	Przykładowy program z przeladowanymi <i>new</i> i <i>delete</i>	1065
28.4.1	Gdy dopuszczamy rzucanie wyjątku <i>std::bad_alloc</i>	1066
28.4.2	Po starym nadal można.....	1071
28.4.3	Rezerwacja tablicy obiektów naszej klasy <i>Twektorek</i>	1071
28.4.4	Nasze własne argumenty wysłane do operatora <i>new</i>	1073
28.4.5	👑 Operatory <i>new</i> i <i>delete</i> odziedziczone do klasy pochodnej.....	1075
28.4.6	A jednak polimorfizm jest możliwy	1077
28.4.7	Tworzenie i likwidowanie tablicy obiektów klasy pochodnej	1077
28.4.8	Operatory <i>new</i> , które nie rzucają wyjątku <i>std::bad_alloc</i>	1078
28.5	Rzut oka wstecz na przeladowanie operatorów	1083
28.6	Ćwiczenia	1084

29 Unie i pola bitowe

29.1	Unia	1086
29.2	Unia anonimowa.....	1088
29.3	Klasa uniopodobna (unia z metryczką)	1090
29.4	Gdy składnik unii jest obiektem jakiejś klasy	1092
29.5	Unia o składnikach mających swe konstruktory, destruktory itp.	1094
29.6	Pola bitowe	1101
29.7	Unia i pola bitowe upraszczają deszyfrowanie słów danych	1105
29.8	Ćwiczenia	1112

30 Wyrażenia lambda i wysłanie kodu do innych funkcji1116

30.1	Preludium: dwa sposoby przesłania kryterium oceniania.....	1116
30.1.1	Sposób I. Kryterium przekazane wskaźnikiem do funkcji (orzekającej)	1119
30.1.2	Sposób II. Kryterium umieszczone w obiekcie funkcyjnym.....	1121
30.1.3	Kryterium oceny z parametrem (czyli o wyższości funktorów).....	1123
30.1.4	Funkcja-algorytm biblioteczny <i>std::count_if</i>	1125
30.1.5	Co lepsze: funkcja orzekająca czy orzekający obiekt funkcyjny?.....	1128
30.2	Wyrażenie lambda	1130
30.3	Formy wyrażenia lambda	1135
30.3.1	Lista argumentów (formalnych).....	1136
30.3.2	Ciało wyrażenia lambda	1136
30.3.3	Typ rezultatu	1137
30.3.4	Lista wychwytywania.....	1138
30.3.5	Słowo kluczowe <i>mutable</i> w wyrażeniu lambda.....	1140
30.3.6	Specyfikacja dotycząca wyjątków rzuconych z wyrażenia lambda.....	1141
30.4	Wyrażenie lambda zastosowane w funkcji składowej.....	1141
30.5	Tworzenie (nazwanych) obiektów lambda słowem <i>auto</i>	1145
30.5.1	Tworzenie obiektów na lambdy słowem kluczowym <i>auto</i>	1146
30.5.2	Tworzenie (nazwanych) obiektów lambda szablonem <i>std::function</i>	1148
30.6	Stowarzyszenie martwych referencji	1153
30.7	Rekurencja przy użyciu wyrażenia lambda	1156
30.8	Wyrażenie lambda jako domniemana wartość argumentu.....	1160
30.9	Rzucanie wyjątków z wyrażenia lambda.....	1164
30.10	Vivat lambda!	1168
30.11	Ćwiczenia	1169

31 Dziedziczenie klas1172

31.1	Istota dziedziczenia.....	1172
31.2	Dostęp do składników	1175
31.2.1	Prywatne składniki klasy podstawowej.....	1175
31.2.2	Nieprywatne składniki klasy podstawowej.....	1177
31.2.3	Klasa pochodna też decyduje	1178
31.2.4	Deklaracja dostępu <i>using</i> , czyli udostępnianie wybiórcze	1180
31.3	Czego się nie dziedziczy.....	1182
31.3.1	„Niedziedziczenie” konstruktorów	1183
31.3.2	„Niedziedziczenie” operatora przypisania.....	1184
31.3.3	„Niedziedziczenie” destruktora	1184
31.4	Drzewo genealogiczne.....	1184
31.5	Dziedziczenie – doskonałe narzędzie programowania	1186
31.6	Kolejność wywoływania konstruktorów	1188
31.7	Przypisanie i inicjalizacja obiektów w warunkach dziedziczenia.....	1193
31.7.1	Klasa pochodna nie definiuje swojego kopiującego operatora przypisania	1194
31.7.2	Klasa pochodna nie definiuje swojego konstruktora kopiującego	1195
31.7.3	Inicjalizacja i przypisywanie według obiektu będącego <i>const</i>	1196
31.8	Przykład: konstruktor kopiujący i operator przypisania dla klasy pochodnej	1196
31.8.1	Jak zainstalować mechanizm kopiowania w klasie pochodnej	1202
31.8.2	Jak w klasie pochodnej zainstalować mechanizm przenoszenia	1206
31.9	Dziedziczenie od kilku „rodziców” (wielodziedziczenie)	1209
31.9.1	Konstruktor klasy pochodnej przy wielodziedziczeniu.....	1211
31.9.2	Ryzyko wieloznaczności przy wielodziedziczeniu	1213
31.9.3	Czy bliższe pokrewieństwo usuwa wieloznaczność?.....	1215
31.9.4	Poszlaki	1216
31.10	Sposób na „odziedziczenie” konstruktorów	1217
31.11	Pojedynek: dziedziczenie klasy contra zawieranie obiektów składowych	1224

31.12	Wspaniałe konwersje standardowe przy dziedziczeniu	1226
31.12.1	Panorama korzyści	1230
31.12.2	Czego się nie opłaca robić.....	1232
31.12.3	Tuzin samochodów nie jest rodzajem tuzina pojazdów	1233
31.12.4	Konwersje standardowe wskaźnika do składnika klasy	1237
31.13	Wirtualne klasy podstawowe.....	1239
31.13.1	Publiczne i prywatne dziedziczenie tej samej klasy wirtualnej.....	1243
31.13.2	Uwagi o konstrukcji i inicjalizacji w przypadku klas wirtualnych	1243
31.13.3	Dominacja klas wirtualnych.....	1247
31.14	Ćwiczenia	1248

32 Wirtualne funkcje składowe1255

32.1	Wirtualny znaczy: (teoretycznie) możliwy	1255
32.2	Polimorfizm	1262
32.3	Typy rezultatów różnych realizacji funkcji wirtualnej.....	1265
32.3.1	Zamiast „odpowiedni typ rezultatu” kompilator powie „kowariant”	1266
32.4	Dalsze cechy funkcji wirtualnej.....	1268
32.5	Wczesne i późne wiązanie	1270
32.6	Kiedy dla wywołań funkcji wirtualnych zachodzi jednak wczesne wiązanie?.....	1272
32.7	Kulisy białej magii, czyli jak to jest zrobione.....	1273
32.8	Funkcja wirtualna, a mimo to <i>inline</i>	1275
32.9	Destruktor? Najlepiej wirtualny!	1276
32.10	Pojedynek – funkcje przeładowane, zastaniające się i wirtualne (zacierające się)	1278
32.11	Kontekstowe słowa kluczowe <i>override</i> i <i>final</i>	1279
32.11.1	Przykład użycia <i>override</i> i <i>final</i> , a także wirtualnych destruktorów	1281
32.12	Klasy abstrakcyjne.....	1293
32.13	Wprawdzie konstruktor nie może być wirtualny, ale	1300
32.14	Rzutowanie <i>dynamic cast</i> jest dla typów polimorficznych.....	1306
32.15	POD, czyli Pospolite Stare Dane	1309
32.16	Wszystko, co najważniejsze	1312
32.17	Finis coronat opus	1315
32.18	Ćwiczenia	1315

33 Operacje wejścia/wyjścia – podstawy1319

33.1	Biblioteka <i>iostream</i>	1320
33.2	Strumień	1320
33.3	Strumienie zdefiniowane standardowo.....	1322
33.4	Operatory <i>>></i> i <i><<</i>	1323
33.5	Domniemania w pracy strumieni zdefiniowanych standardowo	1324
33.6	Uwaga na priorytet	1327
33.7	Operatory <i><<</i> oraz <i>>></i> definiowane przez użytkownika	1328
33.7.1	Operatorów wstawiania i wyjmowania ze strumienia nie dziedziczy się.....	1333
33.7.2	Operatory wstawiania i wyjmowania nie mogą być wirtualne. Niestety.....	1334
33.8	Sterowanie formatem.....	1337
33.9	Flagi stanu formatowania	1337
33.9.1	Znaczenie poszczególnych flag sterowania formatem	1339
33.10	Sposoby zmiany trybu (reguł) formatowania	1344
33.11	Manipulatory	1344
33.11.1	Manipulatory bezargumentowe	1345
33.11.2	Manipulatory mające argumenty	1350
33.11.3	Manipulator <i>setw(int)</i>	1350
33.11.4	Manipulator <i>setfill</i>	1353
33.11.5	Manipulator <i>setprecision(int)</i>	1353
33.11.6	Manipulator <i>std::setbase(int)</i>	1355
33.11.7	Manipulatory <i>setiosflags</i> , <i>resetiosflags</i>	1356

33.11.8	Tabele z zestawieniem manipulatorów	1356
33.12	Definiowanie swoich manipulatorów	1358
33.12.1	Manipulator jako funkcja	1358
33.12.2	Definiowanie manipulatora z argumentem	1360
33.13	Zmiana sposobu formatowania funkcjami <i>setf</i> , <i>unsetf</i>	1363
33.14	Dodatkowe funkcje do zmiany parametrów formatowania	1369
33.14.1	Funkcja <i>width</i>	1370
33.14.2	Funkcja składowa <i>fill</i>	1371
33.14.3	Funkcja <i>precision</i>	1372
33.14.4	Funkcja <i>copyfmt</i>	1373
33.15	Nieformatowane operacje wejścia/wyjścia	1373
33.16	Omówienie funkcji wyjmujących ze strumienia.....	1375
33.16.1	Funkcje do pracy ze znakami i napisami.....	1375
33.16.2	Wczytywanie binarne – funkcja <i>read</i>	1382
33.16.3	Funkcja <i>ignore</i>	1383
33.16.4	Pożyteczne funkcje pomocnicze	1384
33.16.5	Funkcje wstawiające do strumienia.....	1386
33.17	Ćwiczenia	1388

34 Operacje we/wy na plikach.....1394

34.1	Strumień płynący do lub od plików	1394
34.1.1	Otwieranie i zamykanie strumienia	1396
34.2	Błędy w trakcie pracy strumienia	1401
34.2.1	Flagi stanu błędu strumienia	1401
34.2.2	Funkcje do pracy na flagach błędu.....	1402
34.2.3	Kilka udogodnień dla sprawdzania poprawności	1403
34.2.4	Ustawianie i kasowanie flag błędu strumienia	1404
34.2.5	Trzy plagi, czyli „gotowiec”, jak radzić sobie z błędami	1408
34.3	Przykład programu pracującego na plikach.....	1412
34.4	Przykład programu zapisującego dane tekstowo i binarnie	1414
34.4.1	Zapis w trybie tekstowym	1418
34.4.2	Odczyt z pliku tekstowego	1419
34.4.3	Zapis danych w plikach binarnych.....	1421
34.4.4	Odczyt danych z pliku binarnego.....	1422
34.5	Strumień a technika rzucania wyjątków	1424
34.6	Wybór miejsca czytania lub pisania w pliku	1428
34.6.1	Funkcje składowe informujące o pozycji wskaźników	1429
34.6.2	Wybrane funkcje składowe do pozycjonowania wskaźników	1429
34.7	Pozycjonowanie w przykładzie większego programu	1432
34.8	Tie – harmonijna praca dwóch strumieni	1438
34.9	Ćwiczenia	1440

35 Operacje we/wy na stringach.....1443

35.1	Strumień zapisujący do obiektu klasy <i>string</i>	1443
35.1.1	Przykłady ilustrujące użycie klasy <i>ostream</i>	1447
35.2	Strumień czytający z obiektu klasy <i>string</i>	1450
35.2.1	Prosty przykład użycia strumienia <i>istream</i>	1452
35.2.2	Strumień <i>istream</i> a wczytywanie parametrów-danych	1455
35.2.3	Wczytywanie argumentów wywoływania programu	1460
35.3	Ożenek: strumień <i>stringstream</i> czytający i zapisujący do stringu	1464
35.3.1	Przykładowy program posługujący się klasą <i>stringstream</i>	1465
35.4	Ćwiczenia	1469

36 Projektowanie programów orientowanych obiektowo1471

36.1	Przegląd kilku technik programowania	1471
36.1.1	Programowanie liniowe (linearne)	1472
36.1.2	Programowanie proceduralne (czyli „orientowane funkcyjnie”)	1472
36.1.3	Programowanie z ukrywaniem (zgrupowaniem) danych	1472
36.1.4	Programowanie obiektowe – programowanie bazujące na obiektach	1473
36.1.5	Programowanie obiektowo orientowane (OO).....	1473
36.2	O wyższości programowania OO nad Świętami Wielkiej Nocy	1474
36.3	Obiektowo orientowane: projektowanie	1477
36.4	Praktyczne wskazówki dotyczące projektowania programu techniką OO	1478
36.4.1	Rekonesans, czyli rozpoznanie zagadnienia.....	1479
36.4.2	Faza projektowania	1479
36.4.3	Etap 1. Identyfikacja zachowań systemu.....	1481
36.4.4	Etap 2. Identyfikacja obiektów (klas obiektów).....	1481
36.4.5	Etap 3. Usystematyzowanie klas obiektów	1483
36.4.6	Etap 4. Określenie wzajemnych zależności klas	1484
36.4.7	Etap 5. Składanie modelu. Sekwencje działań obiektów i cykle życiowe	1486
36.5	Faza implementacji.....	1487
36.6	Przykład projektowania	1487
36.7	Rozpoznanie naszego zagadnienia	1488
36.8	Projektowanie	1492
36.8.1	Etap 1. Identyfikacja zachowań naszego systemu.....	1492
36.8.2	Etap 2. Identyfikacja klas obiektów, z którymi mamy do czynienia.....	1493
36.8.3	Etap 3. Usystematyzowanie klas obiektów z naszego systemu.....	1496
36.8.4	Etap 4. Określamy wzajemne zależności klas	1498
36.8.5	Etap 5. Składamy model naszego systemu.....	1500
36.9	Implementacja modelu naszego systemu.....	1505

37 Szablony – programowanie uogólnione1513

37.1	Definiowanie szablonu klas.....	1514
37.2	Prosty program z szablonem klas	1516
37.2.1	Ostrożnie z referencją jako parametrem aktualnym	1518
37.3	Szablon do produkcji funkcji.....	1519
37.4	Cudów nie ma. Sorry.....	1523
37.5	Jak rozmieszczać w plikach szablony klas?.....	1524
37.6	Tylko dla orłów	1525
37.7	Szablony klas, drugie starcie	1525
37.8	Co może być parametrem szablonu – zwiastun	1526
37.9	Rozbudowany przykład z szablonem klas.....	1526
37.9.1	Definiowanie funkcji składowych szablonu klas	1531
37.9.2	Składniki statyczne w szablonie klasy	1532
37.9.3	Obiekt klasy szablonej tworzony operatorem <i>new</i>	1534
37.9.4	Dyrektywa <i>using</i> składnikiem szablonu klas	1535
37.9.5	Przeładowany operator << w szablonie klas	1537
37.9.6	Jawne wywołanie destruktoru klasy szablonej	1538
37.10	Reguła SFINAE.....	1539
37.11	Kiedy kompilator sięga po nasz szablon klas?	1543
37.12	Co może być parametrem szablonu? Szczegóły	1544
37.13	Parametry domniemane	1553
37.13.1	Szablon klas z domniemanymi parametrami.....	1553
37.13.2	Domniemane parametry w szablonie funkcji	1554
37.14	Zagnieżdżenie a szablon	1556
37.14.1	Szablon funkcji składowych zagnieżdżony w szablonie klasy	1557
37.14.2	Szablon klasy zagnieżdżony w zwykłej klasie.....	1563

37.14.3	Szablon klasy z zagnieżdżoną definicją klasy	1565
37.15	Poradnik: jak pisać deklaracje przyjaźni w świecie szablonów	1567
37.15.1	Szablon obdarza przyjaźnią swój parametr	1573
37.16	Użytkownik sam może specjalizować szablon klas	1574
37.16.1	Kompletna (zupełna) specjalizacja szablonu klasy	1577
37.16.2	Częściowa specjalizacja szablonu klasy	1579
37.16.3	Częściowa specjalizacja pozwala wybrać parametry będące wskaźnikami	1581
37.17	Specjalizacja funkcji składowej szablonu klas	1585
37.18	Specjalizacja użytkownika szablonu funkcji	1587
37.19	Ćwiczenia	1589

38 Posłowie

1595

38.1	Per C++ ad astra	1595
------	------------------------	------

A Dodatek: Systemy liczenia

1597

A.1	Dlaczego komputer nie liczy tak jak my?	1597
A.2	System szesnastkowy (heksadecymalny)	1603
A.3	Ćwiczenia	1605

Skorowidz.....

1607



18

Deklaracje przyjaźni

Funkcja zaprzyjaźniona z klasą to funkcja, która – mimo że nie jest składnikiem tej klasy – ma dostęp do jej wszystkich (nawet prywatnych) składników.

18.1 Przyjaciele w życiu i w C++

Wyobraź sobie taką sytuację. W Twoim domu jest dużo roślin. Rośliny te są prywatnym składnikiem obiektu klasy `dom`. Pewnego dnia wyjeżdżasz na wakacje na Majorce. Chcesz jednak, by kwiatki Ci nie „zdechły”. Masz dwa wyjścia:

- ❖ Ewentualność pierwsza: sprawić, by kwiatki stały się publiczne, czyli wystawić je na klatkę schodową (kwiatki globalne). Każdy wtedy może wykonać na nich funkcję „podlewanie”. Ryzykujesz jednak, że ktoś nieproszony wykona na nich funkcję „modyfikacja”, czyli przerobi je na pokarm dla swojego królika czy węża boa. Wyjście – jeśli kochasz swoje kwiatki – nie jest dobre.
- ❖ Ewentualność druga: masz zaufanego przyjaciela. Dajesz mu klucze do swojego mieszkania i prosisz go, by podlewał kwiatki. Przyjaciel ma dostęp do wszystkich Twoich prywatnych składników (np. brylantowej koliai w komo-dzie), ale ponieważ mu ufasz, więc nie boisz się o nic. Przyjaciel przychodzi co drugi dzień i podlewa. Jak dotąd obrazek jest idylliczny.

Wścibscy sąsiedzi zauważają jednak, że ktoś obcy wchodzi do Twojego domu i dzwonią na policję, która pewnego popołudnia urządza zasadzkę – wykopuje przed drzwiami trzymetrowy dół i nakrywa go gałęziami. Przyjaciel wpada w zasadzkę. Policja zaciera ręce, że złapała złodzieja. Co prawda przyjaciel krzyczy coś z dna dołu o przyjaźni, ale nikt go nie słucha. I słusznie, prawdziwy złodziej krzychałby to samo. Nadjeżdża specjalnym wozem nadinspektor. Ocenia sytuację i mówi:

„Chwileczkę: oto mam w ręce definicję klasy pod tytułem `Dom_Czytelnika` i widzę, że na liście składników jest deklaracja, iż pana `X` uznaje się za przyjaciela `Domu_Czytelnika`. Ma on zatem prawo zrobić wszystko ze składnikami tej klasy. Proszę go więc wyciągnąć z dołu, bo jeszcze kwiatki zwiędną”.

Przełożmy ten obrazek na język pojęć C++

Konstruując klasę, ustalamy, że pewne składniki będą prywatne. Mogą więc na nich pracować funkcje składowe tej klasy. Inne nie.



W pewnych sytuacjach jednak może być korzystne, by jakaś funkcja spoza zakresu tej klasy miała także dostęp do składników prywatnych. Robi się to bardzo prosto. Wewnątrz definicji klasy wystarczy umieścić deklarację tej funkcji poprzedzoną słowem `friend`¹⁾. Dzięki temu zwykła funkcja ma prawo dostępu do prywatnych składników klasy. To tak, jakby składniki te stały się dla niej publiczne.

Ważne jest, że to nie funkcja ma twierdzić, iż jest zaprzyjaźniona. To klasa ma zadeklarować, że przyjaźni się z funkcją i tym samym nadaje jej prawo dostępu do swoich składników prywatnych. Zatem słowo `friend` pojawia się tylko wewnątrz definicji klasy.

Funkcja zaprzyjaźniona ma oczywiście także na mocy tej przyjaźni dostęp do składników `protected`.

Przykład deklaracji przyjaźni z funkcją:

Oto klasa `Tpionek`, w której jest deklaracja przyjaźni z funkcją `raport`:

```
class Tpionek
{
private:
    int kolor, pozycja;
    // dotychczasowe deklaracje
    // .....
    friend void raport(Tpionek);
};
```

Sama funkcja jest gdzieś w programie zdefiniowana następująco:

```
void raport (Tpionek p)
{
    cout << p.kolor << " pionek jest na pozycji " << p.pozycja << endl;
}
```

Funkcja `raport` wywoływana jest z argumentem typu `Tpionek`. Najważniejsze jest, że:

Wewnątrz tej zaprzyjaźnionej funkcji `raport` możemy odwoływać się do prywatnych składników obiektu klasy `Tpionek`. To tak, jakby te składniki były publiczne. To wszystko.

Jeśli chcemy, by funkcja wypisała dane o jakimś `Tpionku`, to po prostu wysyłamy go jako argument funkcji.

```
Tpionek niebieski; // definicja obiektu
...
raport(niebieski); // wywołanie funkcji
```

Może Ci się nasunąć pytanie: „Skoro chcemy, by funkcja pracowała na danych składowych klasy, to dlaczego nie zrobić z niej po prostu funkcji składowej tej klasy?”.

Pochwalam ten pomysł. Tak właśnie powinno być w tym przypadku. Lepiej mieć funkcję jako składnik, bo łatwiej wtedy nad nią panować. Tak samo jak lepiej, by to domownik podlewał kwiatki, niż przychodził w tym celu ktoś obcy.

Funkcje zaprzyjaźnione mają pewne cechy, które je wyróżniają i czynią z nich bardzo dobre narzędzie

Najważniejsza cecha to:

- 1) ang. *friend* – przyjaciel [czytaj: „frend”]

Dzięki deklaracji przyjaźni możemy nadać dostęp do prywatnych składników naszej klasy nawet takiej funkcji, która nie mogłaby być funkcją składową naszej klasy z powodów zasadniczych.

Na przykład dlatego, że:

- jest już funkcją składową innej klasy
- albo musi być funkcją globalną (np. przeładowany operator <<).

Funkcja może być przyjacielem więcej niż jednej klasy. (Tak się dzieje, gdy więcej niż jedna klasa zadeklaruje z nią przyjaźń). Wtedy taka funkcja może mieć dostęp do prywatnych składników *kilku* klas.

18.2 Przykład: dwie klasy deklarują przyjaźń z tą samą funkcją

W programie mamy dwie klasy. Klasa Tpunkt opisuje współrzędne jakiegoś punktu. Klasa Tkwadrat opisuje współrzędne lewego dolnego rogu kwadratu i długość jego boku. W obu są deklaracje przyjaźni z funkcją globalną o nazwie sedzia.



```
#include <iostream>
#include <string>
using namespace std;
//-----
class Tkwadrat;                               // deklaracja zapowiadająca ❶
////////////////////////////////////
class Tpunkt
{
    int x, y;
    string nazwa;
public:
    Tpunkt(int a, int b, string opis);
    void ruch(int n, int m)
    {
        x += n;
        y += m;
    }
    // ...może coś jeszcze...

    friend int sedzia(Tpunkt & p, Tkwadrat & k); ❷
};
////////////////////////////////////
class Tkwadrat
{
    int x, y;
    int bok;
    string nazwa;
public:
    Tkwadrat(int a, int b, int dd, string opis);
    // ...może coś jeszcze...

    friend int sedzia (Tpunkt & p, Tkwadrat & k); ❸
};
////////////////////////////////////
Tpunkt::Tpunkt(int a, int b, string opis)      // konstruktor
```

```

{
    x = a;
    y = b;
    nazwa = opis;
}
//*****
Tkwadrat::Tkwadrat(int a, int b, int dd, string opis)           // konstruktor
{
    x = a;
    y = b;
    bok = dd;
    nazwa = opis;
}
//*****
// Z tą funkcją przyjaźnią się obie klasy.
int sedzia (Tpunkt & pt, Tkwadrat & kw)                         ④
{
    if( (pt.x >= kw.x) && (pt.x <= (kw.x + kw.bok) )
        &&
        (pt.y >= kw.y) && (pt.y <= (kw.y + kw.bok) )
        )
    {
        cout << pt.nazwa << " lezy na tle " << kw.nazwa << endl;
        return 1;
    }else {
        cout << "AUT! " << pt.nazwa << " jest na zewnatrz " << kw.nazwa << endl;
        return 0;
    }
}
//*****
int main()
{
    Tkwadrat    bo(10, 10, 40, "boiska");
    Tpunkt      pi(20, 20, "pilka");

    sedzia(pi, bo);                                           ⑤
    cout << "kopiemy pilke!\n";
    while(sedzia(pi, bo))
    {
        pi.ruch(20,20);
    }
}

```



Po wykonaniu programu na ekranie zobaczymy:

```

pilka lezy na tle boiska
kopiemy pilke!
pilka lezy na tle boiska
pilka lezy na tle boiska
AUT! pilka jest na zewnatrz boiska

```



Przyjrzyjmy się ciekawszym miejscom programu

- ② Wewnątrz definicji klasy Tpunkt widzimy deklarację przyjaźni. Klasa Tpunkt stwierdza tutaj, że ma zaufanie do funkcji sedzia.

Bardzo ważna uwaga:

Zauważ, że na liście argumentów tej funkcji jest nazwa klasy Tkwadrat. Do tej pory klasa ta jeszcze nie została zdefiniowana. Pamiętamy jednak, że w C++ każda nazwa, zanim zostanie użyta po raz pierwszy, musi zostać zadeklarowana.



Jak ten problem rozwiązać?

- ❶ Oto rozwiązanie. Jest to tak zwana **deklaracja zapowiadająca** (zwiastująca). Mówi ona: „Jakby co, to nazwa Tkwadrat jest nazwą klasy”. To wszystko. Nie ma tu nic więcej na temat wewnętrznej struktury klasy Tkwadrat, ale to nie szkodzi, bo w momencie deklaracji przyjaźni te detale nie są jeszcze kompilatorowi potrzebne.
- ❷ To deklaracja przyjaźni w drugiej klasie. Podobnie: klasa Tkwadrat stwierdza tutaj, że ma zaufanie do funkcji sędzia.
- ❸ Oto definicja funkcji sędzia. Jest zdefiniowana jak najzwyklejsza funkcja.

Różnica polega tylko na tym, że funkcja ta pracuje sobie na prywatnych składnikach obiektów obu klas – tak jakby były one publiczne.

Czym zajmuje się funkcja sędzia? Łatwo się zorientować, że po prostu sprawdza, czy obiekt klasy Tpunkt leży na tle obiektu klasy Tkwadrat („czy piłka leży na boisku”). Robi to przez porównanie współrzędnych punktu z obszarem zajmowanym przez obiekt klasy Tkwadrat.

- ❹ W funkcji main korzystamy z tej funkcji. Zdefiniowaliśmy dwa obiekty i wywołujemy funkcję sędzia. Zauważ, że obiekty te wysyłamy do funkcji jako zwykłe argumenty – nie ma tu żadnego zapisu w stylu

obiekt.funkcja()

bowiem funkcja sędzia nie jest funkcją składową żadnej klasy.



18.3 W przyjaźni trzeba pamiętać o kilku sprawach

Funkcja jest zaprzyjaźniona z klasą, a nie tylko z jakimś konkretnym obiektem danej klasy. To znaczy, że funkcja zaprzyjaźniona otrzymuje prawa przyjaciela w stosunku do **wszystkich** obiektów tej klasy.

- ✧ Deklaracja przyjaźni tylko deklaruje przyjaźń – i nic więcej. Konkretnie: nazwa funkcji zaprzyjaźnionej nie staje się przez to nazwą z zakresu tej klasy.

Po prostu w deklaracji przyjaźni tylko rozmawiamy z kompilatorem, tłumacz! c mu, że taka funkcja ma dostęp...

- ✧ Funkcja zaprzyjaźniona nie jest składnikiem klasy, dlatego nie ma wskaźnika `this` do obiektów klasy, która obdarza ją przyjaźnią.

Dla nas oznacza to, że jeśli chcemy w ciele tej funkcji odnieść się do składnika jakiegoś obiektu klasy, która uznaje nas za przyjaciela, musimy powiedzieć:

- ❖ jak ten obiekt się nazywa (wtedy posługujemy się składnią *obiekt.składnik*)
- ❖ lub pokazać na niego wskaźnikiem (wtedy posługujemy się składnią *wskaźnik->składnik*).



Powtarzam więc wniosek:

Funkcja zaprzyjaźniona to zwykła funkcja, której wyjątkowo nie obowiązują słowa *private* i *protected* w klasach uznających ją za przyjaciela.

✧ Zwykle wewnątrz klasy funkcja zaprzyjaźniona jest tylko deklarowana. Jest to jedynie deklaracja przyjaźni. Nie ma znaczenia, w którym miejscu klasy (*public*, *protected*, *private*) taka deklaracja nastąpiła. Słowa *public*, *protected* i *private* nie mają na to wpływu. Przyjacielem albo się jest, albo nie jest.

✧ Może się tak zdarzyć, że kompilator (pracując nad jakimś plikiem) zobaczy deklarację pewnej funkcji po raz pierwszy dopiero w miejscu deklaracji przyjaźni. Nie jest to błąd, ale uwaga: w tym miejscu kompilator uzna, że chodzi o jakąś funkcję globalną, dostępną ogólnie – także z innych plików tego programu.

Jeśli jednak zostanie przez nas oszukany, czyli gdzieś dalej zobaczy, że definiujemy tę funkcję jako funkcję *static* (a więc widzialną tylko dla jednego konkretnego pliku), zasygnalizuje błąd.

Nie byłoby problemu, gdybyśmy wcześniej zamieścili deklarację tej funkcji jako *static*, bo wtedy przy deklaracji przyjaźni kompilator już wiedziałby, z czym ma do czynienia, i nie musiałby niczego zakładać w ciemno, a potem zmieniać zdania.

Jak to zrobić w naszym niedawnym programie?

U nas w punkcie ❷ w deklaracji przyjaźni po raz pierwszy pojawia się nazwa funkcji sędzia, nieznaną jeszcze kompilatorowi. Skoro nie było jeszcze deklaracji tej funkcji, kompilator zakłada, że chodzi o jakąś funkcję sędzia z zakresu globalnego.

Lepszą praktyką jest jednak deklarowanie wszystkich funkcji, które potem mają wystąpić w deklaracji przyjaźni.

U nas polegałoby to na postawieniu deklaracji funkcji sędzia pod liniijką ❶. Niestety w funkcji sędzia jeden z argumentów jest typu *Tpunkt*, a także ten typ jest tu jeszcze kompilatorowi nieznanym. Rozwiązanie jest proste: i on powinien mieć deklarację zapowiadającą. Łącznie więc: w programie, w miejscu ❶, dobrze by było mieć takie deklaracje:

```
class Tkwadrat;           // deklaracja zapowiadająca ❶
class Tpunkt;            // deklaracja zapowiadająca
int sędzia (Tpunkt & pt, Tkwadrat & kw); // deklaracja funkcji
```

Wiele kompilatorów nie wymaga surowo wcześniejszych deklaracji funkcji, które mają zostać potem obwołane przyjaciółmi, i wybaczy nam ich brak. No ale według standardu C++ te deklaracje powinny jednak być.

Przyjaciel-rezydent, czyli: funkcja zaprzyjaźniona „goszcząca” w klasie

Możemy tak zrobić, że wewnątrz klasy jest nie tylko deklaracja funkcji zaprzyjaźnionej, ale wręcz jej definicja (czyli całe ciało funkcji). Mimo że jest ona umieszczona „w środku” ciała klasy, funkcja jest nadal tylko przyjacielem, a nie składnikiem.

Taka definicja funkcji zaprzyjaźnionej ma następujące konsekwencje:

- ❖ funkcja zaprzyjaźniona jest typu *inline*,
- ❖ funkcja leży w zakresie *leksykalnym* deklaracji tej klasy; oznacza to, że można w definicji tej zaprzyjaźnionej funkcji:

18.4 Obdarzenie przyjaźnią funkcji składowej innej klasy

Funkcja zaprzyjaźniona może być zwykłą funkcją, a może być też funkcją składową zupełnie innej klasy.

Oto tak zmodyfikowany poprzedni przykład, że funkcja `sedzia` jest składnikiem klasy `Tkwadrat`, a klasa `Tpunkt` deklaruje z tą funkcją przyjaźń.



```
#include <iostream>
#include <string>
using namespace std;
//-----
class Tpunkt;           // deklaracja zapowiadająca           ❶
/////////////////////////////////////////////////////////////////
class Tkwadrat
{
    int x, y;
    int bok;
    string nazwa;
public:
    Tkwadrat(int a, int b, int dd, string opis);
    // ...może coś jeszcze...

    int sedzia (Tpunkt & p);           ❷
};
/////////////////////////////////////////////////////////////////
class Tpunkt
{
    int x, y;
    string nazwa;
public:
    Tpunkt(int a, int b, string opis);
    void ruch(int n, int m) {
        x += n;
        y += m;
    }
    // ...może coś jeszcze...

    friend int Tkwadrat::sedzia(Tpunkt & p);           ❸
};
/////////////////////////////////////////////////////////////////
Tpunkt::Tpunkt(int a, int b, string opis)           // konstruktor
{
    x = a;
    y = b;
    nazwa = opis;
}
//*****
Tkwadrat::Tkwadrat(int a, int b, int dd, string opis)           // konstruktor
{
    x = a;
    y = b;
```

```

    bok = dd;
    nazwa = opis;
}
//*****
int Tkwadrat::sedzia (Tpunkt & pt)           ❹
{
    if( (pt.x >= x) && (pt.x <= (x + bok) )   ❺
        &&
        (pt.y >= y) && (pt.y <= (y + bok) )
    )
    {
        cout << pt.nazwa << " lezy na tle " << nazwa << endl;
        return 1;
    }
    else {
        cout << "AUT! " << pt.nazwa << " jest na zewnatrz " << nazwa << endl;
        return 0;
    }
}
//*****
int main()
{
    Tkwadrat    bo(10,10, 40, "boiska");
    Tpunkt      pi( 20, 20, "pilka");
    bo.sedzia(pi);                               ❻
}

```



Po wykonaniu programu na ekranie pojawi się:

pilka lezy na tle boiska



Komentarz

- ❷ To jest deklaracja zwykłej funkcji składowej w klasie Tkwadrat. Argumentem jest obiekt klasy Tpunkt, stąd też konieczna była deklaracja zapowiadająca tę klasę ❶.
- ❸ To deklaracja przyjaźni. Tutaj jednak ważna uwaga. Jeśli przyjacielem ma być *funkcja składowa z innej klasy*, to ta klasa musi być już w tym momencie znana kompilatorowi. Dlatego najpierw w programie umieszczona jest deklaracja klasy Tkwadrat (z funkcją sędzia), a potem dopiero deklaracja klasy Tpunkt i niniejsze ogłoszenie przyjaźni.
- ❹ Oto definicja funkcji sędzia. Na pewno już przy deklaracjach zauważyłeś, że zmieniła się lista argumentów. Teraz argumentem jest tylko obiekt klasy Tpunkt. A co z obiektem klasy Tkwadrat, funkcja go przecież także potrzebuje?! Zapominasz, że teraz funkcja jest funkcją składową klasy Tkwadrat, a więc jest wywoływana na rzecz obiektu klasy Tkwadrat. Zresztą spójrz poniżej.
- ❺ Tak właśnie w main wywołujemy funkcję sędzia. Obiekt klasy Tpunkt wysyłany jest jako argument, a obiekt klasy Tkwadrat – przez ukryty wskaźnik this.
- ❻ Z faktu, iż funkcja sędzia jest funkcją składową klasy Tkwadrat, wynika, że odnosząc się do danej składowej swojej klasy, można posługiwać się zapisem: *składnik*, a nie zapisem: *obiekt.składnik* – widać to wyraźnie w tej linijce.
W stosunku do składników obiektu klasy zaprzyjaźnionej Tpunkt stosujemy tu zapis pt.x, ale w stosunku do składników swojej klasy: x, bok. Przedtem w tym miejscu było konieczne kw.x, kw.bok.

To dlatego, że przecież gdy piszemy `x`, to jest tam naprawdę `this->x`.

„*Coakrêcisz!*” – zawo³a³eczapewne – „*parê stron wczeaniej wmwia³eæmi, æe funkcja zaprzyjaŹniona z klas¹ nie zawiera wskaŹnika this!*”.

Podtrzymuję to!

|| Funkcja `F` zaprzyjaźniona z klasą `K` nie zawiera wskaźnika `this` do klasy `K`, która uznaje ją za przyjaciela, bo nie jest składnikiem tej klasy.

Jeśli jednak sama funkcja `F` jest zwykłą funkcją składową jakiejś innej klasy, to zawiera wskaźnik `this` do obiektu *swojej* klasy. Za jego pomocą pracuje przecież na swoich składnikach.

Posłużmy się analogią do podlewania kwiatków. Załóżmy, że to Ty jesteś osobą podlewającą kwiatki i Twoja znajoma Perfidia zadeklarowała z Tobą przyjaźń.

Gdy określasz swoje czynności, to mówisz: „Idę podlać kwiatki Perfidii”. Jak do tej pory rzeczywistość nie ma wskaźnika `this`. Mówisz przecież o składnikach tych klas `Perfidia.kwiatki`.

Teraz uwaga: okazuje się, Czytelniku, że dostałeś mieszkanie i nie mieszkasz już więcej pod mostem. Nie jesteś już funkcją globalną, tylko należysz do klasy pod nazwą „`mój_dom`”. Załóżmy, że Perfidia deklaruje Cię nadal jako swojego przyjaciela.

Mówisz: podlewam „kwiatki Perfidii” (podlewam `Perfidia.kwiatki`).

Możesz jednak powiedzieć też: „podlewam kwiatki”, myśląc o podlewaniu *swoich* kwiatków

kwiatki *czyli* `this->kwiatki`

gdzie `this` oznacza wskaźnik do obiektu „`mój_dom`”.



Jeśli projektujesz program i widzisz, że przydałoby się, żeby funkcja miała dostęp do składników prywatnych dwóch klas, to masz do wyboru jedno z rozwiązań:

obie klasy deklarują tę funkcję (globalną) jako zaprzyjaźnioną,
funkcja jest składnikiem jednej klasy, a druga klasa deklaruje ją jako funkcję zaprzyjaźnioną.

Który z wariantów wybrać, decydujesz, rozważając wspomniane zalety funkcji zaprzyjaźnionej z cechami funkcji składowej. O podejmowaniu takich wyborów porozmawiamy jeszcze w jednym z następnych rozdziałów. („Przeładowanie operatorów” – str. 956).

18.5 Klasy zaprzyjaźnione

Klasa `K` może deklarować przyjaźń z więcej niż jedną funkcją składową klasy `M`. Może nawet deklarować przyjaźń ze wszystkimi funkcjami klasy `M`. Jest to trochę tak, jakbyś wytrwale zadeklarował przyjaźń klasy `K` z każdą funkcją składową klasy `M`. Łatwo te deklaracje zrobić, ale wymaga to dużo pisania.



Zamiast tego możemy zadeklarować, że klasa `K` uznaje za przyjaciela *całą* klasę `M`.

```
class K
{
    friend class M;
    // ...
};
```

Od tej pory *wszystkie* funkcje składowe klasy M mają dostęp do prywatnych składników klasy K deklarującej tę przyjaźń. To już Cię nie dziwi – przyzwyczyłeś się do tego przy okazji funkcji zaprzyjaźnionych. Jest tu jednak coś jeszcze, coś, czego przy funkcjach być nie mogło. Załóżmy, że mamy zwykłą klasę K, która deklaruje przyjaźń z klasą PRZYJACIEL.



Klasa PRZYJACIEL może używać składników prywatnych klasy K nie tylko w swych funkcjach składowych, ale **także przy inicjalizacji swych składników, nawet tych statycznych**. Jak pewnie jeszcze pamiętasz, ich definicje są umieszczane osobno, jakby na zewnątrz definicji klasy PRZYJACIEL.

|| Nawet więc tam, jakby na zewnątrz ciała klasy PRZYJACIEL, przyjaciel może skorzystać z wartości prywatnego składnika klasy K.

✧ Jeśli klasa K ma w sobie jakieś definicje typów enum lub typedef czy using, to klasa PRZYJACIEL też może z nich skorzystać przy deklaracji swoich składników.

✧ Mówiliśmy o tym, że definicję *funkcji* zaprzyjaźnionej z klasą K można umieścić nawet w samym miejscu deklaracji przyjaźni, czyli wewnątrz klasy K (funkcja: przyjaciel – rezydent, str. 701).

|| Jednak z klasą-przyjacielem tej sztuczki zrobić się nie da. Ta klasa-przyjaciel musi mieć swoją definicję gdzieś na zewnątrz.

Deklaracja przyjaźni jest oczywiście jednostronna

Wyraża ją klasa K wobec klasy PRZYJACIEL i już. Natomiast klasa PRZYJACIEL nie wyraża niczego szczególnego w stosunku do klasy K. Konkretnie – wcale jej nie upoważnia do grzebania w swoich składnikach prywatnych.

Dwie klasy mogą się przyjaźnić także z wzajemnością

Jedyną możliwością zadeklarowania takiej przyjaźni jest właśnie deklaracja przyjaźni z klasą jako całością sposobem, jaki pokazaliśmy.

|| Nie ma możliwości zadeklarowania w jednej klasie, że przyjaźni się ona z funkcjami innej klasy, a w tej innej klasie – że przyjaźni się z wybranymi funkcjami klasy pierwszej.

To z powodu, o którym już wspomnieliśmy: jeśli deklarujemy przyjaźń z funkcją, która jest funkcją składową innej klasy, to kompilator życzy sobie już znać deklarację tej klasy (na przykład, żeby sprawdzić, czy taka funkcja rzeczywiście tam jest).

Żebyśmy się nie wiem jak gimnastykowali, to zawsze definicja jednej klasy będzie znana wcześniej od drugiej, bo tak przecież piszemy tekst programu. Siłą rzeczy klasa, która definiowana jest wcześniej, musiałaby zawierać w sobie deklaracje przyjaźni z funkcjami składowymi klasy drugiej, chwilowo jeszcze nieznanymi. (Sama deklaracja zapowiadająca klasę nie wystarcza kompilatorowi – musi on znać wnętrze klasy).

Nie ma problemu. Wyjście z tego błędnego koła załatwia nam deklaracja przyjaźni z całą klasą.

```
class Tdruga;           // deklaracja zapowiadająca
```

```
class Tpierwsza {
    friend class Tdruga;
    // ...reszta ciała klasy pierwszej
```

```
};

class Tdruga {
    friend class Tpierwsza;
    // ...reszta ciała klasy drugiej
};
```

Przyjaźń nie jest przechodnia

|| Przyjaciel mojego przyjaciela nie jest moim przyjacielem.

Inaczej mówiąc: jeśli klasa A deklaruje przyjaźń z klasą B, natomiast klasa B deklaruje przyjaźń z klasą C, to wcale nie oznacza, że klasa A uznaje klasę C za swojego przyjaciela.

Gdyby o to chodziło, to należałoby w klasie A zamieścić deklarację takiej przyjaźni:
friend class C;

Przechodniość przyjaźni byłaby bardzo niebezpieczna. Zresztą w życiu także się nią nie posługujemy.

Przyjaźń nie jest dziedziczna

|| Przyjaciel mojej prababki nie jest moim przyjacielem.

👑 *Wtajemniczeni wiedzą, że klasa może mieć „potomstwo” (klasy pochodne). Przyjaźń nie jest dziedziczna – jeśli jakaś klasa chce mieć przyjaciela, to powinna to powiedzieć wyraźnie sama.*

W deklaracji przyjaźni nie mogą pojawić się przydomki (specyfikatory)...

✦ ...określające sposób, w jaki przyjaciel został (przez kompilator) umieszczony w pamięci. Te niedozwolone przydomki to `static`, `register`, `extern`, `thread_local` i `mutable`.

W życiu codziennym jest podobnie. Wypada powiedzieć „Przyjaźnię się z Tomaszem”, a nie wypada powiedzieć „Przyjaźnię się z Tomaszem, bo ma willę z basenem” albo „Przyjaźnię się z Tomaszem, bo mieszka za granicą”.

18.6 Konwencja umieszczania deklaracji przyjaźni w klasie

Spotyka się często konwencję definiowania klasy w taki sposób, że najpierw w definicji klasy wyszczególnia się wszystkie składniki publiczne (czyli widziane z zewnątrz klasy). Dopiero dalej występują składniki prywatne, czyli takie, o których zwykły użytkownik klasy nie musi już wiedzieć. (Używając pralki automatycznej, użytkownik nie musi wiedzieć o wszystkich jej elementach elektronicznych i mechanicznych).

Funkcje zaprzyjaźnione są tym, co powinno się od razu zauważyć, patrząc na definicję klasy, więc przyjęło się umieszczać je na samym początku, na samej górze definicji klasy.

Jak mówię – jest to tylko konwencja, która może czasem ułatwić „czytanie” definicji klas.

18.7 Kilka otrzeźwiających słów na zakończenie

Poznaliśmy tu nowe narzędzie pozwalające na dostęp do schowanych składników klasy. Nie daj się jednak ponieść. Przyjaźń jest przecież naruszeniem czegoś, z czego

jesteśmy bardzo dumni, czyli schowania części danych w klasie. Schowania po to, żebyśmy w przypadku gdy program „chodzi źle”, nie martwili się: „Któż to zmienił mi wartość tego składnika bez mojej wiedzy...”. Im mniej przyjaciół, tym łatwiej panować nad działaniem danej klasy.

Zatem szalu nie ma, przesadzanie z przyjaźnią jest złą praktyką.

Naprawdę więc deklaracje przyjaźni będziemy stosowali głównie w sytuacjach:

- ❖ gdy będziemy chcieli, aby obiekt `cout` mógł wypisywać na ekranie treść składników obiektu naszej klasy (obdarzymy wtedy przyjaźnią klasę `ostream`),
- ❖ gdy będziemy chcieli, żeby na obiektach naszej klasy mogły sprawnie pracować globalne funkcje tzw. operatorowe (poznamy je na str. 956).

18.8 Ćwiczenia

I Przyjaźń polega na tym, że dana klasa `K` udziela zezwolenia innej funkcji/klasie:

- a) na dostęp do pracy z obiektami klasy `K`,
- b) na modyfikację jej składników w klasie `K`,
- c) na dostęp do składników niepublicznych w obiektach klasy `K`,
- d) na dostęp do składników niepublicznych w wyznaczonych obiektach klasy `K`.

II Klasa `K` oznajmia przyjaźń z funkcją/klasą `P`. Słowo `friend` umieszczone jest:

- a) w deklaracji funkcji/klasy `P` uznanej za przyjaciela,
- b) w klasie `K`, która ogłasza przyjaźń z funkcją/klasą `P`,
- c) w klasie/funkcji `P`, w instrukcjach, które odnoszą się do prywatnych składników klasy `K`.

III Przyjaźń może deklarować:

- a) klasa, b) funkcja globalna, c) funkcja składowa, d) każde z wymienionych.

IV Co może być przyjacielem?

- a) funkcja globalna, c) funkcja składowa,
- b) funkcja statyczna, d) klasa.

V Jedna funkcja może być przyjacielem

- a) tylko jednej klasy, b) nawet wielu klas.

VI Czy w deklaracji przyjaźni wyrażonej w klasie `K` wobec funkcji `f`, funkcja ta musi być wcześniej zadeklarowana?


VII Jeśli funkcja-przyjaciel klasy `K` odnosi się do składników klasy `K`, to skąd wiadomo, którego obiektu klasy `K` to dotyczy?

- a) działa to na wszystkie obiekty klasy `K`,
- b) funkcja musi powiedzieć, o który konkretny obiekt jej chodzi.

VIII Przyjaciel klasy `K` otrzymuje prawo dostępu do składników niepublicznych klasy `K`

- a) we wszystkich obiektach klasy `K`,
- b) w wybranym obiekcie klasy `K`.

IX Jakie ma konsekwencje fakt, że funkcja-przyjaciel klasy `K` ma swoją definicję dołączoną do deklaracji przyjaźni w klasie `K` („funkcja-rezydent”)?

X  W lokalnej klasie `KL` umieszczamy deklarację przyjaźni z funkcją `f` i równocześnie definicję tej funkcji `f` („funkcja-rezydent”). Czy funkcja `f` może zwracać rezultat będący obiektem typu `KL`?

- XI** Klasa K deklaruje przyjaźń z funkcją składową klasy P. Jaka deklaracja klasy P musi poprzedzać definicję tej klasy?
a) wystarczy deklaracja zapowiadająca P, b) konieczna jest definicja klasy P.
- XII** Wybierz wszystkie poprawne zakończenia następującego zdania: Klasa K deklaruje przyjaźń z funkcją składową klasy P; ta funkcja składowa:
a) jest wywoływana na rzecz obiektu klasy K, która obdarzyła ją przyjaźnią,
b) jest wywoływana na rzecz obiektu swojej klasy P,
c) może pracować na składnikach swego obiektu klasy P,
d) może pracować na obiekcie klasy K pod warunkiem, że wie na którym.
- XIII** Klasa K deklaruje przyjaźń z klasą P. Czy funkcja składowa klasy P otrzymuje wskaźnik `this` pozwalający jej pracować na składnikach klasy K?
- XIV** Co to znaczy, że klasa K deklaruje przyjaźń z klasą P?
- XV** Co to znaczy, że przyjaźń nie jest przechodnia?
- XVI** Czy przyjaźń jest wzajemna?
- XVII** Gdzie należy umieszczać deklarację przyjaźni w klasie K. W jej części `public` czy `private`?
- XVIII** Funkcja składowa klasy P ma przydomek `static`. W klasie K chcemy zamieścić deklarację przyjaźni z tą funkcją. Gdzie umieszczamy ten przydomek?
a) po słowie `friend`, a przed typem rezultatu,
b) na samym końcu deklaracji.



PROGRAM PARTNERSKI

— GRUPY HELION —



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

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

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

GRUPA
Helion 

OPUS MAGNUM

C++

Programowanie w języku C++

Dzięki tej książce poznasz:

- Proste i złożone typy danych
- Instrukcje sterujące
- Funkcje i operatory
- Wskaźniki
- Klasy i dziedziczenie
- Obsługę wyjątków
- Wyrażenia lambda
- Operacje wejścia-wyjścia
- Projektowanie zorientowane obiektowo
- Szablony

Jeżeli chcesz uczyć się języka C++ w łatwy i przyjazny sposób, ta książka jest dla Ciebie!

Jedno C i same plusy!

Dawno, dawno temu, w głębokich latach osiemdziesiątych ubiegłego wieku, pewien duński informatyk, zainspirowany językiem C, opracował jeden z najważniejszych, najbardziej elastycznych i do dziś niezastąpionych języków programowania – C++. Dziś jest on wykorzystywany do tworzenia gier komputerowych, obliczeń naukowych, technicznych, w medycynie, przemyśle i bankowości. NASA posługuje się nim w naziemnej kontroli lotów. Duża część oprogramowania Międzynarodowej Stacji Kosmicznej została napisana w tym języku. Nawet w marsjańskim łaziku Curiosity pracuje program w C++, który analizuje obraz z kamer i planuje dalszą trasę.

Autor tej książki – wybitny specjalista pracujący nad wieloma znaczącymi projektami we francuskich, niemieckich i włoskich instytutach fizyki jądrowej, znany czytelnikom m.in. z genialnej *Symfonii C++* – postawił sobie za cel napisanie nowej, przekrojowej publikacji o tym języku, która w prostym, obrazowym stylu wprowadza czytelnika w fascynujący świat programowania zorientowanego obiektowo. Zobacz, jak potężny jest C++.

Patroni:



programista



Helion 