Jerzy Grębosz

Symfonia C++

Programowanie w języku C++ orientowane obiektowo

Łatwy podręcznik

Jeśli szukasz nowoczesnego języka programowania, to jest to właśnie język C++.
Jeśli chcesz nauczyć się tego języka w łatwy, pogodny sposób - to jest to właśnie książka dla Ciebie.

Symfonia

C++

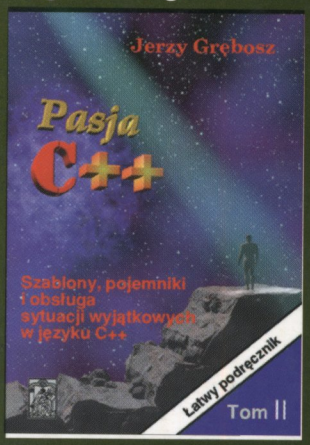
ISBN 83-901689-1-X

Książka ta powstała na podstawie doświadczeń autora w pracy programisty w Hahn-Meitner-Institut w Berlinie (dawniej Zachodnim) i nosi znamiona książek pisanych na Zachodzie:
prostym, wręcz przyjacielskim stylem wprowadza czytelnika w fascynujący świat programowania obiektowo orientowanego.

Dodatkową zaletą książki jest to, że porusza zagadnienie najczęściej pomijane przez innych autorów:
Jak stosować technikę obiektowo orientowaną przy projektowaniu własnego programu.

Autor podjął się zadania karkołomnego - Napisał książkę, która może być czytana zarówno przez zawodowego informatyka jak i przez programistę-amatora znającego tylko język BASIC

Tego samego autora:



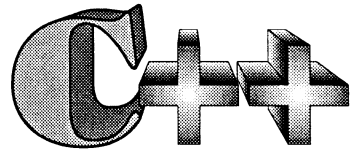
Pasja **C++**

Szablony, pojemniki i obsługa sytuacji wyjątkowych w języku C++

2 tomy, 610 stron

Oficyna Kallimach

Symfonia



Jerzy Grębosz

Symfonia C++

Programowanie w języku C++ orientowane obiektowo

Tom I

Oficyna Kallimach
Kraków 1999

All of the products and software mentioned in this book
are registered trademarks of their owners

Opracowanie graficzne: Jerzy Grębosz

Copyright: Jerzy Grębosz

Wszelkie prawa zastrzeżone

ISBN 83-901689-1-X

Wydanie czwarte uzupełnione

Po informacje na temat tłumaczeń książki
lub w sprawie zakupu hurtowego, zniżek dla studentów, zniżek promocyjnych
prosimy kontaktować się z Oficyną Kallimach.

Oficyna Kallimach realizuje sprzedaż wysyłkową bez naliczania kosztów przesyłki.
Zamówienia telefoniczne i pisemne.

Wydawnictwo: Oficyna Kallimach

Adres do korespondencji:
Elektor Ltd.
30-054 Kraków, Czarnowiejska 72
fax: (0 - 12) 637 18 81
email: jerzy.grebosz@ifj.edu.pl

Druk i oprawa: ZG "Colonel" S.c. 30-532 Kraków, ul. Dąbrowskiego 16

Spis treści

(Wszystkich trzech tomów)

Tom I

0 Proszę nie czytać tego ! 1

1 Startujemy ! 6

1.1 Pierwszy program 6
1.2 Drugi program 10

2 Instrukcje sterujące 15

2.1 Prawda – Fałsz 15
2.2 Instrukcja warunkowa `if` 16
2.3 Instrukcja `while` 18
2.4 Pętla `do...while...` 19
2.5 Pętla `for` 20
2.6 Instrukcja `switch` 22
2.7 Instrukcja `break` 24
2.8 Instrukcja `goto` 25
2.9 Instrukcja `continue` 27
2.10 Klamry w instrukcjach sterujących 28

3 Typy 30

3.1 Deklaracje typów 30
3.2 Systematyka typów z języka C++ 32
3.3 Typy fundamentalne 32
3.3.1 Definiowanie obiektów „w biegu". 34
3.4 Stałe dosłowne 35
3.4.1 Stałe będące liczbami całkowitymi 35
3.4.2 Stałe reprezentujące liczby zmiennoprzecinkowe 36
3.4.3 Stałe znakowe 37
3.4.4 Stałe tekstowe, albo po prostu stringi 39
3.5 Typy pochodne 40
3.5.1 Typ `void` 42

3.6 Zakres ważności nazwy obiektu, a czas życia obiektu ... 42
3.6.1 Zakres: lokalny ... 42
3.6.2 Zakres: blok funkcji ... 43
3.6.3 Zakres: obszar pliku ... 43
3.6.4 Zakres: obszar klasy ... 44
3.7 Zasłanianie nazw ... 44
3.8 Modyfikator `const` ... 45
3.8.1 Pojedynek: `const` contra `#define` ... 47
3.9 Obiekty `register` ... 48
3.10 Modyfikator `volatile` ... 49
3.11 Instrukcja `typedef` ... 50
3.12 Typy wyliczeniowe `enum` ... 52

4 Operatory ... 54
4.1 Operatory arytmetyczne ... 54
4.1.1 Operator % czyli modulo ... 55
4.1.2 Jednoargumentowe operatory + i - ... 57
4.1.3 Operatory inkrementacji i dekrementacji ... 57
4.1.4 Operator przypisania = ... 59
4.2 Operatory logiczne ... 60
4.2.1 Operatory relacji ... 60
4.2.2 Operatory sumy logicznej | | i iloczynu logicznego && ... 61
4.2.3 Operator negacji: ! ... 63
4.3 Operatory bitowe ... 63
4.3.1 Przesunięcie w lewo << ... 64
4.3.2 Przesunięcie w prawo >> ... 65
4.3.3 Bitowe operatory sumy, iloczynu, negacji, różnicy symetrycznej ... 66
4.4 Różnica między operatorami logicznymi, a operatorami bitowymi ... 66
4.5 Pozostałe operatory przypisania ... 67
4.6 Wyrażenie warunkowe ... 68
4.7 Operator `sizeof` ... 69
4.8 Operator rzutowania ... 70
4.9 Operator: przecinek ... 71
4.10 Priorytety operatorów ... 71
4.11 Łączność operatorów ... 74

5 Funkcje ... 75
5.1 Zwracanie rezultatu przez funkcję ... 78
5.2 Stos ... 80
5.3 Przesyłanie argumentów do funkcji przez wartość ... 81
5.4 Przesyłanie argumentów przez referencję ... 83
5.5 Kiedy deklaracja funkcji nie jest konieczna ... 86
5.6 Argumenty domniemane ... 87
5.7 Nienazwany argument ... 90
5.8 Funkcje `inline` (w linii) ... 91
5.9 Przypomnienie o zakresie ważności nazw deklarowanych wewnątrz funkcji ... 95
5.10 Wybór zakresu ważności nazwy i czasu życia obiektu ... 96
5.10.1 Obiekty globalne ... 96

5.10.2 Obiekty automatyczne ... 97
5.10.3 Obiekty lokalne statyczne ... 98
5.11 Funkcje w programie składającym się z kilku plików ... 102
5.11.1 Nazwy statyczne globalne ... 106
5.12 Funkcje biblioteczne ... 107

6 Preprocesor ... 110

6.1 Na pomoc rodakom ... 110
6.2 Dyrektywa `#define` ... 112
6.3 Dyrektywa `#undef` ... 115
6.4 Makrodefinicje ... 115
6.5 Dyrektywy kompilacji warunkowej ... 118
6.6 Dyrektywa `#error` ... 121
6.7 Dyrektywa `#line` ... 122
6.8 Wstawianie treści innych plików w tekst kompilowanego właśnie pliku ... 123
6.9 Sklejacz czyli operator ## ... 124
6.10 Dyrektywa pusta ... 125
6.11 Dyrektywy zależne od implementacji ... 125
6.12 Nazwy predefiniowane ... 125

7 Tablice ... 128

7.1 Elementy tablicy ... 129
7.2 Inicjalizacja tablic ... 131
7.3 Przekazywanie tablicy do funkcji ... 132
7.4 Tablice znakowe ... 136
7.5 Tablice wielowymiarowe ... 144
7.5.1 Przesyłanie tablic wielowymiarowych do funkcji ... 147

8 Wskaźniki ... 149

8.1 Wskaźniki mogą bardzo ułatwić życie ... 149
8.2 Definiowanie wskaźników ... 151
8.3 Praca ze wskaźnikiem ... 152
8.4 L-wartość ... 155
8.5 Wskaźniki typu `void` ... 156
8.6 Cztery domeny zastosowania wskaźników ... 159
8.7 Zastosowanie wskaźników wobec tablic ... 159
8.7.1 Ćwiczenia z mechaniki ruchu wskaźnika ... 159
8.7.2 Użycie wskaźnika w pracy z tablicą ... 163
8.7.3 Arytmetyka wskaźników ... 167
8.7.4 Porównywanie wskaźników ... 169
8.8 Zastosowanie wskaźników w argumentach funkcji ... 172
8.8.1 Jeszcze raz o przesyłaniu tablic do funkcji ... 175
8.8.2 Odbieranie tablicy jako wskaźnika ... 176
8.8.3 Argument formalny będący wskaźnikiem do obiektu `const` ... 178
8.9 Zastosowanie wskaźników przy dostępie do konkretnych komórek pamięci ... 181
8.10 Rezerwacja obszarów pamięci ... 181
8.10.1 Operatory `new` i `delete` albo Oratorium Stworzenie Świata. ... 182
8.10.2 Dynamiczna alokacja tablicy ... 186
8.10.3 Zapas pamięci to nie jest studnia bez dna ... 189
8.10.4 Porównanie starych i nowych sposobów ... 191

8.11 Stałe wskaźniki ... 192
8.12 Stałe wskaźniki, a wskaźniki do stałych ... 193
8.13 Strzał na oślep – Wskaźnik zawsze pokazuje na coś ... 194
8.14 Sposoby ustawiania wskaźników ... 196
8.15 Tablice wskaźników ... 197
8.16 Wariacje na temat stringów ... 199
8.17 Wskaźniki do funkcji ... 206
8.17.1 Ćwiczenia z definiowania wskaźników do funkcji ... 209
8.17.2 Wskaźnik do funkcji jako argument innej funkcji ... 216
8.17.3 Tablica wskaźników do funkcji ... 220
8.18 Argumenty z linii wywołania programu ... 223

9 Przeładowanie nazw funkcji ... 227

9.1 Co to znaczy: przeładowanie ... 227
9.2 Bliższe szczegóły przeładowania ... 231
9.3 Czy przeładowanie nazw funkcji jest techniką obiektowo orientowaną? ... 233
9.4 Linkowanie z modułami z innych języków ... 235
9.5 Przeładowanie a zakres ważności deklaracji funkcji ... 236
9.6 Rozważania o identyczności lub odmienności typów argumentów ... 238
9.6.1 Przeładowanie a typedef i enum ... 238
9.6.2 Tablica a wskaźnik ... 239
9.6.3 Pewne szczegóły o tablicach wielowymiarowych ... 240
9.6.4 Przeładowanie a referencja ... 242
9.6.5 Identyczność typów: T, const T, volatile T ... 243
9.6.6 Przeładowanie a typy: T*, volatile T*, const T* ... 244
9.6.7 Przeładowanie a typy: T&, volatile T&, const T& ... 245
9.7 Adres funkcji przeładowanej ... 246
9.7.1 Zwrot rezultatu będącego adresem funkcji przeładowanej ... 248
9.8 Kulisy dopasowywania argumentów do funkcji przeładowanych ... 250
9.9 Etapy dopasowania ... 251
9.9.1 Etap 1. Dopasowanie dosłownie ... 252
9.9.2 Etap 2. Dopasowanie dosłowne, ale z tzw. trywialną konwersją ... 252
9.9.3 Etap 3. Dopasowanie z awansem ... 253
9.9.4 Etap 4. Próba dopasowania za pomocą konwersji standardowych ... 254
9.9.5 Etap 5. Próba dopasowania z użyciem konwersji zdefiniowanych przez użytkownika. ... 256
9.9.6 Etap 6. Próba dopasowania do funkcji z wielokropkiem ... 256
9.10 Dopasowywanie wywołań z kilkoma argumentami ... 256

Tom II

10 Klasy ... 259

10.1 Typy definiowane przez użytkownika ... 259
10.2 Składniki klasy ... 261
10.3 Składnik będący obiektem ... 263
10.4 Enkapsulacja ... 263
10.5 Ukrywanie informacji ... 264
10.6 Klasa a obiekt ... 267

10.7 Funkcje składowe ... 270
10.7.1 Posługiwanie się funkcjami składowymi ... 270
10.7.2 Definiowanie funkcji składowych ... 271
10.8 Jak to właściwie jest ? (`this`) ... 276
10.9 Odwołanie się do publicznych danych składowych ... 277
10.10 Zasłanianie nazw ... 278
10.11 Przeładowanie i zasłonięcie równocześnie ... 281
10.12 Przesyłanie do funkcji argumentów będącymi obiektami ... 282
10.12.1 Przesyłanie obiektu przez wartość ... 282
10.12.2 Przesyłanie przez referencję ... 285
10.13 Konstruktor – pierwsza wzmianka ... 286
10.14 Destruktor – pierwsza wzmianka ... 291
10.15 Składnik statyczny ... 295
10.16 Statyczna funkcja składowa ... 299
10.17 Do czego może nam się przydać składnik statyczny w klasie? ... 302
10.18 Funkcje składowe typu `const` oraz `volatile` ... 303
10.18.1 Przeładowanie a funkcje składowe `const` i `volatile` ... 306

11 Funkcje zaprzyjaźnione ... 307

12 Struktury, Unie, Pola bitowe ... 318

12.1 Struktura ... 318
12.2 Unia ... 319
12.2.1 Inicjalizacja unii ... 321
12.2.2 Unia anonimowa ... 321
12.3 Pola bitowe ... 323

13 Zagnieżdżona definicja klasy ... 328

13.1 Lokalna definicja klasy ... 331
13.2 Lokalne nazwy typów ... 334

14 Konstruktory i Destruktory ... 336

14.1 Konstruktor ... 336
14.1.1 Przykład programu zawierającego klasę z konstruktorami ... 337
14.2 Kiedy i jak wywoływany jest konstruktor ... 343
14.2.1 Konstruowanie obiektów lokalnych ... 343
14.2.2 Konstruowanie obiektów globalnych ... 343
14.2.3 Konstrukcja obiektów tworzonych operatorem `new` ... 344
14.2.4 Jawne wywołanie konstruktora ... 345
14.2.5 Dalsze sytuacje, gdy pracuje konstruktor ... 346
14.3 Destruktor ... 347
14.4 Konstruktor domniemany ... 349
14.5 Lista inicjalizacyjna konstruktora ... 350
14.6 Konstrukcja obiektu, którego składnikiem jest obiekt innej klasy ... 353
14.7 Konstruktory nie-publiczne ? ... 359
14.8 Konstruktor kopiujący (albo inicjalizator kopiujący) ... 361
14.8.1 Przykład klasy z konstruktorem kopiującym ... 363
14.8.2 Konstruktor kopiujący gwarantujący nietykalność ... 370
14.8.3 Współodpowiedzialność ... 371
14.8.4 Konstruktor kopiujący generowany automatycznie ... 372

14.8.5 Kiedy konstruktor kopiujący jest niezbędny? ... 372

15 Tablice obiektów ... 377

15.1 Tablica obiektów definiowana operatorem `new` ... 379
15.2 Inicjalizacja tablic obiektów ... 380
15.2.1 Inicjalizacja tablic obiektów będących agregatami ... 380
15.2.2 Inicjalizacja tablic nie będących agregatami ... 383
15.2.3 Inicjalizacja tablic tworzonych w zapasie pamięci ... 386

16 Wskaźnik do składników klasy ... 388

16.1 Wskaźniki zwykłe - repetytorium ... 388
16.2 Wskaźnik do pokazywania na składnik-daną ... 390
16.3 Wskaźnik do funkcji składowej ... 394
16.4 Tablica wskaźników do danych składowych klasy ... 396
16.5 Tablica wskaźników do funkcji składowych klasy ... 397
16.6 Wskaźniki do składników statycznych ... 398

17 Konwersje ... 399

17.1 Sformułowanie problemu ... 399
17.2 Konstruktor jako konwerter ... 401
17.3 Funkcja konwertująca – operator konwersji ... 404
17.4 Który wariant konwersji wybrać ? ... 410
17.5 Sytuacje, w których zachodzi konwersja ... 412
17.6 Zapis jawnego wywołania konwersji typów ... 413
17.6.1 Advocatus zapisu przypominającego: „wywołanie funkcji" ... 414
17.6.2 Advocatus zapisu: „rzutowanie" ... 414
17.7 Niecałkiem dobrane małżeństwa, czyli konwersje przy dopasowaniu ... 415
17.8 Kilka rad dotyczących konwersji ... 420

18 Przeładowanie operatorów ... 422

18.1 Przeładowanie operatorów – definicja i trochę teorii ... 424
18.2 Moje zabawki ... 428
18.3 Funkcja operatorowa jako funkcja składowa ... 430
18.4 Funkcja operatorowa nie musi być przyjacielem klasy ... 433
18.5 Operatory predefiniowane ... 434
18.6 Argumentowość operatorów ... 434
18.7 Operatory jednoargumentowe ... 435
18.8 Operatory dwuargumentowe ... 438
18.8.1 Przykład na przeładowanie operatora dwuargumentowego ... 438
18.8.2 Przemienność ... 440
18.9 Przykład zupełnie niematematyczny ... 441
18.10 Cztery operatory, które muszą być niestatycznymi funkcjami składowymi ... 450
18.11 Operator przypisania = ... 451
18.11.1 Przykład na przeładowanie operatora przypisania ... 455
18.11.2 Jak to opowiedzieć potocznie? ... 461
18.11.3 Kiedy operator przypisania nie jest generowany automatycznie ... 463
18.12 Operator [ ] ... 464
18.13 Operator ( ) ... 468
18.14 Operator -> ... 470
18.15 Operator `new` ... 477

18.16 Operator `delete` ... 479
18.17 Operatory postinkrementacji i postdekrementacji, czyli koniec z niesprawiedliwością ... 480
18.18 Rady praktyczne dotyczące przeładowania ... 482
18.19 Pojedynek: Operator jako funkcja składowa, czy globalna ... 484
18.20 Zasłona spada, czyli tajemnica operatora << ... 486
18.21 Rzut oka wstecz ... 492

Tom III

19 Dziedziczenie ... 495
19.1 Istota dziedziczenia ... 495
19.2 Dostęp do składników ... 498
19.2.1 Prywatne składniki klasy podstawowej ... 498
19.2.2 Nieprywatne składniki klasy podstawowej ... 500
19.2.3 Klasa pochodna też decyduje ... 501
19.2.4 Po znajomości, czyli udostępnianie wybiórcze ... 503
19.3 Czego się nie dziedziczy ... 504
19.3.1 Niedziedziczenie konstruktorów ... 504
19.3.2 Niedziedziczenie operatora przypisania ... 505
19.3.3 Niedziedziczenie destruktora ... 505
19.4 Dziedziczenie kilkupokoleniowe ... 506
19.5 Dziedziczenie – doskonałe narzędzie programowania ... 507
19.6 Kolejność wywoływania konstruktorów ... 509
19.7 Przypisanie i inicjalizacja obiektów w warunkach dziedziczenia ... 515
19.7.1 Klasa pochodna nie definiuje swojego operatora przypisania ... 515
19.7.2 Klasa pochodna nie definiuje swojego konstruktora kopiującego ... 516
19.7.3 Inicjalizacja i przypisywanie według obiektu wzorcowego będącego `const` ... 517
19.7.4 Definiowanie konstruktora kopiującego i operatora przypisania dla klasy pochodnej ... 517
19.8 Dziedziczenie wielokrotne ... 522
19.8.1 Konstruktor klasy pochodnej przy wielokrotnym dziedziczeniu ... 524
19.8.2 Ryzyko wieloznaczności przy dziedziczeniu ... 526
19.8.3 Bliższe pokrewieństwo usuwa wieloznaczność ... 528
19.8.4 Poszlaki ... 528
19.9 Pojedynek: Dziedziczenie klasy contra zwieranie obiektów składowych ... 529
19.10 Konwersje standardowe przy dziedziczeniu ... 531
19.10.1 Panorama korzyści ... 535
19.10.2 Czego robić nie można ... 537
19.10.3 Konwersje standardowe wskaźników do pokazywania we wnętrzu klasy ... 540
19.11 Wirtualne klasy podstawowe ... 542
19.11.1 Publiczne i prywatne dziedziczenie tej samej klasy wirtualnej ... 545
19.11.2 Uwagi o konstrukcji i inicjalizacji w wypadku klas wirtualnych ... 546
19.11.3 Dominacja klas wirtualnych ... 549

20 Funkcje wirtualne ... 552
20.1 Polimorfizm ... 559

20.2 Dalsze szczegóły ... 562
20.3 Wczesne i późne wiązanie ... 565
20.4 Kiedy dla wywołań funkcji wirtualnych mimo wszystko zachodzi wczesne wiązanie ... 566
20.5 Kulisy białej magii, czyli: Jak to jest zrobione ? ... 568
20.6 Funkcja wirtualna, a mimo to `inline` ... 570
20.7 Pojedynek – funkcje przeładowane contra funkcje wirtualne ... 570
20.8 Klasy abstrakcyjne ... 571
20.9 Destruktor? to najlepiej wirtualny! ... 578
20.10 Co prawda konstruktor nie może być wirtualny, ale... ... 583
20.11 Finis coronat opus ... 588

21 Operacje Wejścia / Wyjścia ... 590

21.1 Biblioteka iostream ... 591
21.2 Strumień ... 592
21.3 Strumienie predefiniowane ... 593
21.4 Operatory >> i << ... 594
21.5 Domniemania w pracy strumieni predefiniowanych ... 595
21.6 Uwaga na priorytet ... 598
21.7 Operatory << oraz >> definiowane przez użytkownika ... 600
21.7.1 Operatorów wstawiania i wyjmowania ze strumienia - nie dziedziczy się ... 604
21.7.2 Operatory wstawiania i wyjmowania nie mogą być wirtualne. Niestety. ... 606
21.8 Sterowanie formatem ... 607
21.9 Flagi stanu formatowania ... 608
21.9.1 Znaczenie poszczególnych flag sterowania formatem ... 609
21.10 Sposoby zmiany trybu (reguł) formatowania ... 612
21.10.1 Zmiana sposobu formatowania funkcjami `setf`, `unsetf` ... 614
21.10.2 Wygodniejsze funkcje do zmiany stanu formatowania. ... 616
21.11 Manipulatory ... 621
21.11.1 Manipulatory bezargumentowe ... 622
21.11.2 Manipulatory parametryzowane ... 624
21.11.3 Definiowanie swoich manipulatorów ... 628
21.12 Nieformatowane operacje wejścia/wyjścia ... 631
21.13 Omówienie funkcji wyjmujących ze strumienia ... 633
21.13.1 Funkcje do pracy ze znakami i stringami ... 633
21.13.2 Wczytywanie binarne – funkcja `read` ... 638
21.13.3 Funkcja `ignore` ... 639
21.13.4 Pożyteczne funkcje pomocnicze ... 640
21.13.5 Funkcje wstawiające do strumienia ... 643
21.14 Operacje we/wy na plikach ... 644
21.14.1 Otwieranie i zamykanie strumienia ... 646
21.15 Błędy w trakcie pracy strumienia ... 650
21.15.1 Flagi stanu błędu strumienia ... 650
21.15.2 Funkcje do pracy na flagach błędu ... 651
21.15.3 Kilka udogodnień ... 652
21.15.4 Bardziej wyszukane operacje na flagach błędu strumienia ... 654
21.15.5 Trzy plagi - czyli „gotowiec" jak radzić sobie z błędami ... 657

21.16 Przykład programu pracującego na plikach ... 660
21.17 Wybór miejsca czytania lub pisania w pliku ... 662
21.17.1 Funkcje składowe informujące o pozycji wskaźników ... 663
21.17.2 Wybrane funkcje składowe do pozycjonowania wskaźników ... 664
21.18 Przykład większego programu ... 665
21.19 Tie – harmonijna praca dwóch strumieni ... 672
21.20 Attach – zmiana koryta strumienia ... 674
21.21 Dlaczego tak nie lubimy biblioteki `stdio`? ... 675
21.22 Niektóre aspekty współżycia biblioteki `stdio` z biblioteką `iostream` ... 677
21.23 Formatowanie wewnętrzne – Operacje wyjścia ... 679
21.23.1 Mechanizm automatycznej rezerwacji miejsca ... 682
21.23.2 Anonimowy strumień wyjściowy ... 685
21.24 Formatowanie wewnętrzne – operacje wejścia ... 685
21.24.1 Przykładowe zastosowanie oraz anonimowy strumień wejściowy ... 688
21.25 Ożenek: klasy wejściowo – wyjściowe dla formatowania wewnętrznego ... 690
21.26 Jeszcze o strumieniach anonimowych ... 691

22 Projektowanie programów obiektowo orientowanych ... 693

22.1 Przegląd kilku technik programowania ... 694
22.1.1 Programowanie liniowe ... 694
22.1.2 Programowanie proceduralne ... 694
22.1.3 Programowanie z ukrywaniem danych ... 695
22.1.4 Programowanie obiektowe - programowanie „bazujące" na obiektach ... 695
22.1.5 Programowanie Obiektowo Orientowane (OO) ... 696
22.2 O wyższości programowania obiektowo orientowanego nad Świętami Wielkiej Nocy ... 696
22.3 Obiektowo Orientowane: Projektowanie ... 699
22.4 Praktyczne wskazówki dotyczące projektowania programu techniką OO ... 701
22.4.1 Rekonesans – czyli rozpoznanie zagadnienia ... 701
22.4.2 Faza projektowania ... 702
22.4.3 Etap 1: Identyfikacja zachowań systemu ... 703
22.4.4 Etap 2: Identyfikacja obiektów (klas obiektów) ... 704
22.4.5 Etap 3: Usystematyzowanie klas obiektów ... 705
22.4.6 Etap 4: Określenie wzajemnych zależności klas ... 707
22.4.7 Etap 5: Składanie modelu. Określanie sekwencji działań obiektów i cykli życiowych ... 709
22.5 Faza implementacji ... 710
22.6 Przykład projektowania ... 710
22.7 Faza: Rozpoznanie naszego zagadnienia ... 711
22.8 Faza: Projektowanie ... 715
22.8.1 Etap 1 – Identyfikacja zachowań naszego systemu ... 715
22.8.2 Etap 2 – Identyfikacja klas obiektów, z którymi mamy do czynienia ... 716
22.8.3 Etap 3 - Usystematyzowanie klas obiektów z występujących w naszym systemie ... 719
22.8.4 Etap 4 - Określenie wzajemnych zależności klas ... 721
22.8.5 Etap 5 - Składamy model naszego systemu ... 723
22.9 Implementacja modelu naszego systemu ... 727
22.10 Symfonia C++, Coda ... 734
22.11 Posłowie ... 734

Kazimierze Grębosz
Józefowi Gręboszowi

moim rodzicom

O Proszę nie czytać tego !

Zaprzyjaźnijmy się !

Jeśli mamy ze sobą spędzić parę godzin, to proponuję – przejdźmy na „ty". Moje nazwisko zobaczyłeś już na okładce, dodam więc tylko, że książka ta powstała na podstawie moich doświadczeń jako programisty w Hahn-Meitner Institut †) w Berlinie – dawniej Zachodnim. Jestem oczarowany jasnością, prostotą i logiką programowania w języku C++ i dlatego proponuję Ci spędzenie ze mną paru chwil w krainie programowania obiektowo orientowanego.

Wstępne założenie

Książka ta jest napisana z myślą o czytelnikach, którzy znają choćby jeden język programowania – wszystko jedno, czy to będzie BASIC czy język C. Muszę przyznać, że początkowo odczuwałem pokusę napisania książki dla czytelników znających język C, jednak gdy zorientowałem się, że na polskim rynku „C" ma opinię jakiegoś trudnego języka dla specjalistów – postanowiłem zacząć prawie od zera.

Prawdę mówiąc obecnie język C jest już językiem starym. Na przestrzeni lat dostrzegano jego liczne niedoskonałości. Powstała też rewolucyjna idea programowania obiektowo orientowanego, która oprócz swej elegancji ma aspekt wymierny – pozwala oszczędzić tysiące dolarów wydawanych na pracę zespołów programistów pracujących nad dużymi projektami. Środowisko programistów od dawna oczekiwało na pojawienie się czegoś (jeszcze) lepszego niż C. Gdy pojawił się język C++ łączący w sobie prostotę programowania klasycznego C i możliwość programowania obiektowo orientowanego — przyjęty został z ogromnym entuzjazmem.

†) instytut badań jądrowych

Czym jest programowanie techniką obiektowo orientowaną? Dlaczego to taka sensacja i rewolucja?

Najkrócej mówiąc polega ono na zmianie sposobu myślenia o programowaniu. Przypomnij sobie jak to jest, gdy musisz napisać program w znanych Ci do tej pory językach programowania. Wszystko jedno czy to będzie program na sterowanie lotem samolotu, czy program na obsługę eksperymentu fizycznego. Otóż najpierw starasz się ten kawałek rzeczywistości, który masz oprogramować, wyrazić za pomocą liczb – zmiennych występujących w programie. Dziesiątki takich zmiennych są (luźno) rozrzucone po Twoim programie, a samo napisanie programu, to napisanie sekwencji działań na tych liczbach.

Ten sam problem w ujęciu obiektowo orientowanym rozwiązuje się tak, że w programie buduje się małe i większe modele obiektów świata realnego, po czym wyposaża się je w zachowania i po prostu pozwala się im działać. Obiektem może być ster kierunku samolotu, czy też układ elektroniczny przetwarzający jakiś sygnał. Gdy takie modele już w programie istnieją, wydają sobie nawzajem polecenia – np. obiekt drążek sterowy wydaje polecenie sterowi kierunku, aby wychylił się 7 stopni w prawo.

Obiekty nie pouczają się *jak* coś trzeba zrobić, tylko mówią *co* trzeba zrobić. Programowanie tą techniką – poza tym, że jest logiczne, bo odzwierciedla relacje istniejące w świecie rzeczywistym – programowanie tą techniką dostarcza po prostu dobrej zabawy.

Argument o dobrej zabawie

nie przekonałby oczywiście nigdy szefów dużych zespołów programistów — dla nich najważniejszy jest fakt, że ta technika pozwala, by programiści znali się tylko na swoim kawałku pracy bez konieczności opanowywania całości olbrzymiego projektu. Pozwala ona też na bardzo łatwe wprowadzanie poważnych modyfikacji do programu – bez konieczności angażowania w tę pracę całości zespołu. Dla tego ostatniego – programowanie obiektowo orientowane jest wręcz jakby stworzone. Te cechy są łatwo przeliczane na pieniądze – nakłady finansowe na pracę zespołu.

Jest także powód dla którego język C++ zrobił karierę nawet wśród programistów pracujących pojedynczo, a także wśród amatorów. Otóż język ten pozwala na łagodne przejście z klasycznych technik programowania na technikę obiektowo orientowaną.

Ta cecha nazywana jest hybrydowością języka C++. W programie można technikę OO stosować w takim stopniu w jakim się ją opanowało. Tak też dzieje się najczęściej – programiści wybierają sobie z niego najpierw tylko same „rodzynki" i je stosują, a w każdym następnym programie sięgają po coś więcej. W języku C++ można pisać nawet programy nie mające z techniką OO nic wspólnego. Tylko, że to tak, jakby zwiedzać Granadę mając zamknięte oczy[†)].

†) "...kobieto, daj jałmużnę ślepcowi, bo nie ma większego nieszczęścia niż być ślepcem w Granadzie..."

Dość na tym. O technice OO porozmawiamy jeszcze wielokrotnie. Teraz chciałbym wytłumaczyć się z paru spraw.

Dlaczego ta książka jest taka gruba?

Nie wynika to wcale z faktu by język C++ był tak trudny. Uznałem tylko, że to, co uczy naprawdę – to przykłady. W książce więc oprócz „teorii" są setki przykładowych programów, a każdy jest szczegółowo omówiony. Przykładom towarzyszą wydruki pokazujące wygląd ekranu po wykonaniu programu. To wszystko właśnie sprawia, że książka jest tak obszerna. Jednak wydruki te załączam w przeświadczeniu, że często łatwiej zorientować się „co program robi" – rzucając okiem na taki wydruk (ekran). Dopiero potem radzę analizować sam program.

Do wykonania przykładowych programów użyłem kompilatora Borland C++ v 3.1 (komputer IBM PC) gdyż sądzę, że jest to kompilator, z którym przeciętny polski czytelnik może się spotkać najczęściej. Oczywiście w książce zajmujemy się samym językiem C++ – więc powinna Ci ona pomóc niezależnie od tego, z którym komputerem i typem kompilatora masz do czynienia.

Wersja języka

Sam język C++ ciągle się unowocześnia. Opisuję go tu bez tzw. *templates* i *exception handling*, jako że w tej wersji istnieją na razie tytułem eksperymentu[†]. Absolutnym autorytetem w sprawie języka była dla mnie książka (twórcy tego języka) Bjarne'a Stroustrup'a i Margaret A. Ellis – The Annotated C++ Reference Manual. W razie jakichkolwiek niejasności odsyłam do niej. (Uprzedzam jednak, że jest napisana w bardzo sformalizowanym stylu i nie jest pomyślana jako podręcznik do nauki).

W tekście czasem wspominam o wersji języka. Pamiętać należy, że termin *wersja języka* nie ma nic wspólnego z *wersją kompilatora*. Wersja kompilatora to wersja konkretnego dostarczonego Ci produktu. Tymczasem wersja języka to reguły gry, według których ten kompilator postępuje.

Pióro

Ktoś kiedyś powiedział, że są dwa style pisania – pierwszy to: „–popatrzcie jaki ja jestem mądry" – a drugi to : „–popatrzcie jakie to proste". Ja w tej książce wybieram ten drugi wariant w przeświadczeniu, że prędzej zaprowadzi do celu. Z drugiej strony wiem, że bezpośredni, wręcz kolokwialny styl tej książki, zupełnie naturalny w książkach na Zachodzie – w Polsce może zaskoczyć niektórych czytelników.

---

†) *Uwaga do bieżącego wydania:* Ponieważ dziś już wiadomo, że ten eksperyment był udany i owe pojęcia weszły do C++ na stałe – poświęciłem im osobną książkę pod tytułem „Pasja C++". Są w niej omówione szablony funkcji, szablony klas, klasy pojemnikowe i obsługa sytuacji wyjątkowych. Zajrzyj do niej jeśli już przeczytasz „Symfonię C++" i ją polubisz.

Dla kogo jest ta książka?

Pisząc tę książkę musiałem najpierw określić sobie czytelnika, dla którego jest ta książka. Otóż jest ona dla tzw. szerokiego grona czytelników. Pisałem ją tak, by była podręcznikiem dla czternastoletniego programisty amatora, jak i dla zawodowego informatyka. Trudno te sprawy pogodzić, więc są w niej kompromisy w obu kierunkach. Co do jednego z takich kompromisów mam najwięcej obaw:

Czołem bracia angliści !

Gdy słyszałem, jak początkujący programiści wymawiają występujące w języku C++ angielskie słowa takie jak np. „unsigned", „volatile", „width" — postanowiłem w miejscach, gdzie się pojawiają po raz pierwszy – zamieścić adnotacje o ich wymowie. Wymowa podana jest nie w transkrypcji fonetycznej, ale za pomocą polskich liter. Wiem, że fakt ten może razić wielu czytelników. Proszę wtedy o wyrozumiałość – pamiętajcie, że ta książka ma służyć również maluczkim. Wymowa podana jest chyłkiem – w przypisie, w nawiasie, a w dodatku jeszcze w cudzysłowie – więc nie trzeba jej koniecznie czytać. Spodziewam się, że fakt zamieszczenia wymowy nie będzie przeszkadzał Czytelnikom, którzy językiem angielskim posługują się na codzień od lat – ale na pewno wzbudzi protest tych, którzy angielskiego nauczyli się przedwczoraj.

A swoją drogą, to nawet wśród moich kolegów fizyków i programistów – nie spotkałem poprawnie wymawiających słowo: „width".

A teraz o strukturze tej książki

Można ją podzielić na dwie zasadnicze części – tę klasyczną, opisującą zwykłe narzędzia programowania, i drugą (zaczynającą się od rozdziału o klasach) opisującą narzędzia do programowania obiektowo orientowanego. Po tym następuje rozdział omawiający operacje wejścia/wyjścia – czyli sposoby pracy z takimi urządzeniami zewnętrznymi jak klawiatura, ekran, dyski magnetyczne. Wreszcie następuje rozdział o projektowaniu programu obiektowo orientowanego – zawierający szczegółowy instruktaż. Uznałem te sprawy za bardzo ważne, gdyż często programista mając w ręce to doskonałe narzędzie programowania, jakim jest C++, nie wie, co z nim począć.

Metodą kolejnych przybliżeń

Nie liczę na to, że czytając tę książkę zrozumiesz wszystko od razu. Wręcz przeciwnie – w tekście wielokrotnie sugeruję, byś opuścił niektóre paragrafy przy pierwszym czytaniu książki. Nie wszystkie aspekty są ważne już na samym początku nauki. Lepiej mieć najpierw ogólny pogląd, a dopiero potem zagłębiać się w trudniejsze szczegóły. Licząc na to, że po pierwszym czytaniu całości w niektóre miejsca wrócisz – pewne fragmenty tekstu opatrzyłem adnotacją: „dla wtajemniczonych". Są tam uwagi wybiegające nieco do przodu, ale dotyczące spraw, które po pierwszym czytaniu będziesz już przecież znał.

Nie szanuj tej książki. Przygotuj sobie kolorowe (tzw. niekryjące) flamastry i czytając zakreślaj ważne dla Ciebie wyrazy czy myśli. Na marginesach pisz ołówkiem swoje uwagi i komentarze. Chociażby miały to być teksty typu „Co za bzdura!" albo „Nie rozumiem dlaczego ..." albo „porównaj dwie strony

wcześniej". Ta aktywność zaprocentuje Ci natychmiast, bo mając tak osobisty stosunek do tekstu, łatwiej koncentrować na nim uwagę – co jest warunkiem *sine qua non* szybkiej nauki.

Jeśli znasz język C „klasyczny" (czyli tzw. ANSI C)

to zapewne pierwsze rozdziały będą dla Ciebie wyraźnie za łatwe. Mimo to radzę je przejrzeć chociaż pobieżnie, gdyż występują pewne różnice w językach C i C++ – nawet na tym etapie (oczywiście na korzyść C++).

Gorąco natomiast zachęcam do uważnego przeczytania rozdziału o wskaźnikach. Z doświadczenia wiem, że te sprawy zna się zwykle najgorzej. Tymczasem zarówno w klasycznym C jak i w C++ wskaźnik jest bardzo ważnym i pożytecznym narzędziem. Dlatego ten rozdział tak rozbudowałem.

Podziękowania

Winien jestem wdzięczność przyjaciołom, którzy w jakiś sposób przyczynili się do powstania tej książki. Byli to: Sven Bläser, Olaf Boebel, Johannes Diemer, Jeffrey Erxmeyer, Bengt Skogvall, Kai Sommer, Klaus Spohr, Joachim Steiger i Detlef Weidenhammer. Zupełnie wyjątkowe podziękowania należą się Sycylijczykowi Pierpaolo Figuera za wiele godzin, które mi poświęcił. Ponieważ ta książka pisana była od razu po polsku, dlatego żaden z nich nigdy jej nie czytał. Wszystko, co dobre w tej książce, jest jednak ich zasługą, wszystko co złe – to wyłącznie moja wina. Jestem winien wdzięczność pierwszym czytelnikom tej książki: Mirkowi Zięblińskiemu i Bożenie Potempie. Ich uwagi krytyczne sprawiły, że książka stała się lepsza, ich uwagi pozytywne – zawsze wprawiały mnie w dobry nastrój.

Oczywiście na pewno niejedno da się w tej książce poprawić. Jeśli będziesz miał jakieś uwagi, to proszę przyślij mi je pocztą elektroniczną na adres:

`grebosz@bron.ifj.edu.pl`

lub pocztą tradycyjną na adres wydawnictwa. Z góry wyrażam moją wdzięczność za nawet najdrobniejsze przyczynki.

Kod źródłowy przykładowych programów z tej książki można sobie łatwo sprowadzić z moich stron WWW w Internecie. Jej obecny adres to:

`http://www.ifj.edu.pl/~grebosz`

Nawet jeśli ten adres się kiedyś zmieni - łatwo znaleźć nową lokalizację robiąc tzw. search hasła "SYMFONIA C++", lub mojego nazwiska.

Na koniec wypada mi się wytłumaczyć z tytułu tego wstępu. Chciałem tu powiedzieć parę ważnych, ale i trochę nudnych spraw. Wiedząc, że czytelnicy najczęściej opuszczają wstępy – dałem taki tytuł podejrzewając, że zakazany owoc najlepiej smakuje.

1 Startujemy !

Aby pisać, trzeba pisać.... – Taką radę zwykło się dawać początkującym literatom. Tę samą zasadę można zastosować do piszących programy. Programować w danym języku można nauczyć się tylko pisząc w nim programy. Dlatego bez dalszych wstępów napiszmy nasz

1.1 Pierwszy program

```
#include <iostream.h>
main()
{
     cout << "Witamy na pokladzie" ;
}
```

Wykonanie tego programu spowoduje pojawienie się na ekranie tekstu:

```
Witamy na pokladzie
```

Przyjrzyjmy się bliżej temu programowi

W każdym programie w języku C++ musi być specjalna funkcja zwana `main`.†) Od tej funkcji zaczyna się wykonywanie programu. Treść tej funkcji – jej ciało – czyli innymi słowy instrukcje wykonywane w ramach tej funkcji – zawarte są między dwoma nawiasami klamrowymi: { }

†) main – ang. główna (czytaj: „mejn").
Uwaga zecerska: jeśli z przyczyn typograficznych przypis nie może pojawić się u dołu strony, należy go szukać na stronie następnej.

W naszym wypadku w funkcji `main` jest tylko jedna instrukcja

```
cout << "Witamy na pokladzie" ;
```

która sprawia, że na standardowym urządzeniu wyjściowym `cout` - czyli po prostu na ekranie – ma się pojawić tekst, zamieszczony tu w cudzysłowie. Skrót `cout`[†)] wymawia się po polsku *Si-aut.* (Błagam, tylko nie: *Śi-aut !*)

Znaki << oznaczają właśnie akcję, którą ma podjąć `cout` - czyli wyprowadzić na ekran tekst. Umieszczony na końcu średnik jest znakiem końca instrukcji.

Operacje wejścia/wyjścia

Należy tu wyraźnie podkreślić, że operacje związane z wprowadzaniem i wyprowadzaniem informacji na urządzenia takie, jak np. ekran – czyli tzw. operacje wejścia/wyjścia – nie są częścią definicji języka C++. Podprogramy odpowiedzialne za to są w jednej ze standardowych bibliotek, w które zwykle wyposażane są kompilatory. Abyśmy mogli z takiej biblioteki skorzystać w programie, musimy na początku umieścić linijkę

```
#include <iostream.h>
```

która oznacza, że życzymy sobie, by kompilator przed przystąpieniem do pracy nad dalszymi linijkami programu wstawił [††)] w tym miejscu tak zwany plik nagłówkowy biblioteki `iostream`.

> *Dla zainteresowanych dodam, że tenże plik nagłówkowy biblioteki zawier dokładne deklaracje funkcji bibliotecznych, z których ewentualnie możn korzystać. Dzięki temu, że kompilator zapoznaje się z tymi deklaracjami może od tej pory sprawdzać nas czy posługujemy się tymi funkcjam poprawnie. To bardzo korzystna cecha.*

Biblioteką tą zajmiemy się bliżej pod koniec tej książki.

Wolny format zapisu programu

A teraz uwaga natury ogólnej. Program pisze się umieszczając kolejne instrukcj w linijkach jedna pod drugą. Otóż w niektórych językach programowania (np FORTRAN'ie, BASIC'u) obowiązują ścisłe reguły określające pozycję (w linijce) na której dany składnik instrukcji może się znaleźć.

Np. w FORTRAN'ie – Jeśli chcemy umieścić znak komentarza, to stawiamy g w kolumnie pierwszej, jeśli ma to być numer etykiety – to do tego służą kolumn 1-5, jeśli chcemy umieścić znak kontynuacji z poprzedniej linijki — to umiesz czamy go w kolumnie szóstej.

Podobnie w BASIC'u – linia instrukcji musi się zacząć od numeru etykiety.

W języku C++ jest inaczej. Język C++ (podobnie jak C) jest językiem o tzw wolnym formacie. Krótko mówiąc – nie ma żadnych przymusów. Wszystk może znaleźć się w każdym miejscu linii, a nawet zostać rozpisane na 10 linijek Poza nielicznymi sytuacjami, w dowolnym miejscu instrukcji można przejść d

†) od ang. C-onsole OUT-put
††) ang. include (czytaj: „inklud")

nowej linii i tam kontynuować pisanie. To dlatego, że koniec instrukcji określany jest nie przez koniec linii, ale przez średnik, który stawiamy na końcu.

Białe znaki

Wewnątrz instrukcji można postawić dodatkowe znaki spacji i tabulatory, czy nawet znaki nowej linii. Są to tzw. białe znaki – białe, bo na papierze objawiają się jako niezadrukowane. Znaki te napotkane wewnątrz instrukcji są prawie zawsze ignorowane.

Zatem nasza instrukcja równie dobrze mogłaby wyglądać tak:

```
cout
    <<
            "witamy na pokladzie"
        ;
```

Wstawianie białych znaków służy nie kompilatorowi lecz programiście. Pomaga w tym, żeby program wyglądał czytelnie. Jeśli nam na tym wcale nie zależy, to możemy równie dobrze napisać program tak:

```
#include <iostream.h>
main(){cout<<"witamy na pokladzie";}
```

Nikt rozsądny jednak tak nie robi z dwóch powodów:

- program staje się wtedy nieprzejrzysty,
- korzystając z różnych specjalnych narzędzi do uruchamiania programów (tzw. debuggerów)[†] mamy możliwość śledzenia wykonywania programu krokowo: linijka za linijką. Dobrze jest więc mieć w jednej linijce tylko jedną instrukcję.

Kompilator i Linker

W ten sposób omówiliśmy sobie pierwszy program. Oczywiście program w takiej postaci, jak napisaliśmy, jest dla komputera niezrozumiały. Musi zostać przetłumaczony na język maszyny. Służy do tego kompilator. Nasz program poddajemy więc kompilacji i otrzymujemy wersję skompilowaną.

Taka skompilowana wersja jest jeszcze niepełna – musi zostać połączona z bibliotekami.

> Ten proces łączenia wykonywany jest przez program zwany zwykle *linkerem* (link – ang. łączenie), a w żargonie programistów określane jest to jako linkowanie. Nie słyszałem by ktoś nazywał to inaczej. Poprawne, lansowane niegdyś określenie „konsolidacja" i „konsolidowanie" chyba się nie przyjęło. My używać będziemy sformułowania ***linkowanie*** względnie ***łączenie.***

Przyłączenie funkcji bibliotecznych następuje dopiero w czasie linkowania. Nasza dyrektywa (instrukcja) `#include` zapoznała kompilator jedynie z samym nagłówkiem biblioteki. Potrzebny był on po to, by kompilator mógł

†) bug – ang. owad ; stąd debugger (czytaj: „debager") – jakby: „odpluskwiacz"

sprawdzić poprawność naszego odnoszenia się do biblioteki. Natomiast sama treść funkcji bibliotecznych dołączana jest dopiero na etapie linkowania.

A teraz do dzieła. W rezultacie linkowania otrzymaliśmy program w wersji nadającej się do uruchomienia.

Zachęcam Cię czytelniku, byś teraz spróbował uruchomić nasz program na swoim komputerze. Mimo, że program jest prymitywny. Ważne tu jest, byś opanował technikę kompilacji i linkowania. Niestety nie mogę Ci tu nic pomóc. Istnieje bardzo wiele typów komputerów i wiele różnych typów kompilatorów. Jak Twojego kompilatora używać – musisz przeczytać w swojej dokumentacji lub zapytać kolegę.

Wszystko to nie jest trudne, najczęściej wcale nie musisz myśleć o linkowaniu, bo kompilator sam to uruchamia. Wtedy jest to tylko jedna komenda, a np. w kompilatorze Borland C++ naciśnięcie jednego tylko klawisza.

Uwaga: ą kompilatory, które mogą kompilować programy napisane w klasycznym C, a także w C++. Wówczas trzeba im powiedzieć w jakim języku jest napisany program przedstawiony właśnie do kompilacji. Jednym ze sposobów powiedzenia tego może być rozszerzenie nazwy naszego programu. Jeśli rozszerzeniem jest CPP, wówczas kompilator podejdzie do pracy jak do programu w języku C++. Czyli nasz program powinien się nazywać na przykład

```
moj_progr.cpp
```

Kiedy już program zadziała poprawnie i na ekranie pojawi się nasz tekst `Witamy...` wówczas możemy spróbować zmodyfikować ten program. Działania te pozwolą nam bliżej zapoznać się z techniką wypisywania na ekran. To się nam bardzo szybko przyda. A zatem w środek tekstu ujętego w cudzysłów wpiszmy znaki `\n`

```
"Witamy \nna pokladzie"
```

Zmiana ta powoduje, że tekst wypisany na ekranie wyglądał będzie następująco:

```
Witamy
na pokladzie
```

Znak `\n` (n – jak: new line, czyli: nowa linia) powoduje, że w trakcie wypisywania tekstu na ekranie następuje przejście do nowej linii i dalszy ciąg tekstu wypisywany jest poniżej.

Ewentualna następna instrukcja wyprowadzania tekstu zacznie go wyprowadzać od miejsca, w którym poprzednia skończyła. Zatem dwie instrukcje

```
cout << "Witamy \nna pokladzie" ;
cout << "Lecimy na " << "wysokosci 3500 stop" ;
```

spowodują pojawienie się na ekranie tekstu

```
Witamy
na pokladzieLecimy na wysokosci 3500 stop
```

Nie ma też znaczenia czy napiszemy

```
cout << "lecimy na " ;
cout << "wysokosci 3500 stop" ;
```

czy też może złożymy te instrukcje i zapiszemy krócej

```
cout << "lecimy na " << "wysokosci 3500 stop" ;
```

W obu sytuacjach rezultat na ekranie będzie ten sam:

```
lecimy na wysokosci 3500 stop
```

Dlaczego właściwie możliwe jest tak wygodne składanie – nie mogę Ci jeszcze teraz wyjaśnić. Musisz być cierpliwy i doczytać do rozdziału o bibliotece iostream.

1.2 Drugi program

To było wypisywanie informacji na ekranie. Natomiast z wczytywaniem danych z klawiatury spotykamy się w naszym drugim programie.

Oto on:

```
/* -----------------------------------------------------
Program na przeliczanie wysokosci podanej
w stopach na wysokosc w metrach.
Cwiczymy tu operacje wczytywania z klawiatury
i wypisywania na ekranie
-----------------------------------------------------*/
#include <iostream.h>
main()
{
int       stopy ;                  // to do przechowywania
                                   //        liczby stop
float     metry ;                  // do wpisania wyniku
float     przelicznik = 0.3 ;      // przelicznik:
                                   //        stopy na metry

     cout << "Podaj wysokosc w stopach : " ;
     cin >> stopy ;                // przyjecie danej
                                   //        z klawiatury

     metry = stopy * przelicznik; // wlasciwe przeliczenie

     cout << "\n" ;          // to samo co: cout << endl ;

     // -----wypisanie wynikow
     cout << stopy << " stop - to jest : "
          << metry << " metrow\n" ;
}
```

Komentarze

Już na pierwszy rzut oka widać, że w programie pojawiły się opisy w „ludzkim" języku. Są to **komentarze**. Komentarze są tekstami zupełnie ignorowanymi przez kompilator, ale za to są bardzo pożyteczne dla programisty, bo przypominają nam co w danym miejscu programu chcieliśmy zrobić.

W języku C++ komentarze można umieszczać dwojako.

- Pierwszy sposób to ograniczenie jakiegoś tekstu znakami `/*` (z lewej) oraz `*/` (z prawej).
  Sposób ten zastosowaliśmy na początku programu. Komentarz taki może się ciągnąć przez wiele linijek – to także widzimy w przykładzie.
- Drugi sposób to zastosowanie znaków `//` (dwa ukośniki). Kompilator po napotkaniu takiego znaku ignoruje resztę znaków do końca linijki – traktując je jako komentarz.

Uwaga:

> Komentarze typu /* ... */ nie mogą być w sobie zagnieżdżane.

To znaczy, że niepoprawna jest taka forma

```
/* KOMENTARZ "ZEWNETRZNY"
TAK SIE CIAGNIE /* komentarz wewnetrzny */ DOKONCZENIE
ZEWNETRZNEGO */
```

Chciałoby się powiedzieć: niestety! Całe szczęście niektóre kompilatory, mimo zaleceń ANSI, pozwalają na to[†)]*. Możliwość zagnieżdżeń komentarzy jest czasem bardzo pomocna.*
Wyobraźmy sobie kawałek większego programu, wyposażonego już w komentarze. Nagle chcielibyśmy zrezygnować chwilowo z 10 linijek. Usunąć je z procesu kompilacji, ale nie skasować zupełnie. Naturalnym odruchem jest wtedy przerobienie tego fragmentu na komentarz.
Robi się to przez ujęcie tych linijek w symbole komentarza: `/*` oraz `*/` – stawiając pierwszy symbol na początku, a drugi na końcu tych 10 linijek. Jeśli jednak w rzeczonym fragmencie programu są już jakieś komentarze typu `/* ... */`, to kompilator (niepozwalający na zagnieżdżanie ich) uzna to za błąd.
Jak powiedziałem, postąpi tak kompilator niepozwalający na zagnieżdżanie, bo są kompilatory, które po wybraniu odpowiedniej opcji pozwalają na to.

Ograniczenie o zagnieżdżaniu komentarzy nie dotyczy współżycia komentarzy typu `/*...*/` z komentarzami typu `//`

```
cout << "Uwaga pasazerowie : \n" ;
/* chwilowo rezygnuje z tych dwoch linijek ---------
cout << "Pali sie drugi silnik \n" ;  // opis sytuacji i
cout << "Nie ma szans ratunku \n" ;   // niech sie modla
```

†) ANSI – American National Standards Institute – amerykański instytut normalizacyjny.

```
-----------------------------------------------------*/
cout << "Prosze zapiac pasy...\n" ;
```

Jeśli Twój kompilator nie pozwala na zagnieżdżanie komentarzy, to możesz sobie poradzić stosując tak zwaną kompilację warunkową. (Patrz rozdz. o preprocesorze, str 122).

Na koniec dobra rada natury ogólnej:

> Czas zużyty na pisanie komentarzy nigdy nie jest czasem straconym. Bardzo szybko zauważysz, że czas ten odzyskasz z nawiązką w trakcie uruchamiania programu lub przy późniejszych jego modyfikacjach.

Opisuj znaczenie każdej zmiennej, opisuj funkcje i ich argumenty, opisuj też to, co w danym fragmencie programu robisz. Nawet jeśli wtedy, gdy to piszesz, jest to jeszcze dla Ciebie jasne. Opisuj przede wszystkim wszystkie „kruczki", które zastosowałeś. Dla samego siebie i dla tych, którzy modyfikować będą Twoje programy wtedy, gdy Ty będziesz już pisał w języku C++ systemy operacyjne.

Zmienne

Dosyć już o komentarzach, wróćmy do naszego drugiego programu. W funkcji `main` zauważamy linijki

```
int       stopy ;
float     metry ;
```

Są to **definicje zmiennych** występujących w programie. Zmiennym tym nadaliśmy nazwy: `stopy` oraz `metry`. Nazwy te są, jak widać, tak przez nas wybrane, by określały zastosowanie zmiennych. Można jednak zapytać ogólniej:

Co może być nazwą w języku C++ ?

Może to być dowolnie długi ciąg liter, cyfr oraz znaków podkreślenia '_'. Nazwa nie może się zacząć od cyfry. Małe i wielkie litery w nazwach są rozróżniane.

Nazwa nie może być identyczna z żadnym ze słów kluczowych języka C++. Te słowa kluczowe to:

asm	auto	break	case	catch	char
class	const	continue	default	delete	do
double	else	enum	extern	float	for
friend	goto	if	inline	int	long
new	operator	private	protected	public	register
return	short	signed	sizeof	static	struct
switch	template	this	throw	try	typedef
union	unsigned	virtual	void	volatile	while

Zatem jeśli wymyślamy swoją nazwę, to musimy unikać powyższych słów.

Nazwy są nam potrzebne po to, by za ich pomocą odnosić się w programie do naszych zmiennych, czy stałych - ogólnie mówić będziemy: do **obiektów**.

W języku C++ każda nazwa musi zostać zadeklarowana zanim zostanie użyta.

Tę złotą myśl możesz napisać sobie na ścianie. Swoją drogą – jeśli o tym zapomnisz, to kompilator Ci natychmiast o tym przypomni.

We wspomnianym fragmencie programu *deklarujemy*, że

- zmienna `stopy` służy do przechowywania liczby typu całkowitego `int`[†].

Deklarujemy też, że

- `metry` oraz `przelicznik` są zmiennymi do przechowywania liczby rzeczywistej – inaczej: liczby zmiennoprzecinkowej – `float`[††].

Definicja a deklaracja

Jest subtelna różnica między tymi dwoma pojęciami. Załóżmy, że chodzi o naszą nazwę `stopy`.

Deklaracja w momencie, gdy napotka ją kompilator mówi mu:

> „Jakbyś zobaczył kiedyś słowo `stopy`, to wiedz, że oznacza ono obiekt typu całkowitego."

Definicja zaś mówi mu:

> „A teraz zarezerwuj mi w pamięci miejsce na obiekt typu całkowitego o nazwie `stopy`"

Nasz fragment programu zawiera w sobie jednocześnie i jedną i drugą wypowiedź. Rezerwujemy miejsce w pamięci i od razu oznajmiamy co to za obiekt. Zatem:

Definicja jest równocześnie deklaracją. Ale nie odwrotnie.

Może być bowiem deklaracja, która nie jest definicją. Możemy przecież tylko zdeklarować, że konkretna nazwa oznacza obiekt jakiegoś typu, ale obiektu tego w danym miejscu nie definiujemy. Chociażby dlatego, że już jest zdefiniowany w zupełnie innym module programu.

Przy definicji obiektu `przelicznik` widzimy taki zapis

```
float     przelicznik = 0.3 ;
```

Jest to nie tylko zwykła definicja rezerwująca miejsce w pamięci, ale taka definicja, która to miejsce w pamięci dodatkowo inicjalizuje wpisując tam wartość 0.3

†) integer – ang. liczba całkowita (czytaj: „intidżer")
††) float (czytaj: „flout") od słów: floating point – ang. zmienny przecinek.

Wczytywanie danych z klawiatury

Przejdźmy dalej. W naszym programie instrukcja

```
cin >> stopy ;
```

jest operacją związaną z klawiaturą, czyli mówiąc inaczej ze standardowym urządzeniem wejściowym `cin` – skrót od: C-onsole IN-put. †)

Instrukcja ta umożliwia wczytanie z klawiatury liczby. Wartość tej liczby zostaje umieszczona w zmiennej `stopy`. Przy odrobinie wyobraźni można powiedzieć, że to właśnie sugeruje kierunek strzałek >>

Dygresja dla programistów klasycznego C.

Programujących w C klasycznym, ucieszył zapewne fakt tak prostego wprowadzania danych z klawiatury. Nie trzeba tu myśleć kiedy podać do funkcji wczytującej (`scanf`) nazwę zmiennej, a kiedy adres zmiennej. Jeszcze kilka tak miłych niespodzianek czeka nas w C++ .

Następne linijki programu to przeliczenie stóp na metry i wydruk wyniku.

Przykładowe wykonanie programu spowoduje wydruk na ekranie poniższego tekstu

(Tłustszym drukiem zaznaczone jest echo tekstu wpisywanego przez użytkownika programu.)

```
Podaj wysokosc w stopach : 3500
3500 stop - to jest : 1050 metrow
```

Spójrz jeszcze raz na tekst programu i zauważ jak to się stało, że na ekranie pojawiła się liczba 1050 – będąca obliczoną wartością umieszczoną w zmiennej `metry`. W komentarzach wyjaśnione są poszczególne kroki.

Zwróć też uwagę, na instrukcję

```
cout << "\n" ;          // cout << endl ;
```

W komentarzu zaznaczyłem inny sposób zapisu tego samego. Jeśli chcemy na ekran wyprowadzić sam znak `'\n'` to możemy go zastąpić skrótem `endl` co oznacza: end of line (koniec linii). Mimo, że jest to dokładnie tyle samo znaków do wystukania na klawiaturze, to jednak łatwiej się to jakoś pisze. W rozdziale poświęconym operacjom wypisywania na ekran poznamy dalsze takie sztuczki.

Tak więc wyglądał nasz program do przeliczania stóp na metry.

No cóż – pomyślałeś zapewne: –Tyle pracy z wpisywaniem tekstu programu, po to, by otrzymać ten wynik... Prościej byłoby obliczyć to „na piechotę".

Masz rację. Cud programowania objawia się dopiero w pętlach. Mówiąc szerzej: w instrukcjach sterujących.

---

†) Skrót ten wymawia się po polsku jako „Si-yn". Z twardym „S" !

2 Instrukcje sterujące

Są to bardzo przydatne polecenia służące do sterowania przebiegiem programu. Ponieważ założyliśmy, że znasz jakikolwiek inny język programowania, zatem bez dalszych komentarzy przystępujemy do prezentacji takich instrukcji. Występują one w każdym języku programowania i nawet wszędzie mają podobny wygląd.
W instrukcjach sterujących podejmowane są decyzje o wykonaniu tych czy innych instrukcji programu. Decyzje te podejmowane są w zależności od spełnienia lub niespełnienia jakiegoś warunku. Inaczej mówiąc, od prawdziwości lub fałszywości jakiegoś wyrażenia. Najpierw więc wyjaśnijmy sobie co to jest prawda, a co fałsz w języku C++

2.1 Prawda – Fałsz

W języku C++ nie ma specjalnego typu określającego zmienne logiczne – czyli takie, które przyjmują wartości: prawda - fałsz.

Za to do przechowywania takiej informacji nadaje się każdy typ. Zasada jest genialnie prosta:

Sprawdza się czy wartość danego obiektu – np. zmiennej – jest równa zero, czy różna od zera.

Wartość zero – odpowiada stanowi: fałsz
Wartość inna niż zero – odpowiada stanowi: prawda

Nie musi to być nawet zawartość jednego obiektu. Może to być także bardziej skomplikowane wyrażenie, które trzeba obliczyć, aby przekonać się jaka jest jego wartość.

Co ciekawe – wynik nie musi być wcale liczbą. Nawet obiekt przechowujący znaki alfanumeryczne może być w ten sposób sprawdzany. Sprawdza się wów-

czas kod liczbowy złożonego tam znaku. Jeśli jest różny od zera, to wyrażenie odpowiada rezultatowi „prawda", jeśli kod jest zerowy (czyli tzw. znak NULL) – odpowiada to rezultatowi „fałsz".

2.2 Instrukcja warunkowa `if`

Instrukcja `if` †) może mieć 2 formy:

`if(`*wyrażenie*`)` instrukcja1 ;

lub

`if(`*wyrażenie*`)` instrukcja1 ;
`else` instrukcja2 ;

Wyrażenie to tutaj coś, co ma jakąś wartość. Może być to po prostu obiekt wybrany przez nas do przechowywania zmiennej logicznej, ale może to być też naprawdę wyrażenie, które najpierw trzeba obliczyć, by w rezultacie tego poznać jego wartość.

Najpierw zatem obliczana jest wartość wyrażenia. Jeśli jest ona niezerowa (prawda), to wykonywana jest instrukcja1. Jeśli wartość wyrażenia jest zero (fałsz), to instrukcja1 nie jest wykonywana.

W drugiej wersji instrukcji `if` widzimy dodatkowo słowo `else` ††), co można przetłumaczyć jako: „w przeciwnym razie". A zatem jeśli w tej drugiej sytuacji wartość wyrażenia jest niezerowa (prawda), to zostanie wykonana instrukcja1 w przeciwnym razie (else!), czyli gdy wartość wyrażenia jest zerowa (fałsz), zostanie wykonana instrukcja2.

Powtarzam – wynik wyrażenia może być różnego typu (całkowity, rzeczywisty itd). Sprawdza się tylko czy jest równy 0 czy nie.

Oto prosty przykład:

```
int i ;              // definicja obiektu int o nazwie i
cout << "Podaj jakas liczbe: " ;
cin >> i ;
if(i - 4) cout << " zmienna i miala wartosc inna niz 4";
else      cout << " zmienna i miala wartosc rowna 4";
```

Załóżmy, że podaliśmy liczbę 15.
Wyrażeniem było tu: `i-4`. Obliczana jest więc jego wartość 15 - 4 = 11 , a to jest różne od 0 (zatem: prawda), więc wykonana zostaje instrukcja pierwsza.

Gdybyśmy podali liczbę 4, wówczas do zmiennej `i` podstawione zostałoby 4. Wyrażenie `i-4` miałoby wartość 0 (czyli: fałsz) i wtedy wykonana zostałaby instrukcja druga.

†) if – ang. jeśli (czytaj: „yf")
††) (czytaj: „els")

Blok instrukcji

Często się zdarza, że chodzi nam o wykonanie warunkowe nie jednej instrukcji, a całego bloku instrukcji. Stosujemy wówczas instrukcję składaną zwaną inaczej blokiem. Są to po prostu zwykłe instrukcje ograniczone nawiasami `{ }`. Zauważ, że po klamrze nie trzeba stawiać średnika.

```
{
     instr1 ;
     instr2 ;
     instr3 ;
}
```

Oto przykład programu, w którym stosujemy instrukcje składane.

```
#include <iostream.h>
main()
{
int wys, punkty_karne ;      // definicja dwóch zmiennych
          // typu int. Obie sa tego samego typu wiec
          // wystarczy przecinek odzielajacy nazwy

     cout << "Na jakiej wysokosci lecimy ? [w metrach] : ";
     cin >> wys ;

     // rozwazamy sytuacje -----------------------
     if(wys < 500)
     {
          cout << "\n" << wys << " metrow to za nisko !\n";
          punkty_karne = 1 ;
     }
     else
     {
          cout << "\nNa wysokosci " << wys
               << " metrow jestes juz bezpieczny \n" ;
          punkty_karne = 0 ;
     }

     // ocena Twoich wynikow -----------------------
     cout << "Masz " << punkty_karne
          << " punktow karnych \n" ;
     if(punkty_karne)cout << "Popraw sie !" ;
 }
```

Przypominam, że zróżnicowane odstępy od lewego marginesu (wypełnione białymi znakami) nie mają dla kompilatora żadnego znaczenia. Pomagają natomiast programiście. Dzięki nim program staje się bardziej czytelny.

Oto przykładowy wygląd ekranu po wykonaniu tego programu

```
Na jakiej wysokosci lecimy ? [w metrach] : 2500
Na wysokosci 2500 metrow jestes juz bezpieczny
Masz 0 punktow karnych
```

Jeśli na zadane pytanie odpowiemy inaczej, to ekran może wyglądać tak:

```
Na jakiej wysokosci lecimy ? [w metrach] : 100
100 metrow to za nisko !
```

```
Masz 1 punktow karnych
Popraw sie !
```

Zauważ jak prosto wypisuje się na ekranie wartość zmiennej `wys`. Wystarczyła instrukcja

```
cout << wys ;
```

Wybór wielowariantowy

Przy użyciu słowa `else` mieliśmy więc możliwość dwuwariantowego wyboru: Robimy *to*, w przeciwnym razie robimy *tamto*.

Możemy jednak pójść dalej – koło słowa `else` możemy postawić następną instrukcję `if`. Dzięki temu zyskamy możliwość wyboru wielowariantowego

```
if(warunek1) instrukcja1 ;
else if (warunek2) instrukcja2 ;
else if (warunek3) instrukcja3 ;
else if (warunek4) instrukcja4 ;
```

(Inną możliwością wykonania wyboru wielowariantowego jest instrukcja `switch`, o której niebawem.)

2.3 Instrukcja `while`

Instrukcja `while` †) ma formę:

```
while(wyrażenie) instrukcja1 ;
```

Najpierw oblicza się wartość wyrażenia. Jeśli wynik jest zerowy, wówczas instrukcja1 nie jest wcale wykonywana. Jeśli jednak wartość wyrażenia jest niezerowa (prawda), wówczas wykonywana jest instrukcja1, po czym ponownie obliczana jest wartość wyrażenia. Jeśli nadal wartość tego wyrażenia jest niezerowa, wówczas ponownie wykonywana jest instrukcja1, i tak dalej, dopóki (while!) wyrażenie ma wartość niezerową. Jeśli w końcu kiedyś obliczone wyrażenie będzie miało wartość zerową, wówczas dopiero pętla zostanie przerwana.

☞ Zwracam uwagę, że obliczenie wartości wyrażenia odbywa się **przed** wykonaniem instrukcji1

```
#include <iostream.h>
main()
{
int ile ;

     cout << "Ile gwiazdek ma miec kapitan ? : " ;
     cin >> ile ;
```

†) while – ang. podczas gdy, dopóki (czytaj: „łajl")

```
        cout << "\n No to narysujmy wszystkie "
             << ile << " : " ;

        // petla while rysujaca gwiazdki
        while(ile)
        {
             cout << "*" ;
             ile = ile - 1 ;
        }
        // na dowod ze mial prawo przerwac petle
        cout << "\n Teraz zmienna ile ma wartosc " << ile
    }
```

A oto przykładowy wygląd ekranu po wykonaniu tego programu

```
Ile gwiazdek ma miec kapitan ? : 4
 No to narysujmy wszystkie 4 : ****
 Teraz zmienna ile ma wartosc 0
```

2.4 Pętla do...while...

Słowa te oznaczają po angielsku: Rób... Dopóki...†) Pętla taka ma formę

do instrukcja1 while(*wyrażenie*) ;

Czyli jakby po polsku

rób instrukcja1 dopóki (*wyrażenie*) ;

Działanie jej jest takie: Najpierw wykonywana jest instrukcja1. Następnie obliczona zostaje wartość wyrażenia. Jeśli jest ono niezerowe (prawda), to wykonanie instrukcji1 zostanie powtórzone, po czym znowu obliczone zostanie wyrażenie... i tak w kółko, *dopóki* wyrażenie będzie różne od zera.

Jak widać działanie tej pętli przypomina tę opisaną poprzednio. Różnica polega tylko na tym, że wartość wyrażenia obliczana jest nie przed, ale **po** wykonaniu instrukcji1. Wynika stąd, że instrukcja1 zostanie wykonana co najmniej raz. Czyli nawet wtedy, gdy wyrażenie nie będzie nigdy prawdziwe.

Na przykład:

```
#include <iostream.h>
main()
{
char litera ;
     do {
          cout << "Napisz jakas litere : " ;
          cin >> litera ;
          cout << "\n Napisales : " << litera << " \n" ;
```

†) (czytamy: „du...łajl")

```
    }while(litera != 'K');                          // ❶

    cout << "\n Skoro Napisales K to konczymy !" ;
}
```

A oto przykładowy wygląd ekranu po wykonaniu tego programu

```
Napisz jakas litere : A
 Napisales : A
Napisz jakas litere : K
 Napisales : K

 Skoro Napisales K to konczymy !
```

Program nasz oczekuje na napisanie litery, pętla wczytywania liter odbywa się *dopóki* nie podamy litery K (wielkiej). Wtedy to wykonywanie pętli zakończy się. Zwracam uwagę – pętla wczytująca znaki zostanie wykonana przynajmniej raz.

W programie – w miejscu, które oznaczyłem jako ❶ – pojawił się w wyrażeniu nieznany nam jeszcze dotąd operator != który oznacza: „różny od ...". Zatem zapis

```
while(litera != 'K')
```

rozumiany jest jako „dopóki `litera` jest różna od K ". Temu i innym operatorom przyjrzyjmy się dokładnie później.

2.5 Pętla `for`

Ma ona formę

`for(`*instr_ini* ; *wyraz_warun* ; *instr_krok*`)` treść_pętli ;

co w przykładzie może wyglądać choćby tak:

```
for(i=0 ; i < 10 ; i=i+1)
{
     cout << "Ku-ku ! " ;
}
```

Wyjaśnijmy co oznaczają poszczególne człony:

- `for` – (ang. dla...) oznacza: `dla` takich warunków rób...
- *instr_ini*- jest to instrukcja wykonywana zanim pętla zostanie po raz pierwszy uruchomiona.
    - W naszym przykładzie jest to podstawienie i = 0.
- *wyraz_warun* – jest to wyrażenie, które obliczane jest przed każdym obiegiem pętli. Jeśli jest ono różne od zera, to wykonywane zostają instrukcje będące treścią pętli.
    - U nas wyrażeniem warunkowym jest wyrażenie: i < 10 . Jeśli rzeczywiście `i` jest mniejsze od 10, wówczas wykonywana

zostaje instrukcja będąca treścią pętli, czyli wypisanie tekstu "Ku-ku!"

- *instr_krok* – to instrukcja wykonywana na zakończenie każdego obiegu pętli. Jest to jakby ostatnia instrukcja, wykonywana bezpośrednio przed obliczeniem wyrażenia *wyraz_warun*.
  - U nas jest to po prostu i = i+1

Praca tej pętli odbywa się więc jakby według takiego harmonogramu:

1) Najpierw wykonują się instrukcje inicjalizujące pracę pętli.
2) Obliczane jest wyrażenie warunkowe. Jeśli jest równe 0 – praca pętli jest przerywana.
3) Jeśli powyżej okazało się, że wyrażenie było różne od zera, wówczas wykonywane zostają instrukcje będące treścią pętli.
4) Po wykonaniu treści pętli wykonana zostaje instrukcja *inst_krok*, po czym powtarzana jest akcja 2).

Oto kilka ciekawostek:

instr_ini - nie musi być tylko jedną instrukcją. Może być ich kilka, wtedy oddzielone są przecinkami. Podobnie w wypadku *instr_krok*

Wyszczególnione elementy: *instr_ini, wyraz_warun, instr_krok* - nie muszą wystąpić. Dowolny z nich można opuścić, zachowując jednak średnik oddzielający go od sąsiada.

Opuszczenie wyrażenia warunkowego traktowane jest tak, jakby stało tam wyrażenie zawsze prawdziwe.

Tak więc zapis

```
for( ; ; ){
      .....
}
```

Jest nieskończoną pętlą. Inny typ nieskończonej pętli to oczywiście:

```
while(1){
      .....
}
```

Przykład

Przyjrzyjmy się pętli `for` w programie

```
#include <iostream.h>
main()
{
int i ,              // licznik
     ile ;           // liczba pasazerów

     cout << "Stewardzie, ile leci pasazerów ? " ;
     cin >> ile ;
```

```
        for(i = 1 ; i <= ile ; i = i + 1)
        {
             cout << "Pasazer nr " << i
                  << " prosze zapiac pasy ! \n" ;
        }
        cout << "Skoro wszyscy juz zapieli, to ladujemy. " ;
}
```

Jeśli w trakcie wykonywanie programu steward odpowie, że leci 4 pasażerów to

na ekranie pojawi się:

```
Stewardzie, ile leci pasazerów ? 4
Pasazer nr 1 prosze zapiac pasy !
Pasazer nr 2 prosze zapiac pasy !
Pasazer nr 3 prosze zapiac pasy !
Pasazer nr 4 prosze zapiac pasy !
Skoro wszyscy juz zapieli, to ladujemy.
```

2.6 Instrukcja `switch`

`Switch` †)- jak sama nazwa sugeruje – służy do podejmowania wielowariantowych decyzji. Przyjrzyjmy się przykładowemu fragmentowi programu.

```
int ktory ;
// .............
cout << "Kapitanie, ktory podzespol sprawdzic ? \n"
     << "nr 10 - Silnik \nnr 35 - Stery \nnr 28 - radar\n"
     << "Podaj kapitanie numer : " ;
cin >> ktory ;
switch(ktory)
{
     case 10 :
          cout << "sprawdzamy silnik \n" ;
          break ;

     case 28 :
          cout << "sprawdzamy radar \n" ;
          break ;

     case 35 :
          cout << "sprawdzamy stery \n" ;
          break ;

     default :
          cout << "Zazadales nr " << ktory
               << " - nie znam takiego ! " ;
          break ;
}
```

†) switch - ang. przełącznik (czytaj: „słicz")

W wypadku, gdy kapitan (czyli Ty!) odpowie, że numer 35, to
na ekranie będzie następujący tekst:

```
Kapitanie, ktory podzespol sprawdzic ?
nr 10 - Silnik
nr 35 - Stery
nr 28 - radar
Podaj kapitanie numer : 35
sprawdzamy stery
```

Jeśli jednak zażąda podzesopłu nr 77 to na ekranie zobaczy:

```
Kapitanie, ktory podzespol sprawdzic ?
nr 10 - Silnik
nr 35 - Stery
nr 28 - radar
Podaj kapitanie numer : 77
Zazadales nr 77 - nie znam takiego !
```

Oto jak taka instrukcja `switch` działa: Obliczane jest wyrażenie umieszczone w nawiasie przy słowie `switch`

```
switch(wyrażenie)
{
     case wart1:
                instrA ;
                break ;

     case wart2 :
                instrB ;
                break ;

     default :
                instrC ;
                break ;
}
```

Jeśli jego wartość odpowiada którejś z wartości podanej w jednej z etykiet `case`†), wówczas wykonywane są instrukcje począwszy od tej etykiety. Wykonywanie ich kończy się po napotkaniu instrukcji `break`††). Powoduje to wyskok z instrukcji `switch` – czyli jakby wyjście poza jej dolną klamrę.

Jeśli wartość wyrażenia nie zgadza się z żadną z wartości podanych przy etykietach `case`, wówczas wykonują się instrukcje umieszczone po etykiecie `default`†††). U nas etykieta ta znajduje się na końcu instrukcji `switch`, jednak może być w dowolnym miejscu, nawet na samym jej początku. Co więcej, etykiety `default` może nie być wcale. Jeśli wartość wyrażenia nie zgadza się z żadną z wartości przy etykietach `case`, a etykiety `default` nie ma wcale, wówczas opuszcza się instrukcję `switch` nie wykonując niczego.

†) case – ang. wypadek, sytuacja (czytaj: „kejs")
††) break – ang. przerwij (czytaj: „brejk")
†††) default – ang. domniemanie (czytaj: „difolt")

Instrukcji następujących po etykiecie `case` nie musi kończyć instrukcja `break`. Jeśli jej nie umieścimy, to zaczną się wykonywać instrukcje umieszczone pod następną etykietą `case`. Konkretniej: w naszym ostatnim programie brak instrukcji `break` w `case 10` spowodowałby, że po wykonywaniu instrukcji dla `case 10` nastąpiłoby wykonywanie instrukcji z `case 28`.

Nie jest to nieudolność języka C++. Czasem się to przydaje. Lepiej przecież, gdy programista sam może zadecydować czy przerwać (`break`) wykonywanie danych instrukcji, czy też kontynuować. Czasem więc celowo nie umieszczamy instrukcji `break` .

Np.

```
switch(nr)
{
     case 3 :  cout << "*" ;
     case 2 :  cout << "-" ;
     case 1 :  cout << "!" ;
               break ;
}
```

Zależnie od wartości zmiennej `nr` możliwe są następujące wydruki na ekran:

dla nr = 3 `*-!`
dla nr = 2 `-!`
dla nr = 1 `!`
dla innego n nic się nie wydrukuje

2.7 Instrukcja `break`

Zapoznaliśmy się powyżej działaniem instrukcji `break` - polegającym na przerwaniu wykonywania instrukcji `switch`. Jest jeszcze inne, choć podobne działanie `break` w stosunku do instrukcji pętli: `for`, `while`, `do...while`. Instrukcja ta powoduje natychmiastowe przerwanie wykonywania tych pętli.

Jeśli mamy do czynienia z kilkoma pętlami – zagnieżdżonymi jedna wewnątrz drugiej, to instrukcja `break` powoduje przerwanie tylko tej pętli, w której bezpośrednio tkwi. Jest to więc jakby przerwanie z wyjściem tylko o jeden poziom wyżej.

Oto jak instrukcja `break` przerwie pętlę `while`:

```
int i = 7 ;
while(1)
{
     cout << "Petla, i = " << i << "\n" ;
     i = i -1 ;
     if(i < 5){
          cout << "Przerywamy !" ;
          break ;
     }
}
```

Wykonanie tego fragmentu programu spowoduje wypisanie na ekranie

```
Petla, i = 7
Petla, i = 6
Petla, i = 5
Przerywamy !
```

A oto przykład z zagnieżdżonymi pętlami.

```
int i, m ;
int dlugosc_linii = 3 ;
for(i=0 ; i < 4 ; i = i + 1)
{
     for(m = 0 ; m < 10 ; m = m + 1)                    // ❶
     {
          cout << "*" ;
          if(m > dlugosc_linii)break ;      // tu wyskok
                                            // z for (m...)
     }
     cout      << "\nKontynuujemy zewnetrzna petle"
               << " for dla i = "
               << i << "\n" ;
}
```

Wykonanie tego fragmentu objawi się na ekranie jako:

```
*****
Kontynuujemy zewnetrzna petle for dla i = 0
*****
Kontynuujemy zewnetrzna petle for dla i = 1
*****
Kontynuujemy zewnetrzna petle for dla i = 2
*****
Kontynuujemy zewnetrzna petle for dla i = 3
```

To instrukcja `break` sprawiła, że nie było 10 gwiazdek w rzędzie, (jakby to wynikało z zapisu pętli w linii ❶). Za pomocą instrukcji `break` przerywana została ta pętla, w której `break` tkwiło **bezpośrednio**.

2.8 Instrukcja `goto`

W zasadzie na tym moglibyśmy skończyć omawianie instrukcji sterujących, gdyby nie jeszcze jedna, wstydliwa instrukcja `goto`. †) Ma ona formę:

`goto` *etykieta* ;

Po napotkaniu takiej instrukcji wykonywanie programu przenosi się do miejsca, gdzie jest dana etykieta.

†) go to - ang. idź do (czytaj: „goł tu")

> Powiedzmy jasno: używanie instrukcji goto zdradza, że się jest złym programistą. To dlatego, że instrukcji tej zawsze da się uniknąć. Program (nad-) używający instrukcji goto jest dla programisty nieczytelny, a z kolei dla kompilatora stanowi to przeszkodę w eleganckim skompilowaniu.

Z instrukcja goto wiąże się zawsze etykieta, do której należy przeskoczyć. **Etykieta** jest to nazwa, po której następuje dwukropek.

W języku C++ nie można sobie skoczyć z dowolnego punktu programu w dowolne inne. Etykieta, do której przeskakujemy, musi leżeć w obowiązującym w danej chwili tzw. zakresie ważności. (O tym pomówimy na stronie 42). Oto przykład użycia goto

```
        cout << "Cos piszemy \n" ;
        goto aaa ;                      // stad przeskok
        cout << "Tego nie wypiszemy " ;
aaa:                                    // w to miejsce
        cout << "Piszemy " ;
```

Ten fragment objawi się na ekranie jako:

```
Cos piszemy
Piszemy
```

Przypominam, że to, iż w naszym przykładzie etykietę wysunąłem bliżej lewego marginesu, nie ma żadnego znaczenia dla kompilatora. Jemu jest to wszystko jedno. Nam jednak nie. Dla nas chyba lepiej, by etykieta bardziej rzucała się w oczy, łatwiej ją odszukać w tekście programu.

Mimo tej niesławy są sytuacje, gdy instrukcja goto się przydaje

Na przykład dla natychmiastowego opuszczenia wielokrotnie zagnieżdżonych pętli. Instrukcją break przerwać możemy przecież tylko tę najbardziej zagnieżdżoną pętlę. Dzięki instrukcji goto możemy w wyjątkowych wypadkach od razu wyskoczyć na zewnątrz. Na zewnątrz – oznacza tu – na zewnątrz tych zagnieżdżonych pętli. (Zawsze jednak tylko w ramach tego bloku programu, w którym znana jest etykieta).

Oto przykład:

```
int m, i, k ;
while(m < 500)
{
     while(i < 20)
     {
          for(k = 16 ; k < 100 ; k = k+4 )
          {
               // .............
               // tu wyskoczymy !
               if(blad_operacji)goto berlin ;          // ❶
          }
     }
}
```

```
berlin :                                    // etykieta,    ❷
     cout << "Po opuszczeniu wszystkich petli " ;
```

Jeśli w jakiś sposób w trakcie pracy tych pętli zmienna blad_operacji przybierze wartość niezerową, wówczas nastąpi wyskok ❶ z pętli, i wykonywane będą instrukcje począwszy od etykiety berlin ❷

2.9 Instrukcja continue

Instrukcja continue przydaje się wewnątrz pętli for, while, do...while. Powoduje ona zaniechanie realizacji instrukcji będących treścią pętli, jednak (w przeciwieństwie do instrukcji break) sama pętla nie zostaje przerwana. Continue przerywa tylko ten obieg pętli i zaczyna następny, kontynuując pracę pętli. Oto przykład :

```
int k ;
for(k = 0 ; k < 12 ; k = k + 1)
{
     cout << "A" ;
     if(k > 1) continue ;
     cout << "b" << endl ;
}
```

W rezultacie wykonania tego fragmentu programu na ekranie pojawi się:

```
Ab
Ab
AAAAAAAAAA
```

Innymi słowy napotkanie instrukcji continue odpowiada jakby takiemu użyciu instrukcji goto

```
for(...)
{
     ...
     continue ;          // goto sam_koniec
     ...
sam_koniec:
}
```

czyli skokowi do etykiety stojącej przed zamykającą pętlę klamrą. W rezultacie komputer „pomyśli", że już wykonał treść pętli, i przystąpi do wykonywania następnego obiegu.

Identycznie zachowa się wobec pętli while

```
while(warunek)
{
          ...
          continue ;          // goto sam_koniec ;
          ...
sam_koniec :
}
```

Czy też pętli `do..while`

```
do
{
          ...
          continue ;          // goto sam_koniec ;
          ...
sam_koniec :
}while(warunek) ;
```

2.10 Klamry w instrukcjach sterujących

Pamiętasz, mówiłem kiedyś, że język C++ jest językiem o dowolnym formacie. Wynika z tego także, iż klamry { } w naszych instrukcjach sterujących możemy stawiać w różnych miejscach. Oto kilka wariantów

```
while(i < 4) {                    ❶
     ...
}
-------------------------------------
while(i < 4)                      ❷
{
     ...
}
-------------------------------------
while(i < 4)                      ❸
     {
          ...
     }
```

Wszystkie trzy sposoby są jednakowo dobre. Namawiam jednak do przyjęcia jednego standardu.

Dlaczego to takie ważne? Otóż jednym z najczęstszych błędów jest zapomnienie o zamknięciu klamry. Gdy stosujemy zapis ❷ i ❸ to wyraźnie widzimy, które klamry należą do siebie. W sposobie ❶ tego nie widać, ale za to jest on o jedną linjkę krótszy. Osobiście stosuję zapis ❶ lub ❷. W tej książce używał będę zapisu ❷.

Dlaczego nie przeszkadza mi wspomniana wada zapisu ❶ ?

Z dwóch powodów:

1) W moim edytorze jest komenda, która pozwala mi odszukać drugi nawias: pokazuję na nawias lewy, a edytor odszukuje mi odpowiadający mu prawy. To samo w wypadku klamer. Jeśli nawet pogubię się z tymi klamrami w długim programie, to dzięki tej opcji łatwo znaleźć błąd. (Sprawdź czy i w Twoim edytorze jest podobna komenda).

2) Mam sposób, który prawie całkowicie pozwala mi uniknąć błędu.

Dawniej robiłem mianowicie tak:

Pisałem np: `if(warunek)`, potem otwierałem klamrę i długo pisałem wszystkie instrukcje, po czym klamrę uroczyście zamykałem – o ile tylko jeszcze pamię-

tałem, że mam jakąś klamrę zamknąć. Łatwo o tym zapomnieć, szczególnie wtedy, gdy we wnętrzu są zagnieżdżone inne instrukcje z klamrami.

Teraz robię tak:

> Piszę: `if` (warunek), otwieram klamrę, przechodzę dwie linijki niżej i zamykam od razu klamrę, po czym wracam linijkę wyżej i dopiero przystępuję do pisania instrukcji.

Sposób jest naprawdę dobry. Od czasu, gdy go stosuję nawet kilkakrotne zagnieżdżanie instrukcji `while`, `for`, `do` – nie jest mi straszne. Spróbuj.

Mimo wszystko zapewne czasem pogubisz się w stawianiu klamer. Radzę Ci: spróbuj to zrobić świadomie po to, by się przekonać, jak na to zareaguje Twój kompilator. Albowiem jego komunikat o błędzie wcale nie musi mówić o braku klamry.

W działającym programie usuń lub dodaj jedną z zamykających klamer i spróbuj to skompilować. Komunikat o błędzie może być w stylu „zła deklaracja funkcji" albo coś w tym rodzaju.

Dobrze to zapamiętaj, bo gdy potem otrzymasz taki sam komunikat – będziesz już wiedział, że przyczyny można szukać również w nawiasach klamrowych.

3 Typy

3.1 Deklaracje typów

Każda nazwa w C++ zanim zostanie użyta, musi zostać zadeklarowana. Deklarujemy mianowicie, że opisuje ona obiekt jakiegoś typu: całkowitego, zmiennoprzecinkowego, itd. – (np. `int, float`). To informuje kompilator, jak ma w przypadku napotkania tej nazwy postępować.
Wyjaśnijmy to na przykładzie. Załóżmy, że kompilator napotkał w naszym programie wyrażenie

```
a + b
```

Jest to dodawanie, zatem trzeba uruchomić specjalny podprogram zajmujący się dodawaniem. (Taki podprogram nazywa się też czasem operatorem dodawania.)

W związku z tym, że w komputerze liczby całkowite przechowywane są inaczej niż liczby zmiennoprzecinkowe – zatem operator dodawania musi inaczej postępować w stosunku do liczb całkowitych, a inaczej w stosunku do liczb zmiennoprzecinkowych.

Napotykając ów zapis – kompilator musi więc dokładnie wiedzieć czy symbol `a` oznacza u nas liczbę całkowitą, czy zmiennoprzecinkową. Skąd to będzie wiedział? Właśnie z deklaracji. Deklarując wcześniej zmienną `a` jako typu integer, czyli pisząc

```
int a ;
```

powiedzieliśmy kompilatorowi tak:

> Jakbyś napotkał nazwę `a` to wiedz, że oznacza ona zmienną typu `int`.

Tutaj deklaracja jest równocześnie definicją. Oznacza to, że nie tylko, iż informujemy kompilator o tym, co oznacza nazwa `a`, lecz także żądamy w tym miejscu, by zarezerwował w pamięci obszar na tę zmienną – czyli powołał ją do życia.

Definicja mówi więc kompilatorowi:

> A teraz zarezerwuj mi w pamięci miejsce na obiekt typu całkowitego o nazwie a

Nie zawsze jednak chodzi o powołanie do życia.

Może być przecież sytuacja, gdy zmienna ta już gdzieś w programie istnieje, a tu chcemy tylko poinformować kompilator o jej typie.

```
extern int a ;
```

Słowo `extern` (ang. zewnętrzny) informuje kompilator, obiekt typu `int` o nazwie `a` już gdzieś istnieje, na przykład na zewnątrz pliku, którym zajmuje się właśnie kompilator. To „na zewnątrz" może oznaczać też jakąś funkcję biblioteczną, którą dołączymy dopiero na etapie linkowania. W trakcie kompilacji naszego pliku, kompilator musi już jednak wiedzieć jakiego typu jest ta „zewnętrzna" zmienna `a`.

Powtórzmy więc różnicę między deklaracją a definicją

> Deklaracja – informuje kompilator, że dana nazwa reprezentuje obiekt jakiegoś typu, ale nie rezerwuje dla niego miejsca w pamięci.
>
> Definicja zaś – dodatkowo rezerwuje miejsce. Definicja jest miejscem, gdzie powołuje się obiekt do życia.

Oczywiście definicja jest przy okazji zawsze także deklaracją, bo przecież jeśli rezerwuje miejsce w pamięci, to musi ona kompilatorowi wyjaśnić na co rezerwuje.

Deklarować obiekt można w tekście programu wielokrotnie. Definiować go (powoływać go do życia) można tylko raz.

Studium języków obcych

Różnicę między deklaracją a definicją łatwo zrozumieć i zapamiętać tłumacząc sobie dosłownie te słowa.

Deklaracja :

od łacińskiego clarus: jasny, zrozumiały. Zresztą po polsku też mówi się: „klarować coś komuś". De-klarować to jakby: wy-jaśniać. Deklaracja nazwy N jest więc tylko wyjaśnieniem kompilatorowi co dana nazwa oznacza.

Definicja :

pochodzi od łacińskiego słowa finis – koniec, granica. Definiować – to jakby zakreślać granicę. W naszym wypadku tę granicę wykreśla się wokół komórek pamięci, które są przydzielane w ten sposób obiektowi. Mówi się: ta, tamta i jeszcze ta komórka – stają się od tej pory obiektem o danej nazwie N. W ten sposób odbyły się narodziny obiektu o nazwie N.

Oto przykłady definicji i deklaracji:

```
int liczba ;            // definicja + deklaracja
extern int licznik ;    // deklaracja (tylko ! )
```

3.2 Systematyka typów z języka C++

W deklaracji określa się, że dany obiekt jest jakiegoś typu. Jakie mamy do dyspozycji typy? Jest ich dużo. Wprowadźmy więc pewien porządek.

Typy z języka C++ można podzielić dwojako:

Pierwszy podział to podział na:

- typy fundamentalne
- typy pochodne, które są jakby wariacjami na temat typów fundamentalnych

Drugi podział to na:

- typy wbudowane (ang. build-in) – czyli takie, w które język C++ jest wyposażony,
- typy zdefiniowane przez użytkownika – czyli typy, które możesz sobie wymyślić samemu. Ta cecha jest chyba jednym z najlepszych pomysłów w języku C++

Zajmiemy się teraz typami wbudowanymi. Natomiast typom definiowanym przez użytkownika poświęcona jest dalsza część książki.

3.3 Typy fundamentalne

Są identyczne jak w klasycznym C. Oto ich lista:

Typy reprezentujące liczby całkowite

`short int`	*inaczej:*	`short`
`int`		
`long int`	*inaczej:*	`long`

oraz tak zwany typ wyliczeniowy `enum`, o którym porozmawiamy niebawem. (Str. 52).

Typ reprezentujący obiekty zadeklarowane jako znaki alfanumeryczne

`char`

Wszystkie powyższe typy mogą być w dwóch wariantach – ze znakiem i bez znaku. Do wybrania wariantu posługujemy się modyfikatorem `signed` lub `unsigned`[†)]

np.

†) signed, unsigned – ang. ze znakiem, bez znaku (czytaj: „sajned", „ansajned")

```
signed int
unsigned int
```

Wyposażenie typu w znak sprawia, że może on reprezentować liczbę ujemną i dodatnią. Typ bez znaku reprezentuje liczbę dodatnią.

Przez domniemanie przyjmuje się, że zapis

```
int a ;
```

oznacza, że chodzi nam o typ

```
signed int a ;
```

czyli typ ze znakiem.

Natomiast w wypadku typu `char` sprawa nie jest tak prosta. To, czy przez domniemanie będziemy mieli `signed` czy `unsigned` - zależy od typu kompilatora czy komputera. Mówimy krótko: zależy to od implementacji.

Typy reprezentujące liczby zmiennoprzecinkowe

```
float
double
long double
```

umożliwiają pracę na liczbach rzeczywistych z różną dokładnością.†)

Zastanawiasz się zapewne po co są aż trzy typy reprezentujące liczby całkowite oraz trzy typy reprezentujące liczby zmiennoprzecinkowe.

Chodzi o to, by można było lepiej wykorzystać możliwości danego typu komputera. Zależnie od tego, ile dany komputer przydziela komórek pamięci na zapis danej liczby – otrzymujemy mniejszą lub większą precyzję obliczeń.

Przyjrzyjmy się jak to załatwiane jest na różnych komputerach.

typ	**Komputer**	
	IBM PC/AT	VAX
`short`	2 bajty	2 bajty
`int`	2 bajty	4 bajty
`long`	4 bajty	4 bajty
`float`	4 bajty	4 bajty
`double`	8 bajtow	8 bajtów
`long double`	10 bajtów	8 bajtów

Za lepszą dokładność płaci się dłuższym czasem obliczeń, dlatego zapewnienie programiście aż 3 typów dla liczb całkowitych daje możliwość wyboru między dokładnością, a szybkością obliczeń.

†) double - (czytaj: „dabl") ang. podwójny. Zapewne od: double precision - podwójna dokładność.

Jak widać z zestawienia – sposób przechowywania liczby może zależeć od typu komputera. O tym, jak zapisuje dany typ Twój komputer, będziesz się mógł przekonać stosując operator `sizeof` (rozmiar). Operator ten przedstawimy niebawem (str. 69).

3.3.1 Definiowanie obiektów „w biegu".

W naszych dotychczasowych programach już kilkakrotnie spotkaliśmy się z definicjami zmiennych. W niektórych językach programowania definicje obiektów powinny nastąpić przed wykonywanymi instrukcjami. Tak też jest w klasycznym C.

W C++ zasada ta nie obowiązuje. Obiekt można zdefiniować „w biegu", „w locie" [ang: on flight], między dwoma instrukcjami – wtedy, gdy uznamy, że jest on nam właśnie potrzebny.

Oto przykład:

```
#include <iostream.h>
main()
{
     float dlugosc_fali ;                                  // ❶
     // ..
     cout << "Podaj wspolczynnik zalamania : " ;
     float wspolczynnik ;                                  // ❷
     cin >> wspolczynnik ;
     cout << " Zrozumialem, wspolczynnik ma byc : "
          << wspolczynnik ;
     // .... dalsze obliczenia
}
```

W tym przykładzie widzimy, że przed instrukcjami wykonywanymi w funkcji `main` jest definicja obiektu typu `float` ❶. Jest to definicja w starym, klasycznym stylu.

Tymczasem w trakcie pisania programu dochodzimy do wniosku, że potrzebujemy jeszcze jednego obiektu `float` – na współczynnik załamania. Możemy wrócić do ❶ i tam dopisać następną definicję, ale możemy też zrobić to tu, gdzie sobie o zmiennej przypomnieliśmy ❷, lub gdzie wynikła konieczność jej istnienia. Robimy więc tę definicję w biegu, nie przerywając normalnego toku pisania programu. Ten sposób jest nawet logiczniejszy.

Co prawda, w naszym przykładzie wartość wczytywaliśmy z klawiatury, jednak w innych wypadkach — często w linijce, w której wynikła konieczność istnienia obiektu, już dokładnie wiemy, jaką wartość powinien on od razu zawierać. Możemy więc nie tylko zdefiniować obiekt, ale od razu (w tej samej instrukcji), nadać mu wartość. Takie postępowanie daje szybszy program (bo to mniej pracy niż wariant z powtórnym wracaniem do zmiennej, by jej nadawać wartość).

3.4 Stałe dosłowne

W programach często posługujemy się stałymi. Mogą to być liczby, mogą to być znaki (litery) albo ciągi znaków (z angielska: *stringi*). Stałych tych używamy na przykład, by wstawić je do jakichś zmiennych

```
x = 10.52 ;
```

lub wtedy, gdy występują one w wyrażeniach arytmetycznych

```
i = i + 5 ;        // powiekszenie obiektu i o stałą 5
```

albo wtedy, gdy chcemy coś z nimi porównać

```
if(m > 12)
```

Zwracam uwagę, że w tym miejscu chodzi nam o stałe dosłowne czyli o sam *zapis* liczby, a nie o jakiś obiekt, który ma akurat taką wartość. Zapoznamy się tu ze sposobami zapisywania takich stałych.

3.4.1 Stałe będące liczbami całkowitymi

Stałe takie zapisujemy tak, jak do tego przywykliśmy w szkole

17 -33 0 1000 itd.

Jeśli natomiast zapis stałej zaczniemy od cyfry 0 (zero), to kompilator zrozumie, że zastosowaliśmy zapis liczby w systemie ósemkowym (oktalnym).

010	-	czyli dziesiątkowo 8
014	-	czyli dziesiątkowo 8+4 = 12
091	-	błąd, w zapisie ósemkowym cyfra 9 jest nielegalna

Jeżeli stała zaczyna się od 0x (zero i x) to kompilator uzna, że w stosunku do stałej zastosowaliśmy zapis szesnastkowy (heksadecymalny) (heXadecymalny).

0x10	-	czyli dziesiątkowo 1 *16 + 0 = 16
0xa1	-	czyli dziesiątkowo 10*16 + 1 = 161
0xff	-	czyli dziesiątkowo 15*16 + 15 = 255

Nie muszę chyba przypominać, że występujące w zapisie znaki `a, b, c, d, e, f` oznaczają odpowiednio `10, 11, 12, 13, 14, 15` w zapisie dziesiątkowym. W zapisie można się posługiwać zarówno wielkimi literami `X, A, B, C, D, E, F,` jak i małymi.

Stałe całkowite traktuje się tak, jak typ `int`, chyba, że reprezentują tak wielkie liczby, które nie zmieściłyby się w `int`. Wówczas stała taka jest typu `long`.

Można świadomie zmienić typ nawet niewielkiej stałej – z typu `int` na typ `long`. Robi się to przez dopisanie na końcu liczby: litery `L` (lub `l`)

```
0L   200L
```

Mimo, że liczby te wystarczająco dobrze mieszczą się w typie `int` - dopisana na końcu litera L sprawia, że są typu `long`. Osobiście zawsze używam tu wielkiej litery L, gdyż mała za bardzo przypomina jedynkę.

Jeśli chcemy by dana stała miała typ `unsigned`, to sprawi to dopisanie na końcu litery `u`

```
277u
```

Przypisek `u` może wystąpić razem z przypiskiem `L`

```
50uL
```

wówczas oznacza to, że dana stała ma być typu `unsigned long`.

Oto przykład zapisu tych stałych w programie:

```
#include <iostream.h>
main()
{
int i     ;                                    // definicja obiektu
int k, n, m, j ;

     i = 5 ;
     k = i + 010 ;                             // czyli 5 + 8

     cout << "k= " << k << endl ;

     m = 100 ;
     n = 0x100 ;
     j = 0100 ;

     cout << "m+n+j= " << (m+n+j) << endl ;

     cout << "Wypisujemy : " << 0x22 << " "
          << 022 << " " << 22 << endl ;

}
```

W wyniku wykonania na ekranie pojawi się

```
k= 13
m+n+j= 420
Wypisujemy : 34 18 22
```

Zauważ, że zastosowanie któregoś z zapisów (dziesiątkowego, oktalnego, hexadecymalnego) jest tylko jednym ze sposobów powiedzenia o jaką liczbę nam chodzi. Komputer i tak przetłumaczy to sobie na swój własny sposób (binarny). To tak, jakbyśmy rachmistrzowi powiedzieli czasem `trzy`, czasem `drei`, czasem `three`.

3.4.2 Stałe reprezentujące liczby zmiennoprzecinkowe

Stałe takie zapisać można na dwa sposoby. Pierwszy to normalny zapis liczby z kropką dziesiętną.

```
12.3      3.1416    -1000.3   -12.
```

Drugi zapis jest nazywany notacją naukową (scientific notation). W zapisie tym występuje litera `e`, po której następuje wykładnik potęgi o podstawie 10. A zatem:

8e2	oznacza	8	$* 10^2$	czyli	800
10.4e8	oznacza	10.4	$* 10^8$	czyli	1 040 000 000
5.2e-3	oznacza	5.2	$* 10^{-3}$	czyli	0.0052

Stałe takie traktuje się tak, jakby były typu `double`

Oto przykład użycia takich stałych:

```
#include <iostream.h>
main()
{
float pole, promien ;

     promien = 1.7 ;
     pole = promien * promien * 3.14 ;
     cout << "\nPole kola o promieniu "
          << promien << " wynosi " << pole ;

     promien = 4.1e2 ;
     pole = promien * promien * 3.14 ;
     cout << "\nPole kola o promieniu "
          << promien << " wynosi " << pole ;
}
```

W wyniku wykonania tego programu na ekranie pojawi się

```
Pole kola o promieniu 1.7 wynosi 9.0746
Pole kola o promieniu 410 wynosi 527834
```

3.4.3 Stałe znakowe

Stałe znakowe są to stałe reprezentujące na przykład znaki alfanumeryczne. Zapisuje się je ujmując dany znak w dwa apostrofy

`'a'` - oznacza literę a
`'7'` - oznacza cyfrę 7 (cyfrę, nie liczbę)

Oto przykład:

```
char znak ;
     znak = 'A' ;
```

Oczywiście wiesz na pewno, że komputer nie potrafi przechowywać w swojej pamięci żadnej litery 'A'. Może jednak przechowywać liczby. Dlatego wszystkie litery alfabetu i znaki specjalne zostały po prostu ponumerowane i to ten numer (kod) danego znaku jest przechowywany w pamięci.

Są różne sposoby numerowania (kodowania) znaków. Jednym z najbardziej popularnych jest chyba kod ASCII.[†)] *Z tabelą kodów ASCII, czyli tym, jakimi liczbami reprezentowane są jakie znaki – spotkałeś się już na pewno przy programowaniu w innych znanych Ci językach programowania.*

†) (czytaj „aski")- jest to skrót od: American Standard of Code Interchanged Information

Są jednak takie znaki, których nie da się wprost umieścić między apostrofami. Służą one do sterowania wypisywaniem tekstu – np. przejście do nowej strony, tabulator, znak nowej linii. Pomagamy sobie wtedy za pomocą kreski ukośnej, tzw. ukośnika [ang. backslash][†]. Obok niego stawiamy umowną literę, która przypomina znaczenie danego znaku.

`'\b'` -	cofacz	(ang. **B**ackspace)
`'\f'` -	nowa strona	(ang. **F**orm feed)
`'\n'` -	nowa linia	(ang. **N**ew line)
`'\r'` -	powrót karetki	(ang. carriage **R**eturn)
`'\t'` -	tabulator poziomy	(ang. **T**abulator)
`'\v'` -	tabulator pionowy	(ang. **V**ertical tabulator)
`'\a'` -	sygnał dzwiękowy	(**A**larm)

Mimo, że widzimy kilka znaków, zapis reprezentuje jeden znak. Czytamy to tak:

Dwa najbardziej zewnętrzne apostrofy mówią nam, że mamy do czynienia ze znakiem. W środku zaś czytamy \ – bekslesz: acha, będzie to coś niezwykłego. Potem następuje np. litera f. Skoro coś niezwykłego, to patrzymy co na liście niezwykłości oznacza litera `f` – jest to skrót od form feed (nowa strona). Proste.

Ponieważ pomagaliśmy sobie stosując znaki takie jak apostrofy, kreska ukośna, więc jak zakodować sam znak apostrofu? Za pomocą trzech apostrofów?

```
char c = ''' ;                    // błąd
```

Nie. kompilator podejrzewałby błąd, dlatego musimy dodać kreskę ukośną

```
char c = '\'' ;                   // apostrof
```

Zapis ten rozumiemy tak: dwa zewnętrzne apostrofy – czyli wewnątrz jest znak. Potem bekslesz czyli „uwaga!", a potem apostrof. Bekslesz jest w tym wypadku ostrzeżeniem, bo znak apostrofu jest już raz w tej konstrukcji używany w innym znaczeniu (ogranicznik).

Poniżej widać, że podobnie radzimy sobie w wypadku bekslesza, cudzysłowu i w kilku innych wypadkach.

`'\\'`	-	bekslesz
`'\''`	-	apostrof
`'\"'`	-	cudzysłów
`'\0'`	-	NULL, znak o kodzie 0
`'\?'`	-	pytajnik

Wśród znaków (już niealfanumerycznych – bo nie da się ich zapisać) jest jeszcze jeden bardzo wyjątkowy. Jest to znak o kodzie 0 zwany znakiem NULL. Na liście widzimy sposób jego zapisu.

Można także stałe znakowe zapisywać bezpośrednio – podając między apostrofami liczbowy kod znaku, zamiast samego znaku. Kod znaku musi być liczbą w zapisie ósemkowym lub szesnastkowym.

†) (czytaj: „bekslesz")

Np. ponieważ w kodzie ASCII litera a reprezentowana jest przez liczbę 97 dlatego poniższe zapisy są równoważne.

`'a'` - to samo co ósemkowo `\0141`

`'a'` - to samo co szesnastkowo `0x61`

3.4.4 Stałe tekstowe, albo po prostu stringi

W językach programowania bardzo często posługujemy się stałymi tekstowymi będącymi ciągami znaków. Zwane są one czasem napisami, czasem łańcuchami znaków.

Spotkaliśmy się już z takimi stałymi:

```
"Witamy na pokladzie"
```

W języku angielskim taka stała tekstowa nazywa się to krótko: string.

W całej tej książce pierwotnie posługiwałem się nazwą „ciąg znaków". Nazwa ta dziesiątki razy odmieniana była przez wszystkie przypadki, co nie zawsze bywało zręczne. Wreszcie zrezygnowałem. Nie znam bowiem programisty, który by na to „coś" mówił inaczej jak: *string*.

A zatem:

> String, czyli stała tekstowa, to ciąg znaków ujęty w cudzysłów.

Oto przykłady:

```
"taki string"
"Pozar na pokladzie"
"Alarm 3 stopnia"
```

Ponieważ string jest ciągiem *znaków*, więc obowiązują podobne zasady, jak opisane przy stałych znakowych: Jeśli chcemy w tekście (stringu) zastosować znak nowej linii, to wystarczy napisać `\n` w żądanym miejscu.

```
"Pozar \n na pokladzie"
```

Zdziwiłeś się dlaczego nie ma teraz dwóch apostrofów po obu stronach znaku `\n` ? Nic w tym dziwnego – nie ma także apostrofów obok liter: `p o z` itd.

Stringi są w pamięci przechowywane jako ciąg liter, a na samym końcu tego ciągu dodawany jest znak o kodzie 0, czyli znak NULL. Tak kompilator oznacza sobie koniec stringu.

Ogranicznikiem stringu są znaki cudzysłowu `" . . . . "` Ponieważ cudzysłów ma takie szczególne znaczenie dla stringu, dlatego nie można już go użyć dodatkowo wewnątrz stringu. W wypadku stałych znakowych problem ten mieliśmy z apostrofami. Do pomocy mamy jednak identyczny chwyt ze znakiem bekslesz:

```
cout << "Lecimy promem \"Columbia\" nad Oceanem Spokojnym";
```

co na ekranie pojawi się jako

```
Lecimy promem "Columbia" nad Oceanem Spokojnym
```

Mówiliśmy kiedyś, że w języku C++ w prawie każdym miejscu instrukcji można przerwać pisanie, przejść do następnej linii i kontynuować instrukcję.

To słowo „prawie" dotyczy między innymi pisania stringów. Tutaj nie można przerwać pisania. Jeśli kompilator zobaczył w linijce cudzysłów otwierający string, to musi w tej samej linijce znaleźć cudzysłów zamykający.

Co zrobić jeśli string jest tak długi, że nie mieści się w jednej linijce?

Jest na to sposób. Spójrz poniżej: w kilku linijkach zapisaliśmy tu string, który kompilator traktuje jako jedną całość.

```
"Caly ten tek"
"st jest traktowa"
"ny jako jeden dlu"          "gi string"
```

Jak widać w ostatniej linijce – nawet w tej samej linii można zamknąć cudzysłów, a potem bezpośrednio go otworzyć – i kompilator uzna to za jeden string. (Zauważ, że nie ma żadnych przecinków).

Zapamiętaj:

> Bezpośrednio przylegające do siebie stringi, kompilator łączy w jeden.

Wynika stąd też, że zamiast pisać

```
cout << "Moj drogi Kapitanie, ktory "
     << "podzespol sprawdzic ? " ;
```

można zapisać

```
cout << "Moj drogi Kapitanie, ktory "
         "podzespol sprawdzic ? " ;
```

oczywiście dlatego, że są to stringi, które bezpośrednio mogą do siebie przylegać (bo akurat nie wstawiliśmy między nimi żadnego wypisywania na ekran wartości jakiejś zmiennej).

3.5 Typy pochodne

Są to jakby wariacje na temat typów podstawowych (fundamentalnych), o których mówiliśmy poprzednio.

Są to takie typy, jak na przykład tablica czy wskaźnik.

Możemy mieć kilka „luźnych" obiektów typu `int`, ale możemy je powiązać w tablice obiektów typu `int`.

Tablica obiektów typu `int` jest typem pochodnym od typu `int`. Nie musisz się jednak tą całą systematyką przejmować, tak jak mechanik nie musi myśleć czy jego tokarka jest typem pochodnym od .... właśnie, czego?

Typy pochodne oznacza się stosując nazwę typu, od którego pochodzą, i operator deklaracji typu pochodnego. Jest to prostsze niż się wydaje.

```
int a ;              // obiekt typu int
int b[10] ;          // tablica obiektów typu int (10 -elementowa)
```

Co to jest tablica tłumaczyć chyba nie trzeba – taki typ obiektu istnieje nawet w języku BASIC czy FORTRAN. Tablicom poświęcimy specjalny rozdział.

Teraz wymienię jeszcze inne operatory do tworzenia obiektów typów pochodnych. Jednak nie przerażaj się. Wszystkie staną się jasne w najbliższych rozdziałach.

Oto lista operatorów, które umożliwiają tworzenie obiektów typów pochodnych:

[ ]	-	tablica obiektów danego typu,
*	-	wskaźnik do pokazywania na obiekty danego typu,
( )	-	funkcja zwracająca wartość danego typu,
&	-	referencja (przezwisko) obiektu danego typu.

Nie mówiliśmy jeszcze o tych typach, będzie o nich mowa w odpowiednim czasie.

Tutaj wyjaśnimy więc krótko, że:

- Tablica – to inaczej macierz, albo wektor obiektów danego typu.
- Wskaźnik – to obiekt, w którym można umieścić adres jakiegoś innego obiektu w pamięci.
- Funkcja – czyli podprogram. Jest to zapewne znane Ci z innych języków programowania.
- Referencja – to jakby przezwisko jakiegoś obiektu. Dzięki referencji na tę samą zmienną można mówić używając jej drugiej nazwy.

A oto przykłady typów fundamentalnych:

```
int a ;              // def. obiektu typu int
short int b ;        // def. obiektu typu short
float x ;            // def. obiektu typu float
```

dalej nie ma co pisać, bo jest to prymitywne. Oto przykłady typów pochodnych:

```
int t[10] ;          // tablica 10 elementów typu int
float *p ;           // wskaźnik mogący pokazać na jakiś
                     //                 obiekt typu float
char func() ;        // funkcja zwracająca (jako rezultat
                     //     wykonania)  obiekt typu char
```

Bardzo zachęcam Cię, drogi czytelniku, abyś teraz oswoił się z takimi deklaracjami, a w przyszłości nauczył się je odczytywać. Da Ci to ogromną swobodę w poruszaniu się po królestwie C++. Jeśli są ludzie, którzy nie lubią C i C++, to dlatego, że stają bezradni, gdy zobaczą takie deklaracje.

Do ćwiczeń w odczytywaniu deklaracji jeszcze wielokrotnie powrócimy.

3.5.1 Typ `void`

W deklaracjach typów pochodnych może się pojawić słowo `void`. [ang. próżny] Słowo to stoi w miejscu, gdzie normalnie stawia się nazwę typu. I tak:

```
void *p ;
```

- tutaj oznacza to, że `p` jest wskaźnikiem do pokazywania na obiekt nieznanego typu. (O tym, do czego taki wskaźnik może się przydać, powiemy sobie w rozdziale o wskaźnikach.)

```
void funkcja() ;
```

- deklaracja ta mówi, że `funkcja` nie będzie zwracać żadnej wartości.

3.6 Zakres ważności nazwy obiektu, a czas życia obiektu

Wiemy już jak zdefiniować lub zadeklarować obiekt jakiegoś typu. Przykładowo dla obiektu typu `int` robi się to instrukcją

```
int m ;
```

Zajmijmy się teraz zakresem ważności nazwy tak zdefiniowanego obiektu i czasem jego życia.

Czas życia obiektu

- to okres od momentu, gdy zostaje on zdefiniowany, (definicja przydziela mu miejsce w pamięci) – do momentu, gdy przestaje on istnieć, (a jego miejsce w pamięci zostaje zwolnione).

Zakres ważności nazwy obiektu

- to ta część programu, w której nazwa znana jest kompilatorowi.

Jaka jest różnica między tymi pojęciami?

Taka, że w jakimś momencie obiekt może istnieć, ale nie być dostępny. To dlatego, że np. znajdujemy się chwilowo poza zakresem ważności jego nazwy.

Zależnie od tego, jak zdefiniujemy obiekt, zakres ważności jego nazwy może być czworakiego rodzaju.

3.6.1 Zakres: lokalny

Zakres ważności jest lokalny, gdy świadomie ograniczamy go do kilku linijek programu. Pisząc program możemy w dowolnym momencie za pomocą dwóch klamer

```
{
    ...
}
```

utworzyć tzw. blok. Zdefiniowane w nim nazwy mają zakres ważności ograniczony tylko do tego bloku. Po prostu poza tym blokiem nazwy te nie są znane.

```
#include <iostream.h>
main()
{
                                  // tu robimy jakieś obliczenia
    {                             // <—— otwieramy lokalny blok
        int x ;                       // definiujemy jakieś zmienne
        ...                           // pracujemy na tych zmiennych
    }                             // <—— zamykamy lokalny blok
    ...                           // poza blokiem lokalne obiekty są już nieznane
}
```

Nazwa zmiennej `x` jest znana od momentu, gdy ją zdefiniowaliśmy do linijki, gdzie jest klamra } kończąca jej lokalny blok.

3.6.2 Zakres: blok funkcji

Zakres ważności ograniczony do bloku funkcji ma etykieta. Znaczy to, że jest ona znana w całej funkcji, nawet w tych linijkach funkcji, które ją poprzedzają.

Uwaga. Z faktu, że etykieta ma zakres ważności funkcji wynika prosty wniosek:

> Nie można instrukcją `goto` przeskoczyć z wnętrza jednej funkcji do wnętrza innej.

Wszystkie etykiety danej funkcji są już poza tą funkcją nieznane. Z zewnątrz tej funkcji nie można więc do nich skoczyć.

3.6.3 Zakres: obszar pliku

Na razie nasze krótkie programy mieściły się zwykle w jednym pliku dyskowym. W przyszłości będziemy pisać dłuższe, które dla wygody rozmieścimy w kilku plikach.

> Jeśli w jednym z nich, na zewnątrz jakiegokolwiek bloku (także bloku funkcji) zadeklarujemy jakąś nazwę, to mówimy wówczas, że **taka nazwa jest globalna**. Ma ona zakres ważności pliku.

Oto przykład:

```
float fff ;                  // nazwa fff jest globalna
main()
{
        //...
}
```

Jednakże taka nazwa nie jest od razu automatycznie znana w innych plikach. Jej zakres ważności ogranicza się tylko do tego pliku, w którym ją zdefiniowaliśmy. I to w dodatku jedynie od miejsca deklaracji, do końca pliku.

3.6.4 Zakres: obszar klasy

To na razie tajemnica. O tym szczegółowo porozmawiamy w rozdziale poświęconym klasom.

3.7 Zasłanianie nazw

Możemy zadeklarować nazwę lokalną, identyczną jak istniejąca nazwa globalna. Nowo zdefiniowana zmienna zasłania wtedy, w danym lokalnym zakresie, zmienną globalną. Jeśli w tym lokalnym zakresie odwołamy się do danej nazwy, to kompilator uzna to za odniesienie się do zmiennej lokalnej. Spójrzmy na prosty przykład:

```
int k = 33 ;                    // ❶ zmienna globalna (obiekt typu int)
main()
{
     cout << "Jestem w main , k =" << k << "\n" ;     // ❷
     { ////////////////////  <------------------------------------------ ❸
          int k = 10 ;                          // zmienna lokalna      ❹
          cout << " po lokalnej definicji k ="
               << k << endl ;                                           // ❺
     } ////////////////////  <------------------------------------------ ❻

     cout << "Poza blokiem k =" << k << endl ;                          // ❼
}
```

Wykonanie tego fragmentu programu spowoduje pojawienie się na ekranie:

```
Jestem w main, k=33                    ❷
 po lokalnej definicji k =10           ❺
Poza blokiem k =33                     ❼
```

Komentarz

❶ Definicja zmiennej globalnej `k` istniejąca gdzieś w programie, poza jakąkolwiek funkcją (także poza funkcją `main`).

❷ Odwołanie się do obiektu `k`. Jeszcze nie nastąpiła definicja lokalna, więc kompilator uznaje, że chodzi nam o obiekt `k` globalny.

❸ Otwieramy lokalny blok.

❹ Definiujemy obiekt lokalny o nazwie `k`.

❺ Lokalna nazwa `k` zasłoniła nazwę `k` globalną. Na ekranie zostaje więc wypisana wartość zmiennej lokalnej.

❻ Zamknięcie lokalnego bloku. Obiekt lokalny `k` przestaje istnieć, a jego nazwa przestaje być ważna. Skończyło się życie obiektu, skończył się także zakres ważności jego nazwy.

❼ Odwołanie się do nazwy `k` jest teraz rozumiane przez kompilator jako odwołanie się do globalnego obiektu `k`.

Mimo wszystko istnieje jednak możliwość odniesienia się do zasłoniętej nazwy globalnej.

Posłuży nam do tego tzw. operator zakresu :: (dwa dwukropki). Oto przykład:

```
#include <iostream.h>

int k = 33 ;          // ❶ zmienna globalna (obiekt typu int)
/*****************************************************/
main()
{
     cout << "Jestem w main , k =" << k << "\n" ;
     {
          int k = 10 ;                    // zmienna lokalna    ❷
          cout << "po lokalnej definicji k ="
               <<   k                                        // ❸
               << "\nale obiekt globalny k ="
               << ::k ;                                      // ❹
     }
     cout << "\nPoza blokiem k =" << k << endl ;
}
```

Wykonanie objawi się na ekranie jako:

```
Jestem w main , k =33
po lokalnej definicji k =10
ale obiekt globalny k =33
Poza blokiem k =33
```

Uwagi

❶ Definicja obiektu globalnego.

❷ Definicja obiektu lokalnego.

❸ Odwołanie się od obiektu lokalnego.

❹ Odwołanie się wewnątrz lokalnego bloku do zasłoniętego obiektu globalnego. Sprawia to zapis z operatorem zakresu `::k`
Ten chwyt możliwy jest tylko w stosunku do zasłoniętego obiektu globalnego:

> Jeśli nazwa lokalna zasłania inną nazwę lokalną, wówczas nie da się do niej dotrzeć takim operatorem zakresu.

3.8 Modyfikator `const`

Czasem chcielibyśmy w programie posłużyć się obiektem (np. typu `int`), którego zawartości nawet przez nieuwagę nie chcielibyśmy zmieniać. Obiekt tego typu, to tak zwany *obiekt stały*.

Mówiliśmy już o stałych dosłownych. Były to po prostu liczby, które napisane były w tekście programu. Tutaj nie chodzi o liczby, ale o obiekty, które mają

w sobie jakąś wartość. Paradoksem byłoby powiedzieć: chodzi o zmienne, które w programie mają się nie zmieniać.

Przykładem może być choćby program na liczenie pola koła, objętości kuli i czegoś jeszcze. Wielokrotnie w takim programie potrzebować będziemy liczby π. W tym celu zdefiniujemy sobie obiekt typu `float` i nadamy mu wartość odpowiadającą liczbieπ.

```
float pi = 3.14 ;
```

Jeśli jednak chcemy mieć pewność, że nigdy, nawet przez nieuwagę nie zmienimy wartości naszej liczby `pi`, wówczas taką definicję poprzedzamy słowem (modyfikatorem) `const`. Mówimy modyfikator, bo modyfikuje on zwykłą definicję tak, że teraz jest to definicja obiektu stałego.

```
const float pi = 3.14 ;
```

Zauważmy, że równocześnie inicjalizujemy tutaj nasze `pi` wartością 3.14 - musimy to zrobić właśnie przy definicji. Później – przepadło! Od tej pory już nie można podstawić do obiektu `const` żadnej wartości. (Nawet takiej samej!)

```
const float pi = 3.14 ;

     pi = 200 ;          //   !!!  (błąd)
     pi = 3.14 ;         //   !!!  (błąd)
```

Wszelkie próby przypisania jakiejkolwiek wartości do obiektu `pi` będą uznawane za błąd. Tutaj po raz pierwszy pojawia się nam różnica między inicjalizacją a przypisaniem.

Inicjalizacją nazywać będziemy nadanie obiektowi wartości w momencie jego narodzin.

Przypisaniem nazywać będziemy podstawienie do niego wartości w jakimkolwiek późniejszym momencie.

Oto przykłady inicjalizacji:

```
int a = 7 ;
const int cztery = 4 ;
```

Oto przykłady przypisania:

```
a = 100 ;
x = 25.5 ;
r = 30 * 7.5 ;
cztery = 4 ;              // BŁĄD - jeśli był to obiekt const !
```

Zapamiętaj

Obiekty `const` można inicjalizować, ale nie można do nich nic przypisać.

Słowa, słowa, słowa

Osobiście mam wstręt do takich „mądrych" słów jak: modyfikator, bo gdy kilka takich słów spotka się obok siebie w jednym zdaniu – trudno je zrozumieć. Dlatego słowa takie jak `const` nazywam sobie po prostu: przydomek. Nasz obiekt `pi` ma przydomek `const` - jest więc obiektem stałym.

Jeszcze jedna uwaga językowa: Mówimy „inicjaLIZAcja", a nie „inicjacja". Jest ogromna różnica między tymi słowami. Jeśli będziesz uparcie mówił „inicjacja", to zajrzyj sobie kiedyś do encyklopedii i sprawdź co to słowo znaczy. Trochę się pośmiejesz, a potem już zawsze będziesz mówił tylko: „inicjalizacja"

3.8.1 Pojedynek: `const` contra `#define`

Jest to paragraf dla programistów klasycznego C. Jeśli nie programowałeś w języku C, to opuść ten paragraf i przejdź do następnego.

Jeżeli programowałeś w języku C, to zapewne pamiętasz, że w klasycznym C stałe najczęściej definiowaliśmy sobie za pomocą dyrektywy preprocesora. Np.

```
#define PI 3.14
```

Ta forma w C++ jest także dopuszczalna. Pokażemy jednak dlaczego jest gorsza.

Działanie dyrektywy `#define` jest mniej więcej takie, jakbyśmy - bezpośrednio po zakończeniu pisania programu - wydali edytorowi polecenie zastąpienia każdego słowa "`PI`" słowem "3.14". Natychmiast po tym przystępujemy do kompilacji.

Oto co straciliśmy:

- Nazwa `PI` jest kompilatorowi zupełnie nieznana. Nigdy się nawet nie domyśli, że w ogóle istniała. W programie jest tylko kilkakrotnie użyta liczba 3.14 Kompilator nie skojarzy, że chodzi o tę samą liczbę, chociaż człowiek od razu by się tu domyślił. Nie zawsze jednak sprawa jest tak oczywista.

```
#define LICZBA_SILNIKOW  4
```

Kompilator nie zgadnie, które z występujących w programie liczb 4 są określeniem liczby silników, a które są liczbą pór roku, liczbą nóg konia itd.

- W związku z tym, że nazwa `PI` jest kompilatorowi nieznana, dlatego kompilator nie potrafi sprawdzić czy dana nazwa została użyta w ramach jej zakresu ważności. Nie ma tu przecież żadnego zakresu ważności. Są tylko luźno porozrzucane liczby 3.14 (albo liczby 4).

Stała określona przy pomocy dyrektywy procesora `#define` jest znana od linijki wystąpienia dyrektywy `#define` do linijki `#undef` lub – gdy takiej nie ma – do końca pliku. Nie ma to jednak nic wspólnego z zakresem ważności. To tylko jakby obszar, na przestrzeni którego wykonujemy edytorem operacji zamiany znaków `PI` na znaki `3.14`

Czy dużo straciliśmy?

Raczej tak. Straciliśmy możliwość świadomego wyboru zasięgu nazwy. Nazwa może być przecież znana w jednej funkcji, a nieznana w innej. Kompilator nie może teraz nas ostrzec wypadku, gdybyśmy popełnili błąd.

Posłużenie się `#define` w naszym wypadku jest jakby zamianą nazwy na liczbę (stałą dosłowną).

Natomiast zdefiniowanie obiektu jako `const` sprawia, że powstaje nam w pamięci normalny obiekt (np. typu `float`, lub `int`). Dodatkowo obiekt ten ma nalepkę: „Nie Zmieniać Pod Żadnym Pozorem!"

Skoro jest to obiekt, to można poznać jego adres, pokazać na niego wskaźnikiem, itd. Gdybyśmy posłużyli się dyrektywą `#define`, to mielibyśmy w programie do czynienia z kilkoma stałymi dosłownymi. Sama liczba 3.14 nie ma adresu, nie można więc posługiwać się wobec niej wskaźnikiem.

Ostatni z argumentów, który chcę przedstawić, dotyczy posługiwania się programami uruchomieniowymi, czyli z angielska – debuggerami. Program taki pozwala na pracę krokową naszego programu, a w dodatku sprawdzenie, co – w danym momencie – tkwi w jakimś obiekcie naszego programu. Robi się to podając po prostu nazwę danego obiektu. Może Ci się wydać śmieszne pytanie debuggera, co w danym momencie tkwi w stałym obiekcie `PI`, jednak jeśli masz stałą `liczba_silników`, będącą jedną z wielu stałych w tym programie, to często się zdarza, że chciałoby się zapytać co tam właściwie jest.

Rozważmy poniższe dwa warianty. (Użycie małych lub wielkich liter w notacji nazw wynika tylko z tradycji).

```
#define ROZDZIELCZOSC 8192
#define KANALOW_W_BLOKU 128
#define CZYNNIK         (RODZIELLCZOSC / KANAL_W_BLOKU)
#define DLUG_BUF        (CZYNNIK*16*CZYNNIK)
```

W takiej sytuacji może się okazać konieczne upewnienie się ile właściwie wynosi `DLUG_BUF`. Przy tym sposobie definiowania stałej, jest to niemożliwe. Debugger odpowie, że nic mu nie wiadomo o nazwie `DLUG_BUF` (Podobnie jak nic nie wie o nazwach `ROZDZIELCZOSC, KANALOW_W_BLOKU, CZYNNIK)`.

Jeśli jednak zastosujemy sposób obiektami `const`

```
const int rozdzielczosc = 8192 ;
const int kanalow_w_bloku = 128 ;
const int czynnik  = (rozdzielczosc / kanalow_w_bloku) ;
const int dlug_buf = (czynnik * 4 * czynnik) ;
```

to zapytanie debuggera o to, co się kryje pod nazwą `dlug_buf` jest zupełnie legalne i w rezultacie otrzymamy odpowiedź: 16384

3.9 Obiekty `register`

`register` to jeszcze jeden typ przydomka (modyfikatora), który może zostać dodany w definicji obiektu. W tym wypadku można ten przydomek zastosować

do tzw. zmiennej automatycznej typu całkowitego. (Por. paragraf o zmiennych automatycznych – str. 97).

```
register int i ;
```

Dopisując ten przydomek dajemy kompilatorowi do zrozumienia, że bardzo zależy nam na szybkim dostępie do tego obiektu (bo np. zaraz będziemy używać takiego obiektu tysiące razy). Kompilator może uwzględnić naszą sugestię i przechowywać ten obiekt w rejestrze, czyli specjalnej komórce, do której ma bardzo szybki, niemal natychmiastowy dostęp.

Nie ma jednak gwarancji na to, że tak będzie w istocie. Niektóre kompilatory nie są aż tak sprytne. Jak powiedziałem, jest to tylko pobożne życzenie, które kompilator może spełnić lub nie. Większość znanych mi kompilatorów, takie sugestie bierze pod uwagę i program wykonuje się nieco szybciej.

Jeśli deklarujemy zmienną jako `register`, to nie możemy starać się uzyskać jej adresu. Rejestr to nie jest kawałek pamięci, więc nie adresuje się go w zwykły sposób. Jeśli więc mimo wszystko spróbujemy dowiadywać się o ten adres, kompilator umieści ten obiekt w zwykłej pamięci, czyli tam, gdzie może nam określić (i podać) jego adres.

3.10 Modyfikator `volatile`

Jest to przydomek, który mówi kompilatorowi, że ma być ostrożny w kontaktach z danym obiektem.

```
volatile int m ;
```

Volatile †) – znaczy po angielsku: ulotny. Słowo to ostrzega, że obiekt może się w jakiś niezauważalny dla kompilatora sposób zmieniać. Kompilator więc nie powinien upraszczać sobie sprawy, tylko za każdym razem, gdy odwołamy się do tego obiektu – kompilator ma rzeczywiście zwrócić się do przydzielonych temu obiektowi komórek pamięci.

Zapytasz: —A czy kiedykolwiek bywa inaczej?

Tak. Dla naszego dobra tak. Wyjaśnijmy to. Otóż załóżmy, że deklarujemy zmienną określającą słowo stanu jakiegoś zewnętrznego urządzenia. Na przykład urządzenia mierzącego temperaturę oleju w silnikach.

```
int stan_miernika ;
```

To słowo może się zmieniać samo z siebie (bez wiedzy kompilatora), bo przypuśćmy, że do komputera dochodzą przewody z zewnętrznych układów pomiaru temperatury.

Domyślasz się już pewnie, że ten specjalny obiekt został utworzony w tej części pamięci komputera, która reprezentuje układ sprzęgający (interface) komputer

†) (czytaj: „woletail")

z miernikiem temperatury. Załóżmy, że ten układ sprzęgający obsługiwany jest przez taki fragment programu

```
int  a = 5,
     b = 6 ;

volatile int temperatura ;

     cout << "Biezaca temperatura = "
          << temperatura << endl ;                    // ❶
     a = b + 8 ;                                      // ❷
     b = 0 ;                                          // ❸
     cout << "Biezaca temperatura = "
          << temperatura << endl ;                    // ❹
```

Jest ryzyko, że kompilator mógłby „pomyśleć" sobie tak: Pobrałem z komórki `temperatura` jej wartość i wypisałem ją na ekranie ❶.

Potem zajmowałem się jakimiś niezwiązanymi z temperaturą zmiennymi `a` oraz `b`, (❷, ❸) a teraz znowu mam wypisać na ekran zmienną `temperatura`. ❹ Zaraz, zaraz, nie muszę tracić czasu na odczytywanie jej z pamięci, miałem ją przecież gdzieś tu zapisaną na boku (czytaj: w rejestrze). Skoro więc jej od tamtej pory nie zmieniałem, to po prostu wypiszę tę wartość.

Kompilator wypisuje, a tu wybuch! Przez ten czas bowiem prawdziwa wartość obiektu `temperatura` zmieniła się bez wiedzy kompilatora i przekroczyła wartość krytyczną.

Słowo `volatile` przestrzega kompilator przed takim właśnie sprytem. Za każdym razem musi on rzeczywiście sięgnąć do komórki `temperatura`, a nie polegać na tym, co sobie zapisał „na boku".

> Modyfikator `volatile` (czyli: ulotny) oznacza, że obiekt tak określony może się zmieniać w sposób rzeczywiście ulotny, wymykający się czasem spod kontroli kompilatora.

3.11 Instrukcja `typedef`

Instrukcja `typedef` pozwala na nadanie dodatkowej nazwy już istniejącemu typowi.

Przykładowo instrukcja

```
typedef int cena ;
```

sprawia, że możliwa staje się taka deklaracja

```
cena x ;                  // co odpowiada:   int x ;
cena a, b, c ;            // co odpowiada:   int a, b, c ;
```

Po co robić takie sztuczki? Dlaczego nie napisać po prostu `int`?

Otóż taka możliwość jest bardzo przydatna. Wyobraź sobie program, w którym wielokrotnie występują zmienne typu `int`. Niektóre z nich mają określać cenę więc zastosowaliśmy tę instrukcję `typedef`. Pewnego dnia decydujemy, że dokładność liczb całkowitych nas nie zadowala - i chcielibyśmy by ceny repre-

zentowane były typem `float`. Co wtedy robimy? Odszukujemy w programie tę instrukcję `typedef` i zamieniamy ją na taką

```
typedef float cena ;
```

Tym sposobem wszystkie miejsca w programie gdzie używamy nazwy typu – np. deklarujemy obiekty typu `cena` – za jednym zamachem zmieniają się we właściwy sposób. Teraz

```
cena x ;                    // odpowiada:      float x ;
cena a, b, c ;              // odpowiada:      float a, b, c ;
```

Jak widać jest to bardzo wygodne.

Typ, który określamy w instrukcji `typedef`, nie musi być wcale typem fundamentalnym. Równie dobrze może być to typ pochodny. Oto kilka przykładów:

```
typedef int    * wskaznik_do_int ;
typedef char * napis ;

wskaznik_do_int w1 ;        // czyli:      int * w1 ;
napis komunikat ;           // czyli:      char * komunikat ;
```

Poniższa instrukcja definiuje więcej nazw typów

```
typedef int calk, * wskc, natur ;
```

co umożliwia takie konstrukcje:

```
calk a;                     // czyli:      int a;
wskc w ;                    // czyli:      int * w ;
natur n ;                   // czyli:      int n ;
```

Zwróć uwagę jak umieszczona jest gwiazdka wskaźnika.

To zastosowanie instrukcji `typedef` radzę zapamiętać. Jeśli w przyszłości będziesz musiał posługiwać się skomplikowanymi wskaźnikami, to ta instrukcja zapewni Ci, że zapis Twoich programów będzie mimo wszystko czytelny.

> Należy pamiętać, że instrukcja `typedef` nie wprowadza nowego typu, a jedynie synonim do typu już istniejącego.

Instrukcją `typedef` nie możemy redefiniować nazwy, która już istnieje w bieżącym zakresie ważności. Konkretnie: jeśli mamy już deklarowaną nazwę – np. `calk` określającą funkcję wykonującą całkowanie – to nie możemy użyć instrukcji

```
typedef int calk ;
```

bo nazwa `calk` jest już zajęta.

3.12 Typy wyliczeniowe enum

To bardzo ciekawa rzecz. Jest to osobny typ dla liczb całkowitych, a przydaje się on w wielu sytuacjach.

Często zdarza się, że w obiekcie typu całkowitego chcemy przechować nie tyle liczbę, co raczej pewien rodzaj informacji. Oczywiście musimy uczynić to wpisując tam liczbę, ale liczba ta ma dla nas szczególne znaczenie. Wtedy warto skorzystać z typu wyliczeniowego.

Jak definiuje się taki typ wyliczeniowy? Pokażemy to od razu na przykładzie. Chcemy za pomocą liczb określać jakieś działanie układu pomiarowego. Chcemy mieć jakąś zmienną o nazwie `co_robic`. Do niej będziemy wstawiali liczbę określającą żądaną akcję. Oto jak te akcje sobie ponumerujemy:

0	start_pomiaru
1	odczyt_pomiaru
54	zmiana_probki
55	zniszczenie_probki

Typ wyliczeniowy definiuje się według schematu

`enum` *nazwa_typu* `{` *lista wyliczeniowa* `} ;`

Co w naszym wypadku wyglądać może tak:

```
enum akcja {
          start_pomiaru = 0,
          odczyt_pomiaru = 1 ,
          zmiana_probki = 54 ,
          zniszczenie_probki = 55 } ;
```

Zdefiniowaliśmy nowy typ o nazwie `akcja`. A oto definicja zmiennej tego typu:

```
akcja co_robic ;
```

Oznacza to, że `co_robic` jest zmienną, do której można podstawić tylko jedną z określonych na liście wyliczeniowej `akcja` wartości.

To znaczy legalne są takie operacje

```
co_robic = zmiana_probki ;
co_robic = start_pomiaru ;
```

a nielegalne są operacje

```
co_robic = 1 ;
co_robic = 4 ;
```

> Nic, co nie jest na liście wyliczeniowej tego typu wyliczeniowego, nie może zostać podstawione do zmiennej tego typu `akcja`.

Na liście wyliczeniowej zauważamy liczby. Mimo jednak, że odczyt pomiaru jest –jak widzimy – reprezentowany przez liczbę 1, to tej liczby nie mogliśmy wstawić do zmiennej typu `akcja`. Tylko to, co jest na liście.

To bardzo ważna cecha. Dzięki temu nawet przez nieuwagę nie wpiszemy do zmiennej `co_robic` czegoś innego. Nawet gdyby to coś przypadkowo pasowało – jako wartość liczbowa.

Lista wyliczeniowa

Reprezentacja liczbowa elementów listy może być przez nas wybierana wtedy, gdy definiujemy dany typ wyliczeniowy. Nawiasem mówiąc gdybyśmy w naszym przykładzie nie napisali tych liczb 0 , 1 -to takie właśnie liczby byłyby podstawione tam przez domniemanie.

Oto przykład innego typu wyliczeniowego, gdzie reprezentacje liczbowe są inne:

```
enum operacja_dyskowa { czytaj_blok,
                        pisz_blok,
                        przeskocz_blok = 5,
                        przeskocz_znacznik } ;
```

kolejne pozycje na tej liście mają następujące reprezentacje liczbowe

- `czytaj_blok` : 0 – bo **jeśli nie określiliśmy inaczej, to wyliczanie zaczyna się od 0**
- `pisz_blok`:1 – znowu **nie było określenia, więc z wyliczanki wynika, że będzie to liczba następna**, czyli 1
- `przeskocz_blok` : 5 – tu widzimy wyraźne życzenie, by była to liczba 5
- `przeskocz_znacznik` : 6 – znowu nie było życzeń, więc kompilator bierze następną liczbę, czyli 6.

Te reprezentacje liczbowe nie muszą koniecznie się różnić. Mogą być na przykład dwa elementy na liście o nazwie `przewin_tasme` oraz `rozladuj_tasme`, które będą miały tę samą reprezentację. Oczywiście robimy to celowo, gdy chcemy umożliwić nadanie dwu nazw tej samej akcji.

To, jakie są reprezentacje liczbowe – nie musi nas wcale obchodzić. Są to jakieś wartości. W podprogramie, który odpowiada za pracę z dyskiem, żądaną akcję porównujemy nie z liczbami, tylko znowu z elementami tej listy.

Dla wtajemniczonych

Typy wyliczeniowe naprawdę bardzo się przydają. W mojej praktyce chyba najczęściej przy wysyłaniu argumentów do funkcji. Funkcja może spodziewać się argumentu typu wyliczeniowego (np. `operacja_dyskowa`) i kompilator będzie pilnował, by tylko argument tego typu został tam wysłany. Jeśli się pomylimy i do funkcji wyślemy coś innego (np. dowolną liczbę `int` lub inny typ wyliczeniowy np. `kolor`) – wówczas kompilator od razu znajdzie nam ten błąd.

4 Operatory

Zwróciłeś może uwagę, że do tej pory nasze programy były bardzo prymitywne. To między innymi dlatego, że milcząco założyłem, iż wiesz co oznaczają symbole:

```
+    -    *    <    >
```

Tak jednak dalej nie można. O tych i innych operatorach musimy porozmawiać teraz bliżej.

Operatorów jest wiele rodzajów. Ich opanowanie nie wymaga jednak większego wysiłku, bo przeważnie określają one podstawowe operacje arytmetyczne i logiczne, znane nam przecież ze szkoły. Przystąpmy zatem do rzeczy.

4.1 Operatory arytmetyczne

Operatory:

+	dodawania,
–	odejmowania,
*	mnożenia,
/	i dzielenia

nie wymagają chyba żadnych wyjaśnień. Oto przykłady wyrażeń, w których występują te operatory:

```
a = b + c ;          // dodawanie
a = b - c ;          // odejmowanie
a = b * c ;          // mnozenie
a = b / c ;          // dzielenie
```

Operatory te wykonują działania na dwóch obiektach i dlatego nazywamy je *dwuargumentowymi.* Po prostu dodają do siebie dwie liczby, albo dwie zmienne jakiegoś typu. Te mianowicie, które stoją po obu stronach symbolu operatora.

Zatem dodawanie

```
a + 7
```

działa tu na dwóch argumentach:

1) obiekcie a (np. zmiennej),
2) na liczbie 7 (stałej dosłownej)

Praktyczna uwaga:

Zapis Twoich programów będzie dla Ciebie i innych czytelniejszy, gdy przyjmiesz zasadę, by każdy operator po obu stronach miał spacje. Na dowód porównaj dwa identyczne wyrażenia:

```
(a+b+0.32)/c-7.1*(12.4+x)+75.3

(a + b + 0.32) / c - 7.1 * (12.4 + x) + 75.3
```

Kompilatorowi jest tu wszystko jedno. Ty jednak czasem pomyśl też o sobie.

4.1.1 Operator % czyli modulo

Także dwuargumentowym operatorem jest operator dzielenia modulo % (symbol procentu). Jest to inaczej mówiąc operator, dzięki któremu otrzymujemy **resztę z dzielenia.**

Wyrażenie

```
10 % 3
```

ma więc wartość 1, gdyż taka jest reszta z dzielenia 10 przez 3. Na przykład po wykonaniu takiego fragmentu programu:

```
int  x = 8,
     n = 5 ;

     cout << "wynik = " << (x % n) << endl ;
```

na ekranie pojawi się

```
wynik = 3
```

a to dlatego, że 8 dzielone przez 5 daje resztę z dzielenia równą 3.

Operator ten może się przydać często w takich sytuacjach jak poniżej:

```
#include <iostream.h>
main()
{
int i ;
     for(i = 0 ; i < 64 ; i = i + 1)
     {
          if( i % 8)                                    // ❶
          {                                             // ❷
               cout << "\t" ;        // wypis tabulatora
          }
          else
          {                                             // ❸
               cout << "\n" ;        // przejście do nowej linii
```

```
            }
            cout << i ;                                    // ❹
        }
    }
```

W rezultacie wykonania tego programu

na ekranie pojawią się liczby w 8 kolumnach.

```
0   1   2   3   4   5   6   7
8   9   10  11  12  13  14  15
16  17  18  19  20  21  22  23
24  25  26  27  28  29  30  31
32  33  34  35  36  37  38  39
40  41  42  43  44  45  46  47
48  49  50  51  52  53  54  55
56  57  58  59  60  61  62  63
```

Uwagi

❶ To miejsce, gdzie działa nasz operator modulo `%`. W zależności czy wynik jego działania jest zerowy czy niezerowy, podejmowana jest później odpowiednia akcja. Wartość wyrażenia `(i % 8)` jest niezerowa na przykład w sytuacjach:

```
1 % 8     2 % 8     3 % 8     ...  7 % 8
```

natomiast wynik jest zerowy na przykład w sytuacjach:

```
0 % 8     8 % 8     16 % 8    24 % 8
```

❷ Tę akcję podejmuje się, gdy wartość wyrażenia warunkowego jest niezerowa. Jest to wypisanie na ekranie tabulatora.

❸ Tę akcję podejmuje się, gdy wartość wyrażenia warunkowego jest zerowa. Jest to wypisanie na ekranie znaku nowej linii. Czyli po prostu przejście do nowej linii.

❹ To jest miejsce, gdzie odbywa się wypisanie liczby na ekranie.

Priorytet omawianych operatorów jest taki sam, jak do tego przywykliśmy w matematyce. Czyli zapis

```
a + b % c * d - f
```

oznacza to samo co

```
a + ((b % c) * d) - f
```

Innymi słowy mnożenie i dzielenie wykonywane jest przed dodawaniem lub odejmowaniem.

4.1.2 Jednoargumentowe operatory + i -

Plus i minus mogą też wystąpić jako operatory jednoargumentowe. Nic w tym dziwnego, to także znamy ze szkoły.

Oto przykład:

```
+12.7
-x
-(a*b)
```

Jednoargumentowy operator + właściwie nie robi nic, natomiast jednoargumentowy operator - zamienia wartość danego wyrażenia na liczbę przeciwną.

```
int i = 5 ;
     cout << "oto dwa wydruki: " << i << " oraz " << -i ;
     cout << "\nA teraz : " << -(-(-(i+1))) ;
```


W rezultacie wykonania tego fragmentu na ekranie pojawi się

```
oto dwa wydruki: 5 oraz -5
A teraz : -6
```

Zwracam uwagę, że nie ma tu żadnego odejmowania, tylko tworzenie liczby przeciwnej w stosunku do wyrażenia, które zostało poddane tej operacji.

Operatory te są jednoargumentowe, bo działają na tylko jednym argumencie (tym stojącym bezpośrednio po prawej stronie znaku).

4.1.3 Operatory inkrementacji i dekrementacji

Inaczej mówiąc: operatory zwiększenia o jeden i zmniejszenia o jeden.

W pętlach bardzo często wykonywaliśmy działania w rodzaju

```
i = i + 1 ;          // zwiększenie o 1
k = k - 1 ;          // zmniejszenie o 1
```

Zwiększenie o 1 (inkrementacja) lub zmniejszenie o 1 (dekrementacja) jest działaniem tak często spotykanym w programowaniu, że w języku C++ mamy dla wygody specjalne operatory. Oto one:

```
i++  ;          // czyli to samo co: i = i + 1
k--  ;          // czyli to samo co: k = k - 1
```

Dygresja:

> Teraz możesz już rozwikłać zagadkę dlaczego C++ nazywa się właśnie C++. Otóż, gdy w czasach archaicznych język B rozwinął się w język C, żartowano jak będzie się nazywał jego następca. Proroctwa mówiły, że pewnie: Język D. Proroctwa były jak widać chybione, bo następca C nazywa się C++, co należy rozumieć jako „lepsza wersja C"

Operatory inkrementacji ++ i dekrementacji -- są jednoargumentowe. Oba mogą mieć dwie formy:

- przedrostkową (prefix) czyli wtedy, gdy operator stoi z lewej strony argumentu, np: `++a`, `--p` (czyli **przed** argumentem)
- końcówkową (postfix), czyli wtedy, gdy operator stoi po prawej stronie argumentu np.: `a++`, `p--` (po prostu **po** argumencie)

Jest w tym pewna, bardzo sprytna różnica. Rozważmy to na przykładzie operatora inkrementacji (zwiększania)

- Jeśli operator inkrementacji (zwiększania) stoi *przed* zmienną, to najpierw jest ona zwiększana o 1, następnie ta zwiększona wartość staje się wartością wyrażenia.
- W wypadku, gdy operator inkrementacji stoi *za* zmienną, to najpierw brana jest stara wartość tej zmiennej i ona staje się wartością wyrażenia, a następnie - jakby na pożegnanie pracy z obiektem - zwiększany jest on o 1. Zwiększenie to nie wpłynęło więc na wartość samego wyrażenia.

Może brzmi to bardzo zawile, jest jednak bardzo proste. Zobaczmy to na przykładzie:

```
#include <iostream.h>
main()
{
int  a = 5,
     b = 5,
     c = 5,
     d = 5 ;

     cout << "A oto wartosc poszczegolnych wyrazen\n"
          << "(nie mylic ze zmiennymi)\n" ;

     cout << "++a = " << ++a << endl
          << "b++ = " << b++ << endl
          << "--c = " << --c << endl
          << "d-- = " << d-- << endl ;

     // teraz sprawdzamy co jest obecnie w zmiennych

     cout << "Po obliczeniu tych wyrazen, same "
              "zmienne maja wartosci\n"
          << "a = " << a << endl
          << "b = " << b << endl
          << "c = " << c << endl
          << "d = " << d << endl ;
}
```

W rezultacie zobaczymy na ekranie:

```
A oto wartosc poszczegolnych wyrazen
(nie mylic ze zmiennymi)
++a = 6
b++ = 5
--c = 4
d-- = 5
Po obliczeniu tych wyrazen, same zmienne maja wartosci
```

```
a = 6
b = 6
c = 4
d = 4
```

Operator inkrementacji stojący **przed** argumentem nazywa się często operatorem ***pre**inkrementacji,* Natomiast stojący **za** argumentem nazywa się operatorem ***post**inkrementacji.*

Podobnie w wypadku operatorów zmniejszania mówimy o operatorach *predekrementacji* i *postdekrementacji.*

4.1.4 Operator przypisania =

To jest chyba najbardziej oczywiste. Do tej pory wielokrotnie posługiwaliśmy się tym operatorem

```
m = 34.88 ;
```

Powoduje on, że do obiektu stojącego po jego lewej stronie przypisana (podstawiona) zostaje wartość wyrażenia stojącego po prawej. Zatem w zmiennej `m` znajdzie się liczba 34.88

Jest to operator dwuargumentowy, gdyż pracuje na dwóch argumentach stojących po jego obu stronach.

Dobrze wiedzieć i pamiętać, że każde przypisanie samo w sobie jest także wyrażeniem mającym taką wartość, jaka jest przypisywana.

Zatem wartość wyrażenia

```
(x = 2)
```

jako całości jest także 2. Zauważ

```
int a, x = 4 ;
cout << "Wart. wyrazenia przypisania : " << (a = x);
```

W rezultacie na ekranie pojawi się napis:

```
Wart. wyrazenia przypisania : 4
```

Bowiem wyrażenie `(a=x)` nie tylko, że wykonuje podstawienie, ale jeszcze samo ma wartość równą temu, co podstawia.

Może się zdarzyć, że po obu stronach operatora przypisania stać będą argumenty różnego typu. Nastąpi wówczas niejawna zamiana typu wartości przypisywanej na typ zgadzający się typem tego, co stoi po lewej stronie. Przykładowo:

```
int a ;
a = 3.14 ;
cout << a ;
```

nastąpi tu zamiana (mówimy *konwersja*) typu zmiennoprzecinkowego na typ `int`. Nastąpi ona niejawnie, bez ostrzeżeń. Po prostu zostanie wzięta pod uwagę tylko część całkowita liczby 3.14 — czyli 3 — i to też zobaczymy na ekranie.

O konwersjach będziemy jeszcze mówić w osobnym rozdziale (str. 399).

Warto wspomnieć, że istnieją jeszcze inne operatory przypisania specyficzne dla języków C i C++. Można się bez nich obejść, najlepszy dowód, że obchodzi się bez nich matematyka. Z drugiej strony jednak bardzo ułatwiają życie. O tych innych operatorach przypisania będziemy mówić w jednym z dalszych paragrafów.

4.2 Operatory logiczne

Po operatorach arytmetycznych pora na operatory logiczne. Jest ich kilka rodzajów.

4.2.1 Operatory relacji

Operatory

<	mniejszy niż...
<=	mniejszy lub równy...
>	większy niż...
>=	większy lub równy...

są operatorami relacji, w wyniku których otrzymujemy odpowiedź typu: prawda lub fałsz.

Używanie tych operatorów nie przysparza żadnych kłopotów. Oto przykład:

```
if(a > 5)
     {
          cout << " a jednak wieksze od 5! ;
     }
```

Następnymi operatorami tego typu są operatory

==	jest równy...
!=	jest różny od...

Zauważyć należy, iż operator == składa się z dwóch stojących obok siebie znaków '='. (Nie ma tam w środku spacji).

Jest bardzo częstym błędem omyłkowe postawienie tylko jednego – zamiast dwóch znaków ==. Rezultat takiej pomyłki możemy prześledzić na następującym przykładzie

```
int a = 5, b = 100 ;
cout << "Dane sa: a= " << a << " b= " << b ;

                    // tu następuje fatalna linijka
if(a = b)                                          // ❶
     cout << "\nZachodzi równosc \n" ;              // ❷
else
     cout << "\nNie zachodzi równosc \n" ;

cout << "Sprawdzam ze: a= " << a << " b= " << b ;  // ❸
```

Po wykonaniu tego fragmentu programu na ekranie ujrzymy

```
Dane sa: a= 5 b= 100
Zachodzi równosc
Sprawdzam ze: a= 100 b= 100
```

Dlaczego tak się stało ?

Otóż w instrukcji `if` ❶ zamiast porównania (operator ==) zapisaliśmy przypisanie (operator =) . Wiemy już z poprzednich paragrafów, że przypisanie to nie tylko przypisanie, ale w dodatku samo wyrażenie przypisania ma wartość równą wartości będącej przedmiotem przypisania – czyli w naszym wypadku zapis

```
if(a = b)...
```

odpowiada zapisowi

```
if(100)...
```

100 jest różne od zera, czyli przez instrukcję `if` traktowane jest jako wynik „prawda" i tym samym wykonywana jest instrukcja ❷. O tym, że zamiast porównania nastąpiło przypisanie (podstawienie) przekonuje nas rezultat wypisany na ekran instrukcją ❸.

Zniszczyliśmy sobie przez nieuwagę wartość zmiennej `a`. Z drugiej jednak strony nie został popełniony żaden błąd składniowy. Po prostu zamiast działania

```
if(a == b)...
```

wykonaliśmy

```
a = b ;
if(a)...
```

Zatem w naszym przykładzie zrobiliśmy nie to, co chcieliśmy.

Jednak są sytuacje, gdzie chcemy w instrukcji `if` takiego właśnie przypisania, a nie porównania. Chcemy bowiem zyskać na czasie wykonania stosując zamiast dwóch instrukcji jedną.

Jest to składniowo poprawne, jednak niektóre troskliwe kompilatory ostrzegają programistę jeśli przy sprawdzaniu warunku `if` znajdą tam operację przypisania. Tak na wszelki wypadek.

4.2.2 Operatory sumy logicznej | | i iloczynu logicznego &&

Operatory te realizują

||	- sumę logiczną	- czyli operację logiczną LUB (alternatywa)
&&	- iloczyn logiczny	- czyli operację I (koniunkcja)

Na przykład:

```
int k = 2 ;
     if( (k == 10) || (k == 2) )                    // alternatywa
     {
          cout << "Hurra ! k jest równe 2 lub 10 ! " ;
     }
```

co czytamy: jeśli `k` równe jest 10 *lub* `k` równe jest 2 to wtedy...

Przykład na koniunkcję:

```
int m = 22 , k =77 ;

     if( (m > 10) && (k > 0) )                      // koniunkcja
     {
          cout << "Hurra ! m jest wieksze od 10 "
               << "i równoczesnie k jest "
               << "wieksze od zera ! \n" ;
     }
```

Przypominam, że obliczanie wartości wyrażenia logicznego odbywa się w ten sposób, że wynik „prawda" daje rezultat 1, a wynik „fałsz" daje rezultat 0.

Wyrażenia logiczne tego typu obliczane są od lewej do prawej. Dobrze pamiętać, że kompilator oblicza wartość wyrażenia dotąd, dopóki na pewno nie wie jaki będzie wynik. Oznacza to, że w wyrażeniu

```
(a == 0) && (m == 5) && (x > 32)
```

kompilator obliczał będzie od lewej do prawej, a jeśli pierwszy czynnik koniunkcji nie będzie prawdziwy, dalsze obliczanie zostanie przerwane. Nie ma bowiem dalszej potrzeby: – co by tam dalej nie zostało obliczone, i tak koniunkcja nie może być już prawdziwa. Kompilator oszczędza sobie więc pracy.

Wydawać by się mogło, że nie musimy o tym wszystkim wiedzieć. A jednak przydaje się to. Wyobraź sobie, że chciałeś być taki sprytny i upiec parę pieczeni na jednym ogniu:

```
int i = 6 , d = 4 ;

if( (i > 0) && (d++) )
{
     cout << "warunek spelniony !" ;
}
```

Nie dość, że wykonujesz operacje logiczne, to jeszcze chciałbyś, by przy okazji pracy na obiekcie `d` – zwiększyć go o 1.

Otóż jest to pułapka, bo jeśli pierwszy człon koniunkcji nie będzie prawdziwy, to kompilator uzna już, że nie opłaca się zajmować dalszymi. W ten sposób nie dojdzie do wykonania wyrażenia `d++` na co może tak liczyliśmy.

Podobnie jest w wypadku alternatywy. Jeśli w wyrażeniu

```
if( i || (k > 4) )
{
```

```
        // ........
    }
```

pierwszy człon alternatywy (zmienna `i`) jest różny od 0 (czyli „prawda"), to dalsze obliczenia nie są już konieczne. Już w tym momencie wiadomo, że alternatywa ta jest spełniona.

4.2.3 Operator negacji: !

Operator negacji jest operatorem jednoargumentowym. Jego argument stoi po prawej jego stronie. Operator ten ma postać `!` (wykrzyknik)

```
!i
```

Powyższe wyrażenie ma wtedy wartość „prawda", gdy `i` jest równe zero.

Oto przykłady jak używamy takiego operatora:

```
int i = 0 ;
int gotowe ;

if(!i)
     cout << "Uwaga, bo i jest równe zero\n" ;

gotowe = 0 ;
if(! gotowe){
     cout << jeszcze nie gotowe !
}
```

4.3 Operatory bitowe

Mówiliśmy o operatorach arytmetycznych, operatorach logicznych, czas teraz na operatory, które są charakterystyczne dla tego specyficznego sposobu przechowywania informacji w komputerze – jakim jest zapis binarny.

Jak powszechnie wiadomo – w komputerze informacje (liczby i znaki) zakodowane są w poszczególnych komórkach pamięci za pomocą różnych kombinacji zer i jedynek. Te elementarne jednostki informacji nazywane są bitami (bit – ang. kawałek).

Komputer nie odnosi się do każdego z takich bitów osobno. Grupuje je w większe jednostki zwane słowami. Słowo jest jednostką informacji przetwarzaną przez komputer. Zależnie od komputera, różne mogą być rozmiary takich słów. Przykładowo: słowo w komputerze klasy IBM PC/AT składa się z kombinacji szesnastu bitów. Czyli z szesnastu zer lub jedynek.

Komputer przetwarza całe słowa, w których zwykle zapisane są liczby. Nie wszystko jednak w komputerze musi oznaczać liczby. To tak, jak w kokpicie samolotu oprócz informacji liczbowych o wysokości, prędkości, wznoszeniu itd, mamy też informacje logiczne: podwozie schowane lub nie. Oświetlenie samolotu włączone lub nie.

W komputerze informacje takie można zebrać i umieścić razem na poszczególnych bitach jednego słowa pamięci. Do pracy na tak umieszczonej informacji służą nam właśnie operatory bitowe.

Oto lista operatorów bitowych:

<<	przesunięcie w lewo
>>	przesunięcie w prawo
&	bitowy iloczyn logiczny (bitowa koniunkcja)
|	bitowa suma logiczna (bitowa alternatywa)
^	bitowa różnica symetryczna (bitowe exclusive OR)
~	bitowa negacja

W nazwach tych operatorów przewija się słowo „bitowy" , „bitowa". Dlatego w przykładach ilustrujących zastosowanie tych operatorów będziemy pokazywali jak działają one na poszczególne bity. Zakładam, że mamy do czynienia z komputerem, gdzie obiekt typu `int` kodowany jest na 16 bitach (takie są chociażby komputery klasy IBM PC/AT).

Dygresja:
operator << (wypisujący na ekran)
oraz operator >> (wczytujący z klawiatury)

Symbole operatorów przesunięcia w lewo i w prawo przypominają nam poznane wcześniej operatory, którymi posługujemy się do wypisywania na ekran i wczytywania z klawiatury.

```
cout << "podaj liczbe : ";
cin >> x ;
```

Zgadza się, to są rzeczywiście te same symbole. Przez bibliotekę wejścia/wyjścia zostały one tylko wypożyczone. (Nawet takie rzeczy da się robić w C++ !)

Nie ma jednak ryzyka nieporozumień. Operatory przesunięcia są rzeczywiście operatorami przesunięcia w sytuacjach, gdy po obu stronach symbolu << lub >> stoją argumenty typu całkowitego.

Jeśli argumentem z lewej strony jest `cin` lub `cout`, to operatory te oznaczają przesyłanie informacji z klawiatury, lub na ekran. Ta pożyczka nastąpiła dlatego, że wygląd operatorów << i >> dobrze sugeruje akcję, o którą chodzi. W rozdziale o tzw. przeładowaniu operatorów dowiesz się jak łatwo samemu dla swoich własnych celów robić takie pożyczki.

4.3.1 Przesunięcie w lewo <<

Jest to dwuargumentowy operator pracujący na argumentach (operandach) typu całkowitego

zmienna << *ile_miejsc*

Bierze on wzór bitów zapisany w danej zmiennej, przesuwa go o żądaną ilość miejsc w lewo i jako rezultat zwraca ten nowy wzór. Bity z prawego brzegu słowa uzupełnione zostają zerami. Bity z lewego brzegu zostają zgubione.

Przykładowo na skutek takiej instrukcji

```
int a = 0x40f2 ;
int w ;
```

```
w = a << 3 ;
```

następuje przesunięcie o trzy miejsca w lewo. Oto jak wyglądają poszczególne obiekty. Dla łatwiejszej orientacji poszczególne bity słowa zgrupowałem w czwórki.

a 0100 0000 **1111** 0010
w 0000 0**111** 1001 0000

Sam obiekt a nie został zmieniony. Posłużył on tylko jako wartość początkowa. Rezultat został złożony w zmiennej w.

Często chodzi nam o to, by przesunąć bity danej zmiennej, a rezultat ma się znaleźć z powrotem w tej zmiennej. To nic trudnego:

```
a = a << 3 ;
```

(Niebawem poznamy jeszcze lepszy sposób na to: operator <<=).

4.3.2 Przesunięcie w prawo >>

Jest to dwuargumentowy operator pracujący na argumentach (operandach) typu całkowitego

zmienna >> *ile_miejsc*

Bierze on wzór bitów zapisany w danej zmiennej, przesuwa go o żądaną ilość miejsc w prawo i jako rezultat zwraca ten nowy wzór. Bity z prawego brzegu wychodzące poza zakres słowa są gubione.

Jest tu jednak coś, co odróżnia go od opisanego wcześniej kolegi: zachowanie przy uzupełnianiu bitów z lewej strony.

Otóż:

jeśli nasz operator pracuje na danej

- `unsigned` (bez znaku)

lub

- `signed` (ze znakiem), **ale** dana zmienna mieści w sobie akurat liczbę nieujemną

- wówczas bity z lewego brzegu są uzupełniane zerami. To jest gwarantowane.

```
unsigned int d = 0x0ff0 ;
unsigned int r ;

r = d >> 3 ;
```

Oto jak wtedy wygląda rozkład bitów:

d 0000 **1111 1111** 0000
r 0000 0001 **1111 1110**

Jednak jeśli pracuje on na danej typu `signed` (ze znakiem) i jest tam liczba ujemna, to rezultat może zależeć od typu komputera, na którym pracujemy. Może nastąpić uzupełnianie brakujących z lewej strony bitów zerami, a może jedynkami. To – jako się rzekło – zależy już od typu komputera.

```
signed int d = 0xff00 ;
signed int r ;

     r = d >> 3 ;
```

oto jak wtedy wygląda rozkład bitów – pokazujemy tu dwa warianty:

d	**1111 1111** 0000 0000
r	0001 **1111 1110** 0000
r'	**1111 1111 1110** 0000

Kompilatorem Borland C++ na komputerze IBM PC/AT uzyskuje się ten drugi wariant oznaczony tu jako `r'`.

4.3.3 Bitowe operatory sumy, iloczynu, negacji, różnicy symetrycznej

Te dwuargumentowe operatory także działają na argumentach całkowitych. Ich działanie zilustrujemy poniżej

```
int m = 0x0f0f ;
int k = 0x0ff0 ;

int a, b, c, d ;

     a = m & k ;          // iloczyn bitowy
     b = m | k ;          // suma bitowa
     c = ~m ;             // negacja bitów
     d = m ^ k ;          // różnica symetryczna (XOR) bitów
```

A oto jak wyglądają poszczególne obiekty. Dane wejściowe i wartości wyrażeń:

m	0000 **1111** 0000 **1111**	
k	0000 **1111 1111** 0000	
m & k	0000 **1111** 0000 0000	bitowa koniunkcja
m | k	0000 **1111 1111 1111**	bitowa alternatywa
~m	**1111** 0000 **1111** 0000	bitowa negacja
m ^ k	0000 0000 **1111 1111**	bitowe *exclusive or*

W zasadzie sprawa jest jasna i nie wymaga komentarza. Podkreślić należy jednak jaka jest:

4.4 Różnica między operatorami logicznymi, a operatorami bitowymi

Czyli między operatorami

&&	oraz	&
||	oraz	|

Otóż pamiętajmy, że wynikiem działania zwykłego operatora logicznego (np. koniunkcja logiczna) jest wynik „prawda" lub „fałsz", czyli wynikiem jest słowo z zapisaną tam wartością 1 lub 0.

W wypadku operacji logicznych np. `m && k`

kompilatora nie interesuje rozkład bitów w zmiennej `m` czy w zmiennej `k`. Sprawdza on tylko czy jest tam wartość równa zero, czy różna od zera. To samo z drugim argumentem. Wreszcie na tych dwóch wartościach typu „prawda" lub „fałsz" dokonuje koniunkcji. Wynikiem jest 1 lub 0 (czyli „prawda" lub „fałsz").

Natomiast operatory bitowe np. `m & k`

zaglądają do wnętrza danego słowa. Na poszczególnych pojedynczych bitach tych słów dokonują operacji koniunkcji. Czyli biorą pierwszy bit z jednej zmiennej i pierwszy bit z drugiej zmiennej i na tych bitach wykonują operacji koniunkcji. Rezultatem jest 0 lub 1 i ten rezultat wstawiają do pierwszego bitu zmiennej wynikowej. Następnie to samo z drugim bitem i wszystkimi dalszymi. W rezultacie więc otrzymujemy wynik – będący słowem o specyficznym układzie bitów.

Taki wynikowy układ bitów można interpretować jako liczbę, dlatego te bitowe operatory przypominają operatory arytmetyczne.

W naszym ostatnim przykładzie wyrażenie `m & k` można zinterpretować jako liczbę

```
3840
```

Możesz się o tym przekonać wypisując wartość wyrażenia:

```
cout << (m & k) ;
```

4.5 Pozostałe operatory przypisania

Z paragrafem tym czekałem do tej pory mimo, że jest banalnie prosty.

Poznaliśmy już wcześniej operator przypisania (podstawienia) =

W zasadzie może on nam wystarczyć, jednak dla wygody mamy jeszcze do dyspozycji następujące operatory:

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

Ich działanie jest bardzo proste – pokażemy to na przykładzie. Kto zrozumie zasadę w pierwszej linijce, nie musi się trudzić z zapamiętywaniem, bo analogie narzucają się same. I tak:

```
i += 2        oznacza      i = i + 2
i -= 2        oznacza      i = i - 2
i *= 2        oznacza      i = i * 2
i /= 2        oznacza      i = i / 2
i %= 2        oznacza      i = i % 2
i >>= 2       oznacza      i = i >> 2
i <<= 2       oznacza      i = i << 2
i &= 2        oznacza      i = i & 2
```

```
i |= 2        oznacza        i = i | 2
i ^= 2        oznacza        i = i ^ 2
```

Analogia ta nie jest jednak zupełna: jeśli `i` jest wyrażeniem, to w naszym nowym zapisie jest ono obliczane tylko jednokrotnie (w starym - dwa razy). Może to mieć znaczenie, jeśli wyrażenie to ma jakieś działanie uboczne (np. inkrementacja).

Zwracam też uwagę, że te operatory wyglądają tak, iż znak równości następuje jako drugi. Jeśli się pomylisz i napiszesz odwrotnie

```
i =- 2 ;                    // czyli i = -2 ;
```

to zrobisz postawienie liczby -2 do zmiennej `i`.

4.6 Wyrażenie warunkowe

Powtarzam: ***wyrażenie***, a nie instrukcja. Jest to wyrażenie, które obliczane jest czasem na taką wartość, a czasem na inną. Oto jego forma:

(*warunek*) ? *wartość1* : *wartość2*

Przykładowo

```
(i > 4) ? 15 : 10
```

wyrażenie to (jako całość) w zależności od spełnienia lub niespełnienia warunku `(i>4)` przyjmuje różną wartość.

Jeśli warunek jest spełniony — to wartość wyrażenia wynosi 15,
jeśli zaś nie spełniony — wartość tego całego wyrażenia wynosi 10

Jest to bardzo wygodna konstrukcja, bo pozwala zapakować ją do wnętrza innych wyrażeń

np.

```
c = (x > y) ? 5 : 12 ;
```

do zmiennej `c` zostanie podstawiona liczba 5 , jeśli rzeczywiście `x` jest większe od `y`, a liczba 12 jeśli `x` nie jest większe od `y`.

Oto inny prosty przykład:

```
#include <iostream.h>
main()
{
int a ;
cout << "Musisz odpowiedziec TAK lub NIE \n"
     << "jesli TAK, to napisz 1 \n"
     << "jesli NIE to napisz 0 \n"
     << " Rozumiesz ? Odpowiedz : " ;

     cin >> a ;                                  // ❷

     cout << "Odpowiedziales : "
          << ( a? "TAK" : "NIE" )                // ❶
          << " prawda ? " ;
}
```

W wyniku wykonania tego programu na ekranie zobaczymy (Jeśli odpowiedzieliśmy 1)

```
Musisz odpowiedziec TAK lub NIE
jesli TAK, to napisz 1
jesli NIE to napisz 0
 Rozumiesz ? Odpowiedz : 1
Odpowiedziales : TAK prawda ?
```

Komentarz

❶ Tutaj tkwi to wyrażenie warunkowe. W zależności od tego, czy zmienna a jest zerowa czy niezerowa, wyrażenie to jako całość ma wartość

"TAK"	- gdy a jest różne od 0
"NIE"	- gdy a jest równe 0

Zauważ, że w naszym wypadku wartość wyrażenia nie jest tu wcale liczbą, tylko albo takim, albo innym stringiem (jeśli wolisz: stałą tekstową).

❷ Gdybyś tutaj zamiast odpowiedzieć 1 odpowiedział 33, to nie ma problemu – sprawdzany jest przecież warunek czy a jest równe, czy różne od zera. Rezultat będzie taki sam jakbyś odpowiedział: 1.

Może na pierwszy rzut oka omówione wyrażenie warunkowe wydaje się trochę mało czytelne. Jednak zobaczysz, że wkrótce bardzo je polubisz, gdyż zaoszczędzi Ci pisania.

4.7 Operator sizeof

Operator sizeof †) jest to wygodny operator, który pozwala nam rozpoznać zachowania kompilatora i komputera, z którymi przyszło nam pracować.

Jest to ważne z dwóch powodów:

- Te same typy obiektów (np. zmiennych) mogą mieć w różnych implementacjach różne wielkości. Przykładowo: o tym, jaką przestrzeń rezerwuje kompilator na danej maszynie dla obiektu typu int dowiedzieć się możemy używając właśnie tego operatora.
- C++ pozwala użytkownikowi na wymyślanie sobie własnych typów obiektów. (Będziemy o tym mówić w późniejszych rozdziałach). Często ważne jest, by wiedzieć ile pamięci maszyny zajmuje obiekt takiego nowo wymyślonego typu.

Operator ten stosuje się w ten sposób

†) size of [something] = ang. rozmiar [czegoś] (czytaj: „sajz of")

```
sizeof(nazwa_typu)
```

albo

```
sizeof(nazwa_obiektu)
```

w rezultacie otrzymujemy rozmiar obiektu danego typu podany w bajtach. Oto przykład zastosowania:

```
#include <iostream.h>
main()
{
int mm ;

     cout << "Godzina prawdy. W tym komputerze "
          << "poszczególne typy\n"
          << "maja nastepujace rozmiary w bajtach: \n" ;

     cout << "typ char : \t" << sizeof(char)    << endl ;
     cout << "typ int : \t" << sizeof(int)      << endl ;
     cout << "typ short: \t" << sizeof(short)   << endl ;
     cout << "typ long : \t" << sizeof(long)    << endl ;
     cout << "typ float : \t" << sizeof(float)  << endl ;
     cout << "typ double : \t" << sizeof(double) << endl ;
     cout << "typ long double \t: "<< sizeof(long double)
                                                << endl ;

     cout << "Nasz obiekt lokalny mm ma rozmiar : "
          << sizeof(mm) << endl ;
}
```

W rezultacie wykonania tego programu na komputerze klasy IBM PC/AT **na ekranie ujrzymy tekst:**

```
Godzina prawdy. W tym komputerze poszczególne typy
maja nastepujace rozmiary w bajtach:
typ char :     1
typ int :      2
typ short:     2
typ long :     4
typ float :    4
typ double :   8
typ long double     : 10
Nasz obiekt lokalny mm ma rozmiar : 2
```

4.8 Operator rzutowania

Poznamy teraz operator rzutowania, albo inaczej - jawnego przekształcania typu. Jest to operator jednoargumentowy. Działa on w ten sposób, że bierze obiekt jakiegoś typu i jako rezultat zwraca obiekt innego typu. Innymi słowy może np. przekształcić typ `float` na `int`, typ `char` na `double` itp.

Operator ten może mieć dwie formy

```
(nazwa_typu)obiekt
```

względnie

```
nazwa_typu(obiekt)
```

O tym, kiedy której formy użyć, porozmawiamy w przyszłości (str. 413), na razie lepiej jako zasadę przyjąć stosowanie formy pierwszej.

Zobaczmy to na przykładzie

```
int a = 0xffff ;
char b ;

     b = (char) a ;
```

Ponieważ obiekt typu `char` nie pomieści całej informacji zawartej w obiekcie typu `int`, dlatego w rezultacie w obiekcie `b` znajdzie się następująca wartość:

```
0xff
```

Takie przypisanie obiektu typu `int` do obiektu typu `char` musi spowodować utratę 8 najbardziej znaczących bitów (czyli bardziej znaczącego bajtu).

Gdybyśmy nie zastosowali operatora rzutowania, to konwersja taka także by nastąpiła. Po co więc tu operator?

Odpowiedź jest prosta: Dobry kompilator widząc, że następuje ryzyko utraty pewnej części informacji powinien nas ostrzec. Ostrzeżenie takie powinno pojawić się w trakcie kompilacji programu.

Jeśli natomiast jawnie zastosujemy w tym miejscu operator rzutowania, to kompilator napotykając go uzna, że widocznie wiemy co robimy. Ostrzeżenia wtedy nie będzie.

Tak naprawdę, to ten operator będzie nam służył do bardziej wyszukanych konwersji. Pomówimy o tym jeszcze w następnych rozdziałach.

Ogólnie mówiąc: **unikajmy nadużywania tego operatora, chyba że naprawdę wiemy, co chcemy zrobić.**

4.9 Operator: przecinek

Jeśli kilka wyrażeń stoi obok siebie oddzielone przecinkiem, to ta całość jest także wyrażeniem, którego wartością jest wartość wyrażenia będącego najbardziej z prawej.

Zatem wartością wyrażenia

```
(2 + 4, a * 4, 3 < 6, 77 + 2)
```

jest 79. Poszczególne wyrażenia obliczane są od lewej do prawej.

4.10 Priorytety operatorów

Na zakończenie spójrzmy na zestawione w tabeli operatory. Do tej pory unikaliśmy rozmów o priorytecie różnych operatorów. Nadmieniłem tylko, że mnożenie ma pierwszeństwo przed dodawaniem.

Zestawiamy więc teraz operatory w tabeli, w której na samej górze są operatory o najwyższym priorytecie. Pod względem priorytetów operatory dzielą się na 17 grup.

Nie przeraź się jeśli w tabeli zobaczysz operatory, których nie rozumiesz. Nie mówiliśmy jeszcze o wszystkich, mimo to taką tabelę dobrze jest oglądać w całości.

Operatory

Priorytet	Symbol	Nazwa	Zastosowanie	Łączność
17	:: ::	określenie zakresu nazwa globalna	nazwa_klasy::składnik ::nazwa_globalna	L P
16	-> [ ] ( ) ( )	wybranie składnika element tablicy wywołanie funkcji nawias w wyrażeniach	wskaźnik->składnik wskaźnik[wyrażenie] funkcja(lista_argumentów) (wyrażenie)	L
15	sizeof sizeof ++ ++ – – – – ~ ! – + & * new delete delete[ ] ( )	rozmiar obiektu rozmiar typu post inkrementacja pre inkrementacja post dekrementacja pre dekrementacja dopełnienie do 2 negacja jednoargumentowy minus jednoargumentowy plus adres czegoś odniesienie się wskaźnikiem kreuj (rezerwuj) zlikwiduj (anuluj rezerwację) zlikwiduj wektor rzutowanie (konwersja typu)	sizeof(wyrażenie) sizeof(typ) lwartość ++ ++ lwartość lwartość -- -- lwartość ~ wyrażenie ! wyrażenie - wyrażenie + wyrażenie & lwartość * wyrażenie new typ delete wskaźnik delete [ ] wskaźnik (typ)wyrażenie	P
14	.* ->*	Wybór składnika wskaźnikiem: – i nazwą obiektu – i wskaźnikiem do obiektu	obiekt.*wsk_do_składn. wsk ->*wsk_do_składn.	L
13	* / %	mnożenie dzielenie modulo (reszta)	wyraż * wyraż wyraż / wyraż wyraż % wyraż	L
12	+ -	dodaj (plus) odejmij (minus)	wyraż + wyraż wyraż - wyraż	L

11	<< >>	przesunięcie w lewo przesunięcie w prawo	lwartość << wyraż lwartość >> wyraż	L
10	< <= > >=	mniejsze niż mniejsze lub równe większe od większe lub równe	wyraż < wyraż wyraż <= wyraż wyraż > wyraż wyraż >= wyraż	L
9	== !=	równe nie równe	wyraż == wyraż wyraż != wyraż	L
8	&	iloczyn bitowy	wyraż & wyraż	L
7	^	bitowa różnica symetryczna	wyraż ^ wyraż	L
6	|	bitowa suma	wyraż | wyraż	L
5	&&	koniunkcja	wyraż && wyraż	L
4	||	alternatywa	wyraż || wyraż	L
3	? :	arytmetyczne if	wyraż ? wyraż : wyraż	L
2	= *= /= %= += -= <<= >>= &= |= ^=	zwykłe przypisanie mnóż i przypisz dziel i przypisz modulo i przypisz dodaj i przypisz odejmij i przypisz przesuń w lewo i przypisz przesuń w prawo i przypisz koniunkcja i przypisanie alternatywa i przypisanie exclusive or i przypisanie	lwartość = wyraż lwartość *= wyraż lwartość /= wyraż lwartość %= wyraż lwartość += wyraż lwartość -= wyraż lwartość <<= wyraż lwartość >>= wyraż lwartość &= wyraż lwartość |= wyraż lwartość ^= wyraż	P
1	,	przecinek	wyraż , wyraż	L

Chciałbym Cię też uspokoić po raz drugi. Nie musisz wcale uczyć się tych priorytetów. Bez tego można sobie doskonale dać radę posługując się nawiasami.

Zapamiętaj tylko że:

1) Mnożenie, dzielenie, dodawanie, odejmowanie mają takie same priorytety, jak to pamiętamy ze szkoły. Wiedząc o tym, nie będziesz musiał używać nawiasów w tak banalnych sytuacjach.

2) Skomplikowane wyrażenia logiczne lepiej zaopatrywać w nawiasy. Nie tylko dlatego, że && jest mocniejsze niż | |. Także dlatego, że wyrażenia takie stają się czytelniejsze.

Inaczej będziesz produkował zapisy w rodzaju

```
i < b && s || a == n
```

nad którymi zawsze trzeba się zastanowić, a ryzyko popełnienia błędu jest kolosalne.

3) Zapamiętaj też, że nawiasy okrągłe ( ) – oznaczające wywołanie funkcji oraz klamry [ ] – odniesienie się od elementu tablicy – mają bardzo wysoki priorytet, w szczególności wyższy niż tajemniczy jednoargumentowy operator * (czyli odniesienie się do obiektu pokazywanego przez wskaźnik).

4.11 Łączność operatorów

W ostatniej kolumnie tabeli widzimy litery L i P określające lewostronną lub prawostronną łączność operatora. Co to oznacza, pokażemy na przykładach.

I tak operator ! jest prawostronnie łączny - co oznacza, że działa na argumencie stojącym po jego prawej stronie

```
!x
```

W wypadku operatorów dwuargumentowych łączność określa w jaki sposób grupowane jest wykonywanie wyrażenia.

Lewostronna łączność operatora + oznacza, że wyrażenie

```
a + b + c + d + e
```

odpowiada wyrażeniu

```
((((a + b) + c) + d) + e)
```

Prawostronna łączność operatora dwuargumentowego = oznacza, że wyrażeniu

```
a = b = c = d = e
```

odpowiada wyrażenie

```
(a = (b = (c = (d = e))))
```

5 Funkcje

Jedną z najsympatyczniejszych cech nowoczesnych języków programowania jest to, że można w nich posługiwać się podprogramami. Podprogram to jakby mały program we właściwym programie. Dzięki podprogramom możemy jakby definiować swoje własne „instrukcje" :
Jeśli napiszemy sobie podprogram realizujący operację liczenia pola koła na podstawie zadanego promienia – to tak, jakbyśmy język programowania wyposażyli w nową instrukcję umiejącą właśnie to obliczać. Od tej pory – ile razy w programie potrzebujemy obliczyć pole koła – wywołujemy nasz podprogram, a jest to tak proste, jak napisanie zwykłej instrukcji.

Podprogram, który jako rezultat zwraca jakąś wartość, nazywamy po prostu funkcją. W języku C++ wszystkie podprogramy nazywane są funkcjami. Funkcję wywołuje się przez podanie jej nazwy i umieszczonych w nawiasie argumentów.

Oto przykład programu zawierającego funkcję:

Nazywa się ona kukułka, a jej zadaniem jest kukać żądaną ilość razy. To, ile razy – jest parametrem wysyłanym do funkcji.

```
#include <iostream.h>
int kukulka(int ile);                                   // ❶
/*************************************************/
main()
{
int m = 20 ;
     cout << "Zaczynamy" << endl ;

     m = kukulka(5) ;                                   // ❷
     cout << "\nNa koniec m = " << m ;                  // ❸
}
/*************************************************/
int kukulka(int ile)                                    // ❹
{                                                       // ❺
```

```
int i ;
     for(i = 0 ; i < ile ; i++)
     {
          cout << "Ku-ku ! " ;
     }
     return 77 ;                                  // ❻
}                                                 // ❼
```

Wykonanie tego programu objawi się na ekranie jako:

```
Zaczynamy
Ku-ku ! Ku-ku ! Ku-ku ! Ku-ku ! Ku-ku !
Na koniec m = 77
```

Przyjrzyjmy się temu programowi

❶ Funkcja ma swoją nazwę, która ją identyfikuje. Jak pamiętamy z poprzednich rozdziałów, wszelkie nazwy - przed pierwszym odwołaniem się do nich - muszą zostać zadeklarowane. Wymagana jest więc także deklaracja nazwy funkcji. W tym miejscu programu widzimy deklarację nazwy funkcji. Deklaracja ta mówi kompilatorowi:
`kukulka` jest funkcją wywoływaną z argumentem typu `int`, a zwracającą jako rezultat wartość typu `int`.

Powtórzmy:

> Przed odwołaniem się do nazwy wymagana jest jej deklaracja. Deklaracja, ale niekoniecznie od razu definicja. Sama funkcja może być zdefiniowana później, nawet w zupełnie innym pliku. Zdefiniować funkcję, to znaczy po prostu napisać jej treść.

Definicja funkcji znajduje się w ❹

❷ Wywołanie funkcji to po prostu napisanie jej nazwy łącznie z nawiasem, gdzie znajduje się argument przesyłany do funkcji. Ponieważ spodziewamy się, że funkcja zwróci jakąś wartość, dlatego widzimy przypisanie tej wartości do zmiennej `m`.

❸ Na dowód, że funkcja rzeczywiście zwróciła jakąś wartość, i że nastąpiło przypisanie tej wartości do obiektu `m` – wypisujemy go w tym miejscu na ekran.

❹ Tu się zaczyna definicja funkcji. Definicja ta zawiera tzw. ciało funkcji, czyli wszystkie instrukcje wykonywane w ramach tej funkcji. Dwie klamry ❺ i ❼ określają ten obszar ciała funkcji, czyli po prostu jej treść.

❻ Jest to moment, gdy funkcja kończy swoją pracę i wraca do miejsca skąd została wywołana [return = ang. powrót].[†)] Obok słowa `return` widzimy wartość, którą zdecydowaliśmy zwrócić jako rezultat wykonania tej funkcji.

Zatrzymajmy się trochę przy deklaracjach funkcji. Oto kilka przykładów:

†) (czytaj: „rytern")

```
float kwadrat(int bok);
void fun(int stopien, char znaczek, int nachylenie);
int  przypadek(void);
char znak_x();
void pin(...);
```

Zauważ, że na końcu każdej z przedstawionych deklaracji jest średnik.
Przeczytajmy teraz te deklaracje.

- `kwadrat` jest funkcją (wywoływaną z jednym argumentem typu `int`), która w rezultacie zwraca wartość typu `float`.
- `fun` jest funkcją (wywoływaną z 3 argumentami typu : `int`, `char`, `int`), która nie zwraca żadnej wartości. Słowo `void` [ang. – próżny] służy tu właśnie do wyraźnego zaznaczenia tego faktu.
  Po takiej deklaracji – jeśli kompilator zobaczy, że przez zapomnienie próbujemy uzyskać z tej funkcji jakąś wartość – ostrzeże nas, sygnalizując błąd.
- `przypadek` to funkcja, która wywoływana jest bez żadnego argumentu, a która zwraca wartość typu `int`.
- `znak_x` to funkcja, która wywoływana jest bez żadnych argumentów, a w rezultacie zwraca wartość typu `char`.
- `pin` jest funkcją, którą wywołuje się z bliżej nieokreślonymi (teraz jeszcze) argumentami, a która nie zwraca żadnej wartości.

Uwaga dla programistów C:

Jest tu zmiana w stosunku do klasycznego C.

Pusty nawias w deklaracji funkcji np. `f() ; oznacza`

- ❖ – w języku C – dowolną liczbę argumentów czyli to samo co `f(...)`
- ❖ – w języku C++ – brak jakichkolwiek argumentów, czyli to samo co `f(void)`

Jeśli masz przyzwyczajenia z klasycznego C, to zapamiętaj, że odtąd

Deklaracja `f();` – oznacza `f(void);`

Nazwy argumentów umieszczone w nawiasach przedstawionych deklaracji są nieistotne dla kompilatora i można je pominąć. Powtarzam: nazwy, a nie typy argumentów.

Dlatego deklarację funkcji

```
void fun(int stopien, char znaczek, int nachylenie);
```

można napisać także jako

```
void fun(int, char, int);
```

To dlatego, że w deklaracji powiadamiamy kompilator o liczbie i typie argumentów. Ich nazwy nie są w tym momencie istotne. (To będzie ważne w definicji).

Masz więc dwa sposoby deklarowania funkcji. Namawiam Cię jednak do stosowania pierwszego sposobu – tego z nazwami argumentów. Przemawiają za nim dwa względy praktyczne:

- jest on czytelniejszy dla programisty, przypomina bowiem lepiej czym zajmuje się funkcja,
- łatwiej taką deklarację napisać. Mimo, że jest dłuższa. Po prostu pracując w edytorze przenosi się we właściwe miejsce (u nas – na górę programu) pierwszą linijkę definicji funkcji i stawia się na końcu tej linijki średnik.

Zauważ, że w naszym ostatnim programie definicję funkcji oddzieliłem za pomocą linijki komentarza składającą się z rzędu gwiazdek. Radzę Ci robić podobnie. Bardzo to polepszy czytelność Twoich programów. Jeśli masz plik, w którym mieści się trzydzieści definicji funkcji, to łatwiej się w nim orientować. Już na pierwszy rzut oka widać, gdzie się jedna funkcja kończy, a zaczyna druga.

5.1 Zwracanie rezultatu przez funkcję

W naszym przykładowym programie funkcja wywoływana była z argumentem i zwracała jakąś wartość. Przyjrzymy się teraz bliżej temu mechanizmowi przekazywania.

Oto przykład programu liczącego potęgi danej liczby:

```
#include <iostream.h>
long potega(int stopien, long liczba) ;
/*****************************************************/
main()
{
int pocz, koniec ;

     cout << "Program na obliczanie poteg liczb"
          << "calkowitych\n"
          << "z zadanego przedzialu \n"
          << "Podaj poczatek przedzialu : ";
     cin >> pocz ;

     cout << "\nPodaj koniec przedzialu : " ;
     cin >> koniec ;

     // pętla drukujaca wyniki z danego przedziału
     for(int i = pocz ; i <= koniec ; i++)
     {
          cout << i
               << " do kwadratu = "
               << potega(2, i)                    // wywołanie funkcji
               << " a do szescianu = "
```

```
                << potega(3, i)                    // wywołanie funkcji
                << endl ;
     }
}
/*****************************************************/
long potega(int stopien, long liczba)
{
long wynik = liczba ;
     for(int i = 1 ; i < stopien ; i++)
     {
          wynik = wynik * liczba ;
                    // zwięźlej można zapisać to samo jako :
                    //          wynik *= liczba ;
     }
     return wynik ;                                   // ❶
}
/*****************************************************/
```

Jeśli na pytania programu odpowiemy np. 10 oraz 14 to

na ekranie pojawi się:

```
Program na obliczanie poteg liczb calkowitych
z zadanego przedzialu
Podaj poczatek przedzialu : 10
Podaj koniec przedzialu : 14
10 do kwadratu = 100 a do szescianu = 1000
11 do kwadratu = 121 a do szescianu = 1331
12 do kwadratu = 144 a do szescianu = 1728
13 do kwadratu = 169 a do szescianu = 2197
14 do kwadratu = 196 a do szescianu = 2744
```

❶ – Najpierw zwróćmy uwagę, jak odbywa się zwracanie wartości funkcji. Wspominaliśmy już, że robimy to przez instrukcję `return`. Stawia się po prostu przy niej żądaną wartość. U nas to wygląda w ten sposób:

```
return wynik ;
```

mogą być też inne warianty:

```
return (wynik + 6) ;
return 12.33 ;
```

Jeśli stoi tam wyrażenie (np. `wynik + 6`), to najpierw obliczana jest jego wartość, a następnie dopiero rezultat jest „przedmiotem" zwrotu.

Jest jeszcze coś zaskakującego. Otóż jeśli deklaracja funkcji jest taka:

```
long potega(int stopien, long liczba) ;
```

to znaczy, że funkcja ma zwracać jako rezultat wartość typu `long` (pamiętamy, że jest to jeden z typów liczb całkowitych). Tymczasem koło słowa `return` stoi liczba zmiennoprzecinkowa 12.33

Co wtedy? Czy jest to błąd?

Nie zawsze. Nastąpi bowiem próba niejawnej zamiany (konwersji) typu. W naszym wypadku będzie to konwersja typu zmiennoprzecinkowego na typ `long`. Kompilator domyśli się jak to zrobić i w rezultacie funkcja zwróci wartość 12.

Nie zawsze jednak taka konwersja może się odbyć.
Jak bowiem zamienić tzw. wskaźnik na liczbę zmiennoprzecinkową? Kompilator wtedy nam nie podaruje i w trakcie kompilacji oznajmi błąd.

Zapytasz pewnie: „A właściwie co to znaczy, że funkcja zwraca jakąś wartość? Wiemy już jak to się robi, ale co to znaczy?!"

To bardzo ważne pytanie.

Odpowiedź jest bardzo prosta, ale dobrze ją sobie wyraźnie uświadomić. Znaczy to, że wyrażenie będące wywołaniem tej funkcji ma samo w sobie jakąś wartość. W naszym wypadku wyrażenie

```
( potega(2,2) )
```

ma samo w sobie wartość 4. Niezależnie od tego, że jest to wywołanie funkcji. Skoro więc takie wywołanie jest wyrażeniem mającym jakąś wartość, to można go użyć w innych, większych wyrażeniach. Przykładowo

```
7 + 1.5 + potega(2,2) + 100
```

odpowiada u nas wyrażeniu

```
7 + 1.5 + 4 + 100
```

Jeśli funkcja jest zadeklarowana jako zwracająca typ `void` – czyli nie zwracająca niczego – a my, przez nieuwagę, użyjemy ją w takim wyrażeniu, to kompilator ostrzeże nas, że popełniamy błąd. To jedna z wielu zalet obowiązkowych deklaracji funkcji.

Także, gdybyśmy wewnątrz definicji takiej funkcji obok słowa `return` postawili coś oprócz średnika

```
return 6 ;          // błąd, gdy f-cja zwraca void
```

to kompilator wykryje błąd. Mieliśmy bowiem nic nie zwracać, a zwracamy ?

Także odwrotnie: jeśli zadeklarowaliśmy, że funkcja ma coś zwracać, a przy słowie `return` stoi sam średnik, kompilator uzna to za nasz błąd. Obiecałeś, że coś tu będzie, a nie ma? Pewnie o czymś zapomniałeś !

5.2 Stos

Może słyszałeś o czymś takim w komputerze, co się nazywa stos. Jeśli nie, to wyobraź sobie taki obrazek: Wszystkie swoje książki trzymasz w biblioteczce. Gdy którąś potrzebujesz, to idziesz do biblioteczki wyjmujesz i czytasz, potem wkładasz z powrotem. Wiem, wiem jestem naiwny. Prawda jest taka, że najpotrzebniejsze książki leżą na Twoim biurku w postaci mniejszego lub większego stosu. Zdjęcie książki z takiego stosu jest szybsze niż przechadzka do biblioteczki. Tak samo postępuje komputer. Ma także swój podręczny stos, na którym trzyma pewne dane. Koniec obrazka.

Jeśli w obrębie funkcji definiujemy jakieś zmienne, to są one przechowywane najczęściej właśnie na stosie – czyli w tej podręcznej pamięci. Stos ma wiele ciekawych własności – znają je przede wszystkim Ci, którzy programują w asemblerze. Z niektórymi własnościami stosu zapoznamy się w następnych paragrafach.

5.3 Przesyłanie argumentów do funkcji przez wartość

Zajmijmy się teraz sposobem przesyłania argumentów do funkcji. Najpierw sprawa nazewnictwa. Załóżmy, że mamy funkcję

```
void alarm(int stopien, int wyjscie)
{
     cout << "Alarm " << stopien
          << "stopnia"
          << " skierowac sie do wyjscia nr "
          << wyjscie << endl ;
}
```

Załóżmy też, że w programie wywołujemy tę funkcję tak

```
int a, m ;
     // ...
     alarm(1, 10) ;
     alarm(a, m) ;
```

Nazewnictwo jest takie: nazwy

```
stopien , wyjscie
```

- które widzimy w pierwszej linijce definicji funkcji są to tzw. *argumenty formalne* funkcji. Czasem zwane parametrami formalnymi. Ważne jest tu słowo: formalne.

To natomiast, co pojawia się w nawiasie w momencie wywoływania tej funkcji – czyli w naszym wypadku

```
1, 10, a, m
```

to tak zwane *argumenty (parametry) aktualne.*

Czyli takie argumenty, z którymi aktualnie funkcja ma wykonać pracę. Często będę na to mówił prościej: *argumenty wywołania funkcji* – bo z tymi argumentami funkcję wywołujemy.

Dla oswojenia się podajmy obrazek z życia: sklep to jakby funkcja.

```
void sklep(int klient);
```

W sklepie obsługuje się klientów. W sklepie mówią o nas – „muszę obsłużyć klienta". Klient jest argumentem formalnym funkcji sklep.

Jednak do sklepu przychodzą jacyś konkretni ludzie. Gdy do sklepu wchodzi Claudia, to ona jest argumentem aktualnym tego sklepu. To dla niej w tym momencie pracuje sklep. W sklepie nikt nie nazywa jej inaczej jak tylko (bardzo) *formalnie:* klient(-ka). Nawet nie znają jej nazwiska, jednak *aktualnie* klientką jest

Claudia. Gdy Claudia wyjdzie ze sklepu, a za chwilę wejdzie Sybilla, to ona staje się parametrem (argumentem) aktualnym tego sklepu. Sklep na nią i tak znowu mówi „klientka", ale nikt nie twierdzi, że to ta sama osoba.

Taka jest więc różnica między argumentami aktualnymi a formalnymi.

> Argumenty formalne to jest to, jak na parametry mówi sobie w środku funkcja,

natomiast

> argumenty aktualne to to, co aktualnie stosujemy w konkretnym wywołaniu funkcji.

Argumenty przesłane do funkcji – tak, jak to w naszym przykładzie - są tylko kopiami. Jakiekolwiek działanie na nich nie dotyczy oryginału.

Oto dowód:

```
void zwieksz(int formalny)
{
     formalny += 1000 ;            // zwiekszenie liczby o 1000  ❶
     cout << "W funkcji modyfikuje arg formalny\n\t"
          << " i teraz arg formalny = "<< formalny <<endl;

}
```

Jak widać, w tej funkcji zwiększa się wartość argumentu formalnego funkcji. Funkcję tę wywołujemy na przykład w takim fragmencie programu.

```
int aktu = 2 ;
     cout << "Przed wywolaniem, aktu = " << aktu << endl;
     zwieksz(aktu) ;
     cout << "Po wywolaniu, aktu = " << aktu << endl;
```

Jeśli wykonamy taki fragment programu to otrzymamy:

```
Przed wywolaniem, aktu = 2
W funkcji modyfikuje arg formalny
     i teraz arg formalny = 1002
Po wywolaniu, aktu = 2
```

Należy uświadomić sobie bardzo ważną rzecz: Do funkcji przesyłamy tylko wartość liczbową zmiennej `aktu` (parametru aktualnego). Wartość ta służy do inicjalizacji parametru formalnego, czyli zmiennej lokalnej tworzonej przez funkcję na stosie. Jest to więc jakby zrobienie kopii w obrębie funkcji.

Funkcja pracuje na tej kopii. Czyli w naszym przykładzie dodanie 1000 (w miejscu ❶) nie nastąpiło do komórki pamięci, gdzie tkwi `aktu`, ale do tej zmiennej lokalnej na stosie, gdzie mieści się kopia (o nazwie `formalny`). Po opuszczeniu funkcji ten fragment stosu jest niszczony, znika więc też kopia, nie ma śladu, że coś tam robiliśmy.

Bardzo pouczająca przypowieść o babci

Jeśli jeszcze to nie jest dla Ciebie, Czytelniku, jasne to rozważmy taki obrazek. My, (program) robimy teściowej (zmienna) zdjęcie (kopia). Zdjęcie to dajemy dziecku (funkcja) by się pobawiło. Dziecko ukochanej babci dorysowuje wąsy (dodanie 1000). Kiedy skończy zabawę zabawki są sprzątane, a śmieci wyrzucane (niszczenie kopii – zdjęcia). Po skończonej zabawie patrzymy na naszą (żywą) teściową: teściowa wąsów nie ma.

5.4 Przesyłanie argumentów przez referencję

Powyżej opisany sposób znany był programistom C. Język C++ przynosi jeszcze inny sposób przesyłania argumentów. Przez referencję. Czyli przez przezwisko.

W dalszej części książki powinniśmy mówić: „przez referencję", jednak dopóki się z tym nie oswoisz – używać będziemy też równolegle terminu: „przez przezwisko"

Oto przykład:

```
#include <iostream.h>
void zer( int wart, int &ref);                          // ❶
/*****************************************************/
main()
{
int a = 44,
     b = 77 ;

     cout << "Przed wywolaniem funkcji: zer \n" ;
     cout << "a = " << a << ", b = " << b << endl ;

     zer(a, b) ;                                        // ❷

     cout << "Po powrocie z funkcji: zer \n" ;
     cout << "a = " << a << ", b = " << b << endl ;  // ❼
}
/*****************************************************/
void zer( int wart, int &ref)
{
     cout << "\tW funkcji zer przed zerowaniem \n"   ;
     cout << "\twart = " << wart << ", ref = "
          << ref << endl ;                              // ❸

     wart = 0 ;
     ref = 0 ;                                          // ❹

     cout << "\tW funkcji zer po zerowaniu \n" ;
     cout << "\twart = " << wart << ", ref = "
          << ref << endl ;                              // ❺

}                                                       // ❻
```

W rezultacie działania tego programu na ekranie pojawi się

```
Przed wywolaniem funkcji: zer
a = 44, b = 77
     W funkcji zer przed zerowaniem
     wart = 44, ref = 77
     W funkcji zer po zerowaniu
     wart = 0, ref = 0
Po powrocie z funkcji: zer
a = 44, b = 0
```

Komentarz

Patrząc na ekran zauważamy, że funkcja, która chciała wyzerować dwa obiekty `a` i `b` wysłane do niej jako argumenty – zaskoczyła nas. Oczywiście obiekt `a` jest nietknięty. To znamy. Jednak obiekt `b` ma wartość 0. Dlaczego?

Jeśli chodzi o zmienną o nazwie `a` – to nic nas tu nie dziwi. Funkcja odebrała ją przez wartość.

❶ Rzućmy więc okiem na deklarację funkcji `zer`. Widzimy, że to funkcja, która przyjmuje dwa argumenty. Pierwszy z nich jest przesyłany – tak jak poprzednio – przez wartość. Drugi natomiast jest przesyłany przez referencję. Zauważ znak: `&`

❷ W `main` mamy dwie zmienne, które wysyłamy do funkcji `zer`. Inaczej mówiąc: wywołujemy funkcję `zer` z parametrami aktualnymi `a,b`

❸ Wewnątrz funkcji `zer`, na moment przed „egzekucją" wypisujemy jeszcze wartość dwóch parametrów formalnych `wart, ref`

❹ Tu następuje jakaś operacja zmieniająca wartość zmiennych `wart` i `ref`. W naszym wypadku to wpisanie tam zer.

❺ Na dowód, że tak się stało w istocie - wypisujemy ich wartość na ekran.

❻ Kończymy pracę funkcji. Ponieważ funkcja zwraca typ `void` (po prostu nic nie zwraca) dlatego możemy sobie tu oszczędzić instrukcji `return`. Gdybyśmy chcieli by ona koniecznie była, to linijkę wcześniej należałoby napisać

```
return ;
```

❼ Po powrocie z funkcji, będąc już w `main` wypisujemy na ekranie wartości zmiennych `a` i `b`. **I tu jest cała niespodzianka.** Z treści, która pojawia się na ekranie widać, że ten argument, który funkcja przyjmowała starym sposobem (przez wartość) nie został zmodyfikowany. Natomiast ta zmienna, którą funkcja odebrała przez referencję (przezwisko) została zmodyfikowana.

Dlaczego ?

Otóż w tym wypadku do funkcji, zamiast liczby 77 (wartość zmiennej `b`), został wysłany adres zmiennej `b` w pamięci komputera.

Ten adres funkcja sobie odebrała i (na stosie) stworzyła sobie referencję. Czyli powiedziała sobie coś takiego: „Dobrze, zatem komórce pamięci o przysłanym mi adresie nadaję pseudonim (przezwisko) `ref`.

Podkreślmy jasno: ta sama komórka, na którą w `main` mówiło się `b`, stała się teraz w funkcji `zer` znana pod przezwiskiem `ref`. Są to dwie różne nazwy, ale określają **ten sam** obiekt.

W ❹ do obiektu o tym przezwisku `ref` wpisano zero. Skoro `ref` było przezwiskiem obiektu `b`,to znaczy, że odbyło się to na obiekcie `b`.

Ponieważ, jak pamiętamy, po zakończeniu działania funkcji likwiduje się śmieci, zobaczmy co zostało zlikwidowane.

- 1) Będąca na stosie **kopia** zmiennej `a`. (Która to kopia początkowo miała wartość 44, a potem 0). Pamiętamy, że ten argument odebrany był przez wartość.
- 2) Drugi argument przesyłany był przez referencję, więc na stosie mieliśmy zanotowany adres tego obiektu, który to obiekt wewnątrz programu przezywaliśmy `ref`.Ten adres został zlikwidowany. (Jeśli podrzemy kartkę z zapisanym adresem jakiegoś budynku, to mimo wszystko budynek ten nadal stoi. My co prawda tracimy adres tego budynku, ale inni – np. funkcja `main` – mają ten adres u siebie zanotowany.

Wniosek: Przesłanie argumentów funkcji przez referencję pozwala tej funkcji na modyfikowanie zmiennych (nawet lokalnych!) znajdujących się poza tą funkcją.

Programistów klasycznego C opanowała na pewno teraz euforia:

„–Wreszcie jest łatwy sposób modyfikowania argumentów! To, na co nie pozwalało przesyłanie argumentów przez wartość, staje się wreszcie możliwe dzięki przesyłaniu przez referencję!"

Hola, hola! Chciałbym Cię tutaj przestrzec. Na pewno w innych, nawet prymitywnych językach programowania, spotkałeś już taki właśnie sposób przesłania argumentów do funkcji, mimo że tam nie nazwał się on przesłaniem przez referencję.

Dlaczego zatem tak wspaniały język jak C (klasyczne) nie pozwalał na to? Widocznie były powody. Nie, nie chodzi o to, że może dla piszących kompilatory byłoby to trudne w realizacji. Powody są inne. Otóż przesyłanie przez referencję jest prostą drogą do pisania programów bardzo trudnych do późniejszej analizy.

Nasze zmienne w jakimś fragmencie programu zmieniają się bowiem **w sposób niezauważony** na skutek działania jakiegoś innego fragmentu programu (innej funkcji). Niezauważony, bowiem z wywołania funkcji `zer` w środku `main` ❷ nie widać, który argument przesyłany przez wartość, a który przez referencję. Nie ma więc ostrzeżenia: acha, ten argument może być tam modyfikowamy!

☞ Ten sposób przesyłania argumentów do funkcji powinien być więc zasadniczo unikany. Są jednak sytuacje, kiedy się bardzo przydaje. To właśnie z powodu takich sytuacji ten sposób przesyłania argumentów został do języka C++ wprowadzony. O tych sytuacjach będziemy jednak mówić dokładniej w dalszych rozdziałach. Tutaj tylko wspomnę, że sposób ten stosuje się do tak dużych obiektów, że przesłanie ich przez wartość (wymagające zrobienia kopii np.

dziesiątków, setek bajtów) powodowałoby znaczące spowolnienie wywoływania takiej funkcji. W wypadku, gdy taka funkcja jest wywoływana bardzo wiele razy, może to być czynnikiem ważnym.

Jeszcze innym sposobem przesłania argumentu może być posłużenie się tzw. wskaźnikiem. Ten sposób omówimy bliżej w rozdziale o wskaźnikach.

5.5 Kiedy deklaracja funkcji nie jest konieczna

Jak wspomnieliśmy – każda nazwa przed odniesieniem się do niej (po prostu użyciem jej) musi zostać zadeklarowana. Dotyczy to też nazw funkcji. Funkcje muszą więc być deklarowane i robi się to w sposób, o którym już mówiliśmy.

Jednakże, jak pamiętamy, każda definicja (funkcji) jest także przy okazji jej deklaracją.

Jeżeli więc w pliku definicja funkcji jest wcześniej (po prostu wyżej) niż linijka z jakimkolwiek wywołaniem tejże funkcji – to nie trzeba osobnej deklaracji tej funkcji.

Jeśli natomiast funkcja nie jest osobno deklarowana, a wywołanie następuje w linijce powyżej definicji tej funkcji – wówczas kompilator zaprotestuje komunikatem o błędzie (-bo nie będzie jeszcze znał nazwy tej funkcji z żadnej deklaracji).

Oto ilustracja obu przypadków:

```
/*****************************************************/
void funkcja_gorna(void)                    // ❶ definicja która
                                            // jest też deklaracją
{
     // dla uproszczenia pusta funkcja - czemu nie ?
}
/*****************************************************/
main()
{
     funkcja_gorna();                                   // ❷
     funkcja_dolna();                                   // ❸
}
/*****************************************************/
void funkcja_dolna(void)              // ❹ definicja która też jest
                                      // deklaracją,
                                      // ale niestety spóźnioną !
{
     // dla uproszczenia pusta funkcja - czemu nie ?
}
/*****************************************************/
```

Jeśli spróbujemy skompilować powyższy program otrzymamy komunikat o błędzie kompilacji w linijce ❸. To dlatego, że w tym momencie kompilator nie zapoznał się jeszcze z deklaracją funkcji `funkcja_dolna`, która to deklaracja (łącznie z definicją) znajduje się kilka linijek niżej ❹.

Zupełnie inaczej jest z wywołaniem funkcji `funkcja_gorna`. Kompilując linijkę ❷ kompilator zna już deklarację funkcji `funkcja_gorna`, bo się na nią natknął powyżej, w linijce ❶.

Nie chodzi tu bynajmniej o pozycję funkcji w stosunku do specjalnej funkcji `main`. Nasza funkcja `main` mogłaby być dowolną funkcją wywołującą dwie inne. Błąd byłby ten sam.

Należało zatem pamiętać, że jej definicja następuje po linijce wywołania. To, czy dana funkcja jest *przed-* czy *po-* łatwe jest do opanowania w niewielkich programach. W programach większych nie radzę na tym polegać i po prostu deklarować wcześniej wszystkie funkcje.

Moim zdaniem nie ma sensu próbować oszczędzać na deklaracjach. Łatwo je przecież zrobić (- kopiując w edytorze pierwszą linijkę definicji funkcji i umieszczając ją na górze programu – zaopatrując przy okazji w średnik).

Deklarujmy więc wszystkie funkcje !

Przyjmując taką zasadę uwalniamy się od pamiętania, która funkcja jest powyżej której.

Uwaga dla programistów klasycznego C

Jedną z pierwszych niemiłych rzeczy, która Was spotka, gdy programy w C będziecie przerabiali na C++ będzie właśnie sprawa braku deklaracji funkcji.

W języku C deklaracje takie (zwane tam predefinicjami lub prototypami funkcji) były zalecane, ale nie wymagane. Jeśli ich więc nie zamieszczaliśmy, kompilator robił milczące założenia co do typu argumentów i typu wartości zwracanej przez funkcję.

Kiedy przerabiałem na C++ jeden z moich dużych programów w C, taki na 25 tysięcy linijek, – musiałem zrobić deklaracje wszystkich funkcji. Zabrało mi to trochę czasu, ale bardzo szybko się opłaciło, gdyż już w trakcie przeróbki okazało się, że niektóre wywołania funkcji nie zgadzały się dokładnie z definicjami.

Dzięki temu, że kompilator C++ tak krytycznie patrzy na tekst programu, dużo błędów wykrywanych jest już na etapie kompilacji. Stąd też C++ ma opinię takiego języka, w którym programy – jeśli przebrną przez kompilację – działają od razu, bez żmudnego procesu uruchamiania.

5.6 Argumenty domniemane

Do pochopnego przesyłania argumentów przez referencję nie zachęcałem.

Jest za to w C++ inna nowość w argumentach funkcji (w stosunku do C klasycznego) – do której zachęcam. Są to tak zwane argumenty domniemane.

Weźmy taki przykład:

```
/*****************************************************/
void temperatura(float stopnie, int skala)
{
     cout << "Temperatura komory : " ;
```

```
        switch(skala)
        {
            case 0 :
                cout << stopnie << " C\n" ;
                break ;

            case 1 :
                cout << (stopnie +273) << " K\n" ;
                break ;

            case 2 :
                cout << cel_to_far(stopnie) << " F\n" ;
                break ;
        }
}
```

Funkcja ta, jak łatwo się zorientować, służy do wydruku informacji o temperaturze. Temperatura jest argumentem przysyłanym do funkcji. Temperatura przysyłana jest zawsze w stopniach Celsjusza, jednak na ekran chcemy tę temperaturę wydrukować czasem w stopnich Celsjusza, czasem w stopniach Kelvina, a czasem w stopniach Fahrenheita.

To, w jakiej skali mamy akurat wypisać bieżącą temperaturę, zależy od drugiego argumentu. Jeśli jest on równy 0 – to w stopniach Celsjusza, jeśli 1 – to w Kelvinach, jeśli równy 2 – to w stopniach Fahrenheita.

Do zamiany stopni Celsjusza na Fahrenheita mamy jakąś funkcję

```
float cel_to_far(float stopnie);
```

To, gdzie ona jest i jak jest zrealizowana – jest teraz nieistotne. Dla uproszczenia załóżmy, że jest w bibliotece.

Do tej pory nie było jeszcze nic nowego. Zobaczmy jak z naszej funkcji `temperatura` korzystamy. Oto przykładowe wywołania:

```
// ...
    temperatura(52.5, 0);
    temperatura(20.1, 2);
    temperatura(52.5, 0);
    temperatura(100, 0);
    temperatura(52.5, 0);
    temperatura(60, 2);
```

Jak dotąd także wszystko jest po staremu.

A teraz zastanówmy się: w naszym programie setki razy drukujemy temperaturę w skali Celsjusza, natomiast bardzo rzadko w innych skalach. Jednakże za każdym razem, gdy chcemy wydrukować w skali Celsjusza, musimy jako drugi argument wysyłać to nieszczęsne zero.

A przecież, gdy pytam laboranta o temperaturę wrzenia wody i nie dodaję w jakiej skali to on *domniemywa*, że chodzi o skalę Celsjusza. Czy kompilator nie mógłby się też inteligentnie domyślić? Mógłby. Po to są właśnie argumenty domniemane.

Wystarczy w naszym wypadku deklarację funkcji napisać tak:

```
void temperatura(float stopnie, int skala = 0 );
```

sprawi to, że jeśli w naszym programie wywołamy funkcję np. tak

```
temperatura(66.3);
```

Czyli z jednym (tylko pierwszym) argumentem, kompilator domniema, że drugi argument jest równy 0, tak jak mu to w deklaracji przykazaliśmy.

> Podkreślam: – o tym, że argument jest domniemany, informujemy kompilator raz, w deklaracji funkcji. Jeśli definicja jest później, to w definicji już się tego nie powtarza.

Od tej pory wolno nam tę funkcję wywoływać także z jednym argumentem. Stary sposób z dwoma argumentami też jest dopuszczalny, wtedy kompilator nie musi nic domniemywać, po prostu robi to, co kazaliśmy w wywołaniu funkcji.

Zatem poniższe sposoby wywołania są legalne

```
temperatura(100.3) ;          // domniemywa się: skala = 0
// poniżej nic się nie musi domniemywać , bo jest napisane jasno
temperatura(36.6, 0) ;        // chcesz w Celsjuszach, bo 0
temperatura(50.3, 1);         // chcesz w Kelwinach, bo 1
temperatura(159.8, 2);        // chcesz w Fahrenheitach, bo 2
```

Jeśli chcemy, by funkcja miała kilka argumentów domniemanych, to argumenty takie muszą być na końcu listy

```
int multi(int x, float m, int a = 4,
                  float y = 6.55, int k = 10) ;
```

Ostatnie argumenty (jako domniemane) mogą więc być w niektórych wywołaniach tej funkcji opuszczane.

Oto przykłady wywołań tej funkcji. W komentarzach podaję jakie wartości mają wówczas argumenty `a`,`y`,`k`. Argumentów `x` i `m` nie podaję, bo są to zwykłe, (nie-domniemane) argumenty i sprawa jest jasna: 2 i 3.14

```
multi(2, 3.14) ;                  // a = 4, y = 6.55, k = 10
multi(2, 3.14, 7 );               // a = 7, y = 6.55, k = 10
multi(2, 3.14, 7, 0.3) ;          // a = 7, y = 0.3, k = 10
multi(2, 3.14, 7, 0.3, 5) ;       // a = 7, y = 0.3, k = 5
```

Nie jest możliwe opuszczenie domniemanego argumentu `a` lub `y`, a umieszczenie argumentu `k`. Zatem wywołanie typu

```
multi(2, 3.14, 7, , 5) ;                // !!!
```

jest traktowane jako błąd. Nie trzeba się tu nic uczyć, aby zapamiętać, które kombinacje są dopuszczalne, a które nie. Zasada jest prosta i logiczna:

> W języku C++ nie może wystąpić taka sytuacja, że obok siebie stoją dwa przecinki.

> Taka sytuacja uważana jest jako błąd literowy. Nie dlatego, żeby kompilator nie potrafił sobie dać rady z zapisem. To dlatego, że gdyby kompilator zezwolił nam na zapisy z dwoma przecinkami – to tym samym znieczuliłby się na wszystkie wypadki, gdzie

rzeczywiście zapomnieliśmy o jakimś argumencie, albo po prostu niepotrzebnie napisały nam się dwa przecinki.

Kompilator znowu robi to w naszym interesie.

Na zakończenie jeszcze raz podkreślmy: Argumenty domniemane określa się w deklaracji funkcji. W definicji nie. Nawet jeśli w definicji napisalibyśmy te same wartości domniemane, dobry kompilator nie powinien się na taką powtórkę zgodzić.

Ale :

Mówiliśmy kiedyś, że jeśli funkcja jest w programie napisana powyżej jakiegokolwiek jej wywołania, to oddzielna deklaracja tej funkcji nie jest potrzebna. Sama definicja funkcji jest wtedy także jej pierwszą występującą w tym programie deklaracją.

Gdzie wtedy określić argumenty domniemane? Oczywiście w deklaracji - a że jest nią tutaj akurat definicja funkcji – to robimy to tutaj.

Zasada jest taka:

> Chodzi o to, żeby kompilator o argumentach domniemanych danej funkcji dowiedział się tylko raz – wtedy, gdy dowiaduje się co to za funkcja (czyli wtedy, gdy dociera do niego deklaracja). Kompilator musi to już wiedzieć w momencie, gdy opracowuje wywołania tej funkcji w programie. Nietrudno się domyślić, że informacje te są mu potrzebne wtedy, by te opuszczone przez nas argumenty uzupełnić.

5.7 Nienazwany argument

Wyobraźmy sobie taką sytuację. Mieliśmy funkcję wywoływaną z jednym argumentem. Była to na przykład funkcja

```
void ton(int wysokosc) ;
```

powodująca, że komputer zapiszczy jednym z sygnałów ostrzegawczych. Dajmy na to, że parametr o nazwie `wysokosc` służy do wyboru wysokości tonu. (Definicji tej funkcji nie przytaczam, bo jej wygląd zależy od implementacji).

Funkcja ta do tej pory dobrze nam służyła i w naszym programie setki razy wywoływaliśmy ją w różnych miejscach.

Pewnego dnia jednak zapragnęliśmy ciszy, albo też uznaliśmy, że program, który piszczy za dużo, nie ma wyglądu naukowego, albo... Krótko mówiąc po drastycznym cięciu definicja naszej funkcji wygląda tak:

```
void ton(int wysokosc)
{                                   // funkcja jest teraz pusta
}
```

Nie chodzi tu w tym przykładzie o to, że w funkcji nie ma żadnej instrukcji - to nie jest problem. Wolno nam. Chodzi o to, że teraz argument formalny `wysokosc` nie został ani raz użyty. Nie jest to błąd, ale kompilator będzie nas ciągle od tej pory ostrzegał, sugerując że może czegoś zapomnieliśmy. Co robić?

Odpowiedź wydaje się prosta: wyrzucić ten argument z deklaracji i definicji funkcji.

Łatwo powiedzieć. Co się wtedy stanie z tymi setkami wywołań funkcji `ton` w naszym programie? (a może nawet w jeszcze innych, jeśli funkcję `ton` traktowaliśmy jako biblioteczną).

Innymi słowy od tej pory zamiast jednego ostrzeżenia, będziemy mieli setki komunikatów o błędzie nieznalezienia funkcji `ton` wywoływanej z jednym argumentem typu `int`. Każdy komunikat – od jednego wywołania funkcji `ton` w programie. Tragedia. Miało być lepiej, a jest gorzej.

Jest jednak wyjście: Argument nienazwany.

Otóż zamiast wyrzucać cały argument z definicji funkcji wyrzucamy tylko jego nazwę, a typ argumentu zostaje. Jak poniżej

```
void ton(int)
{
}
```

Jest to znak dla kompilatora, że jest to funkcja wywoływana z jednym argumentem typu `int`, ale tego argumentu w funkcji nie używamy. Niech nam więc oszczędzi ostrzeżeń o tym, że zapomnieliśmy z nim cokolwiek zrobić.

Zwróćmy uwagę, że tego cięcia dokonujemy w definicji, a nie w deklaracji funkcji. Jak bowiem pamiętamy – w samej deklaracji nazwy argumentów formalnych (nie typy, tylko nazwy) mogą istnieć lub nie. Kompilator w deklaracji i tak nazwy te ignoruje (korzysta tylko z typów).

Na koniec zdradzę Ci, drogi czytelniku, że osobiście czasem trudno mi definitywnie wyrzucić nazwę. Nazwa zawsze przypomina mi do czego dany argument miał służyć. Dlatego ujmuję nazwę w znaki komentarza, co dla kompilatora jest równoznaczne, że tam nic nie ma

```
void ton(int /* wysokosc */ )
{
}
```

a ja zachowuję wspomnienia.

5.8 Funkcje `inline` (w linii)

Jest to jedna z cech specyficznych dla C++ i nie występuje w klasycznym C, chociaż tam, aby osiągnąć podobny efekt można było sobie jakoś poradzić.[†)]

Załóżmy, że mamy niewielką funkcję. Niewielką, to znaczy jej definicja jest bardzo krótka, zawiera niewiele instrukcji. Przykładowo:

```
int zao(float liczba)
{
```

†) Stosując tzw. makrodefinicje.

```
        return (liczba + 0.5);
}
```

Jej deklaracja mówi nam: `zao` jest funkcją wywoływaną z jednym argumentem typu `float`, a zwracającą typ `int`.

Ze spojrzenia na ciało tej funkcji – czyli na instrukcje będące jej treścią – widzimy, że jest to funkcja służąca do zaokrąglania liczb rzeczywistych do całkowitych. Do liczby typu `float` dodaje się 0.5, a następnie tę wartość przedstawia się instrukcji `return`. Funkcja ta wie, że ma zwrócić typ `int`, a więc wartość stojąca koło `return` zamieniana jest na typ `int` w najbardziej drastyczny sposób, czyli przez odcięcie części ułamkowej. Zostaje sama wartość całkowita i to właśnie zwraca funkcja.

Jak to się dzieje w przypadkach dla liczb 6.8 i 6.2 pokazuje poniższy zapis:

6.8 + 0.5 = 7.3 ⟶ 7
6.2 + 0.5 = 6.7 ⟶ 6

Załóżmy teraz, że w naszym programie bardzo, bardzo często posługujemy się tą funkcją.

Czytelnicy, którzy mają jakieś doświadczenie z programowaniem w asemblerze, wiedzą, że za wywołanie funkcji trochę się płaci. Musi na poziomie języka maszynowego wystąpić kilka instrukcji, które obsługują to specyficzne przejście w inne miejsce programu. Także po wykonaniu funkcji trzeba trochę posprzątać, więc też jest niewielka praca do zrobienia.

Co prawda w języku C++ koszt wywołania funkcji jest relatywnie niski, jednak jeśli naszą funkcję naprawdę zamierzamy wywoływać tysiące razy, to czas zużyty na wywołanie i powrót może stać się znaczący. Mamy więc wybór: albo posługujemy się tą funkcja (jak w poniższym wyrażeniu)

```
l = zao(m) + zao(n * 16.7) ;                    // ❶
```

albo rezygnujemy z niej i zaokrąglamy „na piechotę" wpisując algorytm zaokrąglania w linię wyrażenia.

```
l = (int)(m + 0.5) + (int)( (n * 16.7) + 0.5); // ❷
```

Ten drugi zapis wykona się szybciej, ale ostatecznie może się nam nie chcieć w setkach linii z podobnymi wyrażeniami wpisywać ten kod. Co robić?

Jest wspaniałe wyjście kompromisowe. Pozwala ono na

- jasność zapisu – jak w wypadku funkcji (pierwsze wyrażenie) ❶
- szybkość wykonania – jak w wypadku wpisania tego algorytmu zaokrąglania w linię. ❷

Tak: właśnie *w linię – czyli po angielsku: in line.*†) Naszą funkcję definiujemy tak:

```
inline int zao(float liczba)
{
```

†) (czytaj: „yn lajn")

```
        return (liczba + 0.5) ;
    }
```

Różnica żadna z wyjątkiem tego słowa `inline` .

Co ono daje? Otóż teraz, ile razy w programie umieścimy wywołanie funkcji `zao`, kompilator umieści dosłownie jej ciało (treść) w linijce, w której to wywołanie nastąpiło. Nie będzie więc żadnych akcji związanych z wywołaniem i powrotem z tej funkcji. W rezultacie taki kod będzie wykonywał się szybciej.

Wytłumaczmy to jeszcze prościej: – słowo `inline` sprawia, od tej pory możemy stosować zapis jak w ❶, a komputer i tak sam zamieni sobie to na ❷.

Czy rozmiar programu przez to zmniejszy się czy zwiększy ?

To zależy. Jeśli funkcja, którą zdefiniowaliśmy jest niewielka – taka jak np. nasza – to jej treść może zająć mniej miejsca niż kod generowany związany z obsługą wywołania funkcji i powrotem z niej. Wtedy program może być nieco mniejszy. Mówię tu o przypadku, gdy funkcję `zao` wywołuje się w niewielu miejscach w programie, ale za to miliony razy. W innych wypadkach objętość programu może się zwiększyć. To nic, gdy chodzi o szybkość i łatwość zapisu.

Jednakże pamiętać należy, iż:

> Funkcje typu `inline` zostały pomyślane dla naprawdę małych krótkich funkcji i tylko wtedy mają sens. Nie należy ich nadużywać.

Funkcja typu `inline` może zostać skompilowana tak, jak zwykła funkcja

- jeśli dany kompilator nie jest na tyle dobry, że potrafi sobie z nimi radzić. Dlatego słowo `inline` rozumieć należy jako sugestię dla kompilatora. Z tej sugestii może on skorzystać lub nie.

Jeśli nie skorzysta, to zrobi z niej funkcję zwykłą – zwaną czasem żartobliwie `outline` – czyli „poza linią". Inaczej mówiąc taką, w której ciało jest gdzieś poza linią, w której się ją wywołuje.

Są jeszcze inne sytuacje, gdy funkcja może być skompilowana jako `outline`: Wtedy mianowicie, gdy kompilujemy ją dla pracy z programem uruchomieniowym tzw. debuggerem. Wówczas kompilator wszystkie nasze funkcje typu `inline` skompiluje jako `outline` dlatego, że tak wygodniej jest pracować debuggerowi. Jeśli jednak potem skompilujemy program jeszcze raz, tak „na czysto", to kompilator zachowa się tak, jak umie najlepiej.

Umiejscowienie definicji funkcji typu `inline`

Do tej pory wielokrotnie podkreślaliśmy, że kompilator musi znać deklarację funkcji w momencie, gdy napotka pierwsze wywołanie tej funkcji. Po to, by sprawdzić poprawność wywołania. To tyle.

Teraz jednak chodzi o sprawę poważniejszą. Jeśli funkcja jest typu `inline`, to kompilator napotykając w jakiejś linii jej wywołanie, musi w tej linii wstawić właściwe instrukcje. Zatem teraz już sama deklaracja nie wystarczy. Definicja ciała (treść) funkcji musi już być w tym momencie kompilatorowi znana. Kompilator powinien już mieć „na boku" przygotowaną treść tej funkcji i to właśnie włączy do danej linii programu.

Wniosek stąd taki, że wobec tego:

> Funkcje typu `inline` muszą być na samej górze tekstu programu albo nawet w pliku nagłówkowym, gdzie znajdują się deklaracje innych, zwykłych funkcji, a który to plik dołączany jest w czasie kompilacji modułów naszego programu.

A oto przykład programu z naszą funkcją typu `inline`:

```
#include <iostream.h>
float                                                          // ❶
     poczatek_x,          // początek układu współrzędnych
     poczatek_y ,
     skala_x = 1 ,        // skale: pozioma i pionowa
     skala_y = 1 ;
/*******************************************************/
inline float wspx(float wspolrzedna)                           // ❷
{
     return( (wspolrzedna - poczatek_x) * skala_x) ;
}
/*******************************************************/
inline float wspy(float wspolrzedna)                           // ❸
{
     return( (wspolrzedna - poczatek_y) * skala_y) ;
}
/*******************************************************/
main()
{
float     x1 = 100,            // przykładowy punkt
          y1 = 100 ;

     cout << "Mamy w punkt o wspolrzednych \n" ;
     cout << " x = " << wspx(x1)                               // ❹
          << " y = " << wspy(y1) << endl ;                     // ❺

     // zmieniamy początek układu współrzędnych
     poczatek_x = 20 ;
     poczatek_y = -500 ;

     cout << "Gdy przesuniemy uklad wspolrzednych tak, \n"
          << "ze poczatek znajdzie sie w punkcie \n"
          << poczatek_x << ", " << poczatek_y
          << "\nto nowe wspolrzedne punktu \n"
          << "w takim ukladzie sa : "
          << " x = " << wspx(x1)                               // ❻
          << " y = " << wspy(y1) << endl ;                     // ❼

     // zagęszczamy skalę na osi poziomej
     skala_x = 0.5 ;
     cout << "Gdy dodatkowo zmienimy skale pozioma tak, "
          << "ze skala_x = " <<   skala_x
          << "\nto ten sam punkt ma teraz wspolrzedne : \n"
          << " x = " << wspx(x1)                               // ❽
          << " y = " << wspy(y1) << endl ;                     // ❾
}
```

W wyniku wykonania tego programu na ekranie pojawi się

```
Mamy w punkt o wspolrzednych
 x = 100 y = 100
Gdy przesuniemy uklad wspolrzednych tak,
ze poczatek znajdzie sie w punkcie
20, -500
to nowe wspolrzedne punktu
w takim ukladzie sa : x = 80 y = 600
Gdy dodatkowo zmienimy skale pozioma tak, ze skala_x=0.5
to ten sam punkt ma teraz wspolrzedne :
 x = 40 y = 600
```

Komentarz

Jeśli pierwsze Twoje wrażenie jest takie, że powyższy program więcej gada niż robi, to masz rację. A co w zasadzie robi? Przelicza współrzędne z jednego układu odniesienia na drugi.

> *Nie jest to takie nic – operacje robi się najczęściej przy posługiwaniu grafiką na ekranie. Ekran ma rozdzielczość np. 600 na 800 punktów, a my chcemy tam narysować coś, co u nas w programie ma rozmiary 100 na 100 i ma to w dodatku zająć cały ekran. Trzeba wówczas przeskalować współrzędne.*

W tym rozdziale istotne jest dla nas to, że takich przeskalowań przy skompilowanym rysunku dokonuje się tysiące razy. Tu właśnie dochodzimy do sytuacji, gdy opłaca się użyć funkcji typu `inline`, dla przyspieszenia działania programu.

Jeśli nie jest dla Ciebie jasne na czym polega to przeskalowanie, to nie zaprzątaj sobie teraz tym głowy. Tutaj ważne jest bowiem, że zdefiniowaliśmy dwie funkcje typu `inline`: funkcję `wspx` i funkcję `wspy` (do przeliczania współrzędnej poziomej oraz pionowej). Te definicje widzisz w ❷ i ❸. Jak widać funkcje te korzystają także z niektórych zmiennych globalnych — zdefiniowanych w ❶.

Oczywiście definicje tych funkcji są – zgodnie z zasadą – powyżej miejsc, gdzie są po raz pierwszy wywoływane.

W rezultacie za każdym razem, gdy w programie wywołujemy skalowanie funkcji – u nas w ❹, ❺, ❻, ❼, ❽, ❾ wówczas odbywa się to w sposób maksymalnie szybki – mechanizmem `inline`.

5.9 Przypomnienie o zakresie ważności nazw deklarowanych wewnątrz funkcji

- Zakres ważności nazw deklarowanych w obrębie funkcji ogranicza się tylko do bloku tej funkcji. Nie można więc spoza funkcji za pomocą danej nazwy próbować dotrzeć do zmiennej będącej w obrębie funkcji.
- Nazwą deklarowaną w obrębie funkcji jest też etykieta. Nie można więc do niej skoczyć instrukcją `goto` spoza tej funkcji.

❖ Skoro etykieta jest lokalna dla funkcji, dlatego w dwóch różnych funkcjach mogą istnieć bezkonfliktowo identyczne etykiety.

5.10 Wybór zakresu ważności nazwy i czasu życia obiektu

Przez sposób, w jaki definiujemy obiekt, można decydować o zakresie ważności jego nazwy i o czasie jego życia.

Poniżej omówimy kilka możliwych sposobów definiowania obiektów.

5.10.1 Obiekty globalne

Obiekt zdeklarowany na zewnątrz wszystkich funkcji ma zasięg globalny. Znaczy to, że jest dostępny wewnątrz wszystkich funkcji znajdujących się w tym pliku. Z jednym zastrzeżeniem: jest znany dopiero od linijki, w której nastąpiła jego deklaracja, w dół, do końca programu.

Oczywiście praktyka jest taka, że deklaracje umieszcza się na samym początku pliku, dzięki czemu obiekt jest dostępny wszystkim funkcjom z tego pliku.

```
#include <iostream.h>

int liczba ;                                                    // ❶
void fff(void) ;
/*****************************************************/
main()
{
int i ;
     liczba = 10 ;                                              // ❷
     i = 4 ;
     cout << "Wartosci: liczba = " << liczba
          << " i = " << i ;
     fff() ;                                                    // ❸
}
/*****************************************************/
void fff(void) {
     int x ;                                                    // ❹
     x = 5 ;
     liczba -- ;                                                // ❺
     // i = 4 ;                  // blad !                     ❻

     cout << " sumka = " << (x + liczba) ;
}
/*****************************************************/
```

❒ **W rezultacie wykonania tego programu na ekranie zobaczymy**

```
Wartosci: liczba = 10 i = 4 sumka = 14
```

Komentarz

❶ Definicja obiektu globalnego o nazwie `liczba`.

❷ Przykład użycia zmiennej globalnej w funkcji `main`.

❸ Wywołanie funkcji `fff`.

❹ W funkcji `fff` możemy zdefiniować obiekt lokalny o nazwie `x` i go używać.

❺ Wewnątrz funkcji `fff` możemy także używać globalnego obiektu `liczba`. Jest przecież globalnie dostępny.

❻ Karygodny błąd. Obiekt `i` nie jest obiektem lokalnym **tej** funkcji. Niepoprawne jest takie odwoływanie się do obiektów lokalnych innych funkcji, bowiem zakres ważności nazwy `i` nie rozciąga się na funkcję `fff`. Tutaj nazwa `i` nie jest znana, więc kompilator zaprotestuje.

5.10.2 Obiekty automatyczne

W naszym przykładzie zmienne lokalne `i` oraz `x` są to tak zwane *zmienne automatyczne*. Definiujemy je, a one – w momencie gdy kończymy blok, w którym powołaliśmy je do życia – automatycznie przestają istnieć. To dlatego, że obiekty automatyczne komputer przechowuje właśnie na stosie.

Jeśli po raz drugi wejdziemy do danego bloku (np. przy powtórnym wywołaniu funkcji `fff`) to zmienne takie zostaną powołane do życia po raz drugi. Nie ma żadnej gwarancji, że znajdą się akurat w tych samych miejscach w pamięci co poprzednio. Opuszczając blok – znowu zostaną zlikwidowane.

Wynikają z tego dwa wnioski:

- ❖ – skoro obiekt ten przestaje istnieć, to nie możemy liczyć na to, że przy ponownym wywołaniu tej funkcji zastaniemy go tam z wartością, którą miał na moment przed unicestwieniem.
  Przy ponownym wywołaniu tejże funkcji obiekt taki zostanie zdefiniowany na nowo, bardzo możliwe, że w zupełnie innym miejscu pamięci (stosu)
- ❖ – skoro obiekt ten przestaje istnieć, to nie ma sensu by funkcja zwracała jego adres.[†)] Adres po opuszczeniu takiej funkcji opisuje komórkę, która już nie należy do dawnego właściciela.

  Sytuację tę można porównać do takiego obrazka. Właśnie się wyprowadzamy z dotychczasowego mieszkania, a na schodach dajemy jeszcze komuś nasz stary adres. Tymczasem pod tym starym adresem już nas nikt, nawet za chwilę, nie znajdzie.

Dokładnie to samo dotyczy zwracania referencji (przezwiska) zmiennej lokalnej.

†) O zwracaniu rezultatu będącego adresem, porozmawiamy w rozdziale o wskaźnikach.

Inna sprawa. Pamiętać należy, że zmienne automatyczne nie są zerowane w chwili definicji (czyli w chwili powoływania do życia). Jeśli ich sami nie zainicjalizowaliśmy jakąś wartością, to w tych zmiennych automatycznych tkwią początkowo śmieci.

Dlatego:

> Ze zmiennych automatycznych nie należy odczytywać wartości - zanim najpierw coś sensownego do nich nie zapiszemy.

Jeśli mimo tych ostrzeżeń coś stamtąd przeczytasz – będą to zupełnie przypadkowe wartości.

Wytłumaczenie: zmienne automatyczne przechowywane są na stosie. Przydziela się im tam wymagany dla danego obiektu obszar – i nic więcej. Nie inicjalizuje się tego obszaru – (wpisując tam np. zero).

Natomiast:

> Zmienne globalne – te są zakładane w normalnym obszarze pamięci †). Ten obszar przed uruchomieniem programu jest zerowany, zatem zmienna globalna, jeśli jej nie inicjalizowaliśmy specjalnie - ma wartość 0.

Z obiektami automatycznymi łączy się słowo kluczowe `auto` stawiane przed definicją obiektu wewnątrz jakiegoś bloku (np. bloku funkcji lub bloku lokalnego). Jest ono jednak rzadko używane, gdyż obiekty tak definiowane są automatyczne przez domniemanie. Zatem jeśli np. w bloku funkcji definiujemy obiekt

```
auto int m ;
```

to jest to równoważne definicji

```
int m ;
```

5.10.3 Obiekty lokalne statyczne

Powiedzieliśmy, że zmienne lokalne dla jakiejś funkcji powoływane są do życia w momencie ich definicji, a gdy kończy się wykonywanie tej funkcji przestają istnieć. Wywołanie ponowne tej funkcji powoduje ponowne utworzenie takiej zmiennej, jej wartość stanowią śmieci, więc takiej zmiennej powinniśmy znowu nadać jakąś wartość początkową.

Czasem taki proceder nas zadawala, czasem jednak chciałoby się, by zmienna lokalna dla danej funkcji nie ginęła bez śladu, tylko przy ponownym wejściu do tej funkcji miała taką wartość, jak przy ostatnim opuszczaniu tejże funkcji.

Załóżmy, że mamy napisać funkcję, która nic nie robi tylko pisze ile razy ją do tej pory wywołaliśmy. Musi mieć więc w środku jakiś licznik o czasie życia

†) czyli: przypominając obrazek o stosie – nie na biurku, ale jakby w biblioteczce

rozciągającym się na cały czas wykonywania programu. Czyli taki czas życia, jaki mają zmienne globalne. Z drugiej jednak strony chcemy, żeby była znana tylko lokalnie przez tę funkcję. Gdybyśmy tego ostatniego warunku nie postawili – wystarczyłoby nam użyć zmiennej globalnej.

Oto ilustracja. Sprawę rozwiązujemy tu na dwa sposoby.

```
#include <iostream.h>
/* ---------- deklaracje funkcji ----------*/
void czerwona(void) ;                                    // ❶
void biala(void) ;
/*-----------------------------------------------------*/
main()
{
     czerwona();                                         // ❷
     czerwona();
     biala();
     czerwona();
     biala();
}
/*****************************************************/
void czerwona(void)
{
static int ktory_raz ;                                   // ❸
     ktory_raz ++ ;
     cout      << "Funkcja czerwona wywolana "<< ktory_raz
               << " raz\n" ;
}
/*****************************************************/
void biala(void)
{
static int ktory_raz = 100 ;                             // ❹
     ktory_raz = ktory_raz + 1 ;                         // ❺
     cout      << "Funkcja biala wywolana "<< ktory_raz
               << " raz\n" ;                             // ❻
}
/*****************************************************/
```

Po wykonaniu programu na ekranie pojawi się:

```
Funkcja czerwona wywolana 1 raz
Funkcja czerwona wywolana 2 raz
Funkcja biala wywolana 101 raz
Funkcja czerwona wywolana 3 raz
Funkcja biala wywolana 102 raz
```

Przyjrzyjmy się poszczególnym punktom programu

❶ Funkcje przed pierwszym swym wywołaniem powinny być już znane kompilatorowi. Dlatego na górze programu umieściliśmy deklaracje tych funkcji. Dzięki temu kompilator wie jaka jest liczba i typ argumentów wysyłanych do funkcji, a także jaki typ jest zwracany jako rezultat wykonania tej funkcji. Od tej pory kompilator może już nas sprawdzać czy się nie pomyliliśmy w linijkach wywołania tych funkcji.

❷ W funkcji `main` widzimy serię wywołań funkcji `czerwona` i `biala`.

❸ W funkcji `czerwona` istnieje definicja zmiennej (obiektu) typu `int`, o nazwie `który_raz`. Zauważ słowo kluczowe `static`.

❹ W funkcji `biala` także istnieje definicja takiego samego obiektu. Obiekt ma taką samą nazwę jak w funkcji `czerwona`.Nie przeszkadza to nam jednak, gdyż są to obiekty lokalne, czyli zakres ważności ich nazw jest lokalny. Każda ma zakres ważności ograniczony do funkcji, w której została zdefiniowana.
Zauważyłeś przydomek `static`. To on sprawia, że mimo iż obiekt jest lokalny – nie przestaje on istnieć w momencie, gdy kończy się jego zakres ważności. Słowo to nazywa się też modyfikatorem, gdyż zwykłą definicję obiektu lokalnego modyfikuje tak, iż obiekt staje się „nieśmiertelny".

❻ Zwracam uwagę raz jeszcze – kończy się zakres ważności nazwy obiektu, ale obiekt nie ginie. Jakby zapada w stan hibernacji. Kiedy ponownie wywołana zostaje funkcja, obiekt budzi się z zimowego snu i ma taką wartość, z jaką go ostatnio zostawiliśmy. Na dowód tego w rezultacie wykonania na nim zwiększenia o jedynkę i wypisaniu wartości na ekran widzimy, że rzeczywiście mamy coś w rodzaju licznika.

❸ Definicja obiektu statycznego w funkcji `czerwona` nie zawiera inicjalizacji, czyli podstawienia wartości początkowej. Obiekty statyczne – jako obiekty przypominające czasem życia obiekty globalne – nie są przechowywane na stosie, lecz w normalnej pamięci. Wynika stąd fakt, że obiekty takie – (w przeciwieństwie do obiektów na stosie) – bezpośrednio po definicji nie zawierają żadnych „śmieci", tylko mają w sobie wartość zerową. Jeśli ta wartość nam odpowiada, to nie musimy jeszcze raz jawnie wstawiać tam tego zera.
W przypadku definicji ❹ ta wartość nam jednak nie odpowiadała, więc wstawiliśmy tam liczbę 100. Dzięki temu nasz licznik zaczął liczyć od wartości początkowej 100

> Czy tego samego efektu nie moglibyśmy uzyskać za pomocą zmiennych globalnych? Tego samego nie, bowiem nie może być dwóch obiektów globalnych o identycznej nazwie. Musielibyśmy więc zdefiniować dwie zmienne globalne o różniących się nazwach np.

```
int ktory_raz_cz ;
int ktory_raz_bi = 100 ;
```

Używanie zbyt wielu zmiennych globalnych nie jest eleganckim rozwiązaniem i zdradza zły styl programowania. Raz, że trzeba uważać, by wymyślić nazwę do tej pory jeszcze nie istniejącą, a dwa, że obiekt globalny jest dostępny dla każdego – co zwiększa ryzyko błędu.

Co by było gdybyśmy z naszego programu usunęli słowa `static` ?

Obie zmienne `ktory_raz` staną się wówczas obiektami automatycznymi. Co to oznacza łatwo zobaczyć na ekranie po wykonaniu takiej wersji programu

```
Funkcja czerwona wywolana 1259 raz
Funkcja czerwona wywolana 1259 raz
Funkcja biala wywolana 101 raz
Funkcja czerwona wywolana 1259 raz
Funkcja biala wywolana 101 raz
```

Obiekty straciły nieśmiertelność. Giną w momencie, gdy kończy się ich zakres ważności, a po powtórnych narodzinach, nie pamiętają niczego czym były do tej pory. Widać to wyraźnie na tekście wypisywanym z funkcji `biala`. Zmienna jest zakładana za każdym razem od nowa i za każdym razem nadawana jest wartość 100. W rezultacie zwiększenia o 1 na ekranie pojawia się za każdym razem liczba 101.

Z obiektem z funkcji `czerwona` jest o wiele gorzej. Także i on staje się automatyczny i jako taki zakładany jest na stosie. Ponieważ jednak nie nadajemy mu żadnej wartości początkowej, więc są w nim wstępnie „śmieci". Te śmieci zwiększamy o 1 i wypisujemy na ekranie. Stąd taka zdumiewająca liczba. (Na Twoim komputerze będzie to na pewno jakaś inna liczba, mimo wszystko jednak, pozostałość po poprzedniej treści stosu).

Podsumowanie:

Jeśli chcemy, by jakaś zmienna (ogólniej obiekt) definiowana w obrębie funkcji nie była po zakończeniu pracy funkcji niszczona i zachowywała swoją wartość do „następnego razu", to musimy ją zdefiniować jako statyczną.

Jak pamiętamy, definicja taka wygląda identycznie jak definicja zmiennej automatycznej, z tą różnicą, że przed definicją stoi słowo `static`

```
static float m = 1.7 ;
```

Taka zmienna nie jest już lokowana na stosie (jak zmienne automatyczne), tylko w tym obszarze pamięci, gdzie zmienne globalne, zatem nie ma w niej „śmieci" tylko jest inicjalizowana zerem. Chyba, że tak jak w naszym wypadku, zażądamy inicjalizacją inną wartością (1.7)

Gdy do funkcji wejdziemy po raz pierwszy – taka właśnie wartość czekała będzie tam na nas. Gdy po zakończeniu funkcji i ewentualnych modyfikacjach tej zmiennej – opuścimy funkcję, obiekt ten stanie się dla nikogo niedostępny. Jeśli jednak ponownie funkcja ta zostanie wywołana, zastaniemy tę statyczną zmienną z taką wartością, z jaką ją ostatni raz zostawiliśmy. (Inicjalizacja wartością 1.7 odbywa się tylko jeden jedyny raz, potem zmienna statyczna pamięta już swoją ostatnią wartość).

Zapamiętaj:

Obiekty globalne

– są wstępnie inicjalizowane zerami

Obiekty lokalne:

- – automatyczne – zakładane są na stosie, a tam nic nie jest wstępnie inicjalizowanie, zatem obiekty te wstępnie zawierają „śmieci",
- – statyczne – ponieważ mają pożyć dłużej, zakładane są w tym obszarze pamięci co obiekty globalne – a więc są wstępnie inicjalizowane zerami.

5.11 Funkcje w programie składającym się z kilku plików

Dopóki nasz program jest nieduży nie ma problemu: całość może się zmieścić w jednym pliku dyskowym. Ten plik kompilujemy i linkujemy jeśli chcemy otrzymać program w wersji gotowej do uruchomienia.

Przychodzi jednak taki moment, że program rozrasta się tak bardzo, że kompilacja trwa długo. Jakakolwiek minimalna poprawka wymaga znowu tej długiej kompilacji. Wtedy musisz podjąć decyzję, że odtąd program nie będzie już w jednym pliku – dzielisz go na dwa lub więcej. Jak to zrobić?

Program napisany w jednym pliku można podzielić na dwa pliki tylko w miejscu między definicjami funkcji. Nie można więc dzielić tak, że w pierwszym pliku A będzie funkcja pierwsza, druga i pół trzeciej, a w drugim pliku B reszta trzeciej, czwarta i piąta. Funkcja trzecia musi być albo cała w pliku A, albo cała w pliku B.

O czym jeszcze musimy pamiętać? :

> Przede wszystkim o tym, że po to, by funkcje z pliku B miały dostęp do jakichkolwiek zmiennych globalnych z pliku A – trzeba w pliku B umieścić deklaracje tych zmiennych.

Deklaracje – a nie definicje. Definicje (czyli rezerwacja na nie miejsca) odbyły się już w pliku A. W pliku B chcemy mieć tylko do nich dostęp, czyli móc się do nich odwoływać. Aby móc się odwołać do jakichś nazw muszą one zostać zdeklarowane. Do tego, jak wiemy, w przypadku obiektów takich jak na przykład zmienne – służą deklaracje wykonywane za pomocą słowa `extern`.

Jeśli więc w pliku A mamy następujące zmienne globalne

```
int n ;                    // to wszystko są definicje
float x ;
char z ;
```

to aby móc z nazw `x`,`n` `z` korzystać w pliku B musimy tam zamieścić deklaracje.

```
extern int n ;             // deklaracje !
extern float x ;
extern char z ;
```

Deklaracje te są potrzebne kompilatorowi wtedy, gdy będzie kompilował plik B. Mówią mu one mniej więcej coś takiego:

- Jeśli w pliku B napotkasz nazwę `n`, to na razie deklaruję, że oznacza ona obiekt typu `int`. Gdzie się ten obiekt znajduje? Nie mówię teraz, bo może nawet na zewnątrz (`extern`) tego pliku. (Ale to nic pewnego).
- Jeśli w pliku B napotkasz nazwę `x`, to wiedz, że oznacza ona obiekt typu `float`.Także nie określam gdzie dokładnie on jest.
- Jeśli w pliku B napotkasz nazwę `z`... dalej Ci czytelniku oszczędzę...

Dopiero na etapie linkowania (łączenia) tych plików ze sobą, kod z pliku B „dowie się" gdzie to w pamięci naprawdę będą obiekty `n`, `x`, `z`.

Jeśli chcemy, żeby z pliku B można było wywołać jakąś funkcję z pliku A – to także musimy umieścić w pliku B jej deklarację. W wypadku deklaracji nazwy funkcji nie potrzeba już słowa `extern` – jest ono przyjmowane jako domniemane.

W sumie więc zanim w pliku B pojawią się jego funkcje, najpierw muszą wystąpić deklaracje zmiennych i funkcji z pliku A. Nie wszystkich - tych, które z pliku B będą używane.

Czasem jest wygodne umieścić te wszystkie deklaracje w osobnym pliku, – tak zwanym pliku nagłówkowym, który po prostu bezpośrednio przed procesem kompilacji jest włączany do pliku B (a może i dalszych).

To automatyczne wstawianie do pliku wykonuje za nas specjalna dyrektywa

```
#include "naglowek.h"
```

Jest to tzw. dyrektywa preprocesora†), która bezpośrednio przed rozpoczęciem pracy kompilatora wstawia do pliku, inny plik o danej nazwie (tutaj plik: `naglowek.h`) znajdujący się w bieżącym katalogu (bieżącym – bo nazwa jest ujęta w cudzysłów). Rozszerzenie .h jest zwyczajowym rozszerzeniem dawanym plikom nagłówkowym (.h to skrót od angielskiego header – nagłówek)

Co prawda o tych sprawach dopiero będziemy mówić, jednak chyba już zdążyłeś się oswoić z linijką

```
#include <iostream.h>
```

która towarzyszy naszym programom od dłuższego czasu, a jest niczym innym jak wstawieniem do naszych programów – pliku z deklaracjami zmiennych i funkcji z biblioteki wejścia/wyjścia.

Oto przykład programu, który podzielony został na dwa pliki:

Wszystkie deklaracje zebrano w osobnym pliku nagłówkowym o nazwie `nagl.h` Plik ten jest włączany do obu plików programu.

Oto treść pliku afryka.C :

```
#include <iostream.h>
#include "nagl.h"
int ile_murzynow = 9 ;
main()
{
     cout << "Poczatek programu\n" ;
     funkcja_francuska();
     funkcja_niemiecka();
     cout << "Koniec programu \n" ;
}
```

†) Będziemy o tym mówili w następnym rozdziale.

```
/*****************************************************/
void funkcja_egipska()
{
     cout << "Jestem w Kairze !-------- \n" ;
     cout << "Na swiecie jest " << ile_murzynow
          << " murzynow, oraz " << ile_europejczykow
          << " europejczykow \n" ;
}

/*****************************************************/
void funkcja_kenijska()

{
     cout << "Jestem w Nairobi ! --------\n" ;
     cout << "Na swiecie jest " << ile_murzynow
          << " murzynow, oraz " << ile_europejczykow
          << " europejczyków \n" ;
}
/*****************************************************/
```

A oto plik europa.c :

```
#include <iostream.h>
#include "nagl.h"
int ile_europejczykow = 8 ;
/*****************************************************/
void funkcja_francuska()
{
     cout << "Jestem w Paryzu ! ********************\n" ;

     cout << "Na swiecie jest "<< ile_murzynow
          << " murzynow, oraz "
          << ile_europejczykow << " europejczykow \n" ;

     funkcja_egipska() ;
}
/*****************************************************/
void funkcja_niemiecka(void)
{
     cout << "Jestem w Berlinie ! *******************\n" ;

     cout << "Na swiecie jest " << ile_murzynow
          << " murzynow, oraz "
          << ile_europejczykow << " europejczykow \n" ;
     funkcja_kenijska();
}
/*****************************************************/
```

A tak wygląda zawartość pliku nagl.h

```
extern int ile_murzynow ;
extern int ile_europejczykow ;

void funkcja_egipska() ;
void funkcja_kenijska() ;
void funkcja_francuska() ;
void funkcja_niemiecka() ;
```

Po skompilowaniu plików `europa.c` i `afryka.c` oraz po połączeniu (zlinkowaniu) ich ze sobą otrzymamy gotowy program. Program ten po uruchomieniu spowoduje, że

na ekranie pojawi się następujący tekst

```
Poczatek programu
Jestem w Paryzu ! ********************
Na swiecie jest 9 murzynow, oraz 8 europejczykow
Jestem w Kairze !--------
Na swiecie jest 9 murzynow, oraz 8 europejczykow
Jestem w Berlinie ! ******************
Na swiecie jest 9 murzynow, oraz 8 europejczykow
Jestem w Nairobi ! --------
Na swiecie jest 9 murzynow, oraz 8 europejczykow
Koniec programu
```

Istota programu polegała na pokazaniu, że dzięki deklaracjom funkcje z jednego pliku programu mogą być wywoływane przez funkcje z innego pliku składającego się na program.

Także zmienne globalne z jednego pliku mogły być używane w innym pliku (mówimy module programu). Wszystko dlatego, że do obu plików `europa.c` i `afryka.c` wstawiliśmy plik nagłówkowy `nagl.h` zawierający deklaracje funkcji i zmiennych.

–Mam Cię! - zawołałeś już pewnie triumfalnie - Jak można włączać plik, w którym jest linijka:

```
extern int ile_murzynow ;
```

do pliku `afryka.c`, w którym zaraz jest definicja

```
int ile_murzynow = 9 ;
```

Najpierw mówimy, że deklarujemy, iż obiekt `int` o nazwie `ile_murzynow` jest gdzieś indziej (external – znaczy po angielsku: zewnętrzny), a zaraz potem definiujemy zmienną `int ile_murzynow` właśnie w tym pliku! To oszustwo!

Brawo, brawo. To znaczy, że jesteś czujny. Oto moja obrona. Nie jest to oszustwem pod warunkiem, że nie bierzemy na serio słowa `extern`. To słowo nie oznacza, że obiekt jest zdefiniowany koniecznie na zewnątrz. Przypomnij sobie niedawne moje słowa:

> *... Gdzie się ten obiekt znajduje? Nie mówię teraz, bo może nawet na zewnątrz (`extern`) tego pliku. (Ale to nic pewnego)...*

Otóż słowo `extern` znaczy tu tylko to, że **deklarujemy** obiekty, ale ich nie definiujemy. Przypomnij sobie też, że mówiliśmy, iż obiekt może mieć dowolną liczbę deklaracji, ale tylko jedną definicję.

Zatem poprawna jest taka sekwencja w programie:

```
extern int k ;      //deklaracja
extern int k ;      //deklaracja
int k ;                 // definicja - tylko jedna !
extern int k ;      //deklaracja
extern int k ;      //deklaracja
```

Natomiast nie jest poprawna taka sekwencja:

```
extern int k ;          // deklaracja
extern int k ;          // deklaracja, druga, nie szkodzi
int k ;                 // definicja !
int k ;                 // definicja - powtórzona to błąd!
```

Jeśli jednak trawi Cię uczciwość i chciałbyś dosłownie rozumieć słowo `extern`, to czujesz chyba teraz dlaczego zrezygnowano z takiej dosłowności – po to, by nie budować dla każdego modułu programu osobnego pliku nagłówkowego. Jeden plik nagłówkowy może pomieścić te same deklaracje i być wstawianym do różnych modułów programu. W naszym wypadku pliki `europa.c` i `afryka.c` pracują z tym samym plikiem nagłówkowym `nagl.h`

A jeśli i to Cię nie przekonuje i od słowa `extern` żądasz dosłowności, to zrób teraz dwa pliki nagłówkowe – osobny dla `afryki`, a osobny dla `europy`. Za karę !

Zagądka

Jak sądzisz, czy poprawny jest taki zapis:

```
extern int m = 4 ;
```

Jest w tym pewna sprzeczność, bo z jednej strony używamy słowa `extern` - charakterystycznego dla deklaracji, a z drugiej dokonujemy inicjalizacji liczbą 4 – co jest charakterystyczne dla definicji.

Oto odpowiedź:
Kompilator widząc taki zapis uzna, że jest to **definicja**, czyli tak samo jakby był to zapis

```
int m = 4 ;
```

5.11.1 Nazwy statyczne globalne

Jeśli w deklaracji nazwy globalnej postawimy przydomek `static` to oznacza to, że nie życzymy sobie, by ta nazwa była znana w innych plikach (modułach) składających się na nasz program. Podkreślam, że chodzi tu o nazwy, które są deklarowane globalnie (czyli na zewnątrz wszystkich funkcji).

Za sprawą tego przydomka nazwa jest nadal globalna, ale może być znana tylko w tym jednym pliku. Dotyczy to nie tylko nazw obiektów, ale także nazw funkcji. Wyznam, że nie wiem dlaczego służy do tego ten sam przydomek `static`, skoro chodzi tu o zupełnie inne znaczenie niż poprzednio. Sądzę, że musiały tu być jakieś „względy historyczne".

Powodem by globalną nazwę funkcji czy obiektu zaopatrzyć w ten przydomek `static` (czyli tutaj jakby: ściśle tajne) jest najczęściej chęć by inne moduły (pliki) programu, które danej funkcji czy zmiennej nigdy nie mają używać – nie musiały dbać o unikalność nazw.

Przykładowo: w większym zespole piszemy program, a ja pisząc mój fragment wymyślam jakieś nazwy potrzebne mi w moim module. Niektóre z tych nazw będą ważne i ustalane ze wszystkimi kolegami – np. funkcje, które oni mają z mojego modułu wywoływać. Jednak oprócz takich nazw będą też nazwy moich globalnych zmiennych pomocniczych czy nazwy pomocniczych funkcji.

Teoretycznie powinienem wszystkich kolegów o tych nazwach także informować – by nie wymyślili przez przypadek funkcji o identycznej nazwie.

Po co ten kłopot! Wystarczy, że przed tymi globalnymi nazwami umieszczę przydomek `static`. Sprawi to, że owe nazwy nie będą wówczas znane w modułach innych niż mój. Od tej pory nie ma więc obawy o kolizję nazw.

Sposób ten bardzo przydaje się w wypadku pisania bibliotek.

5.12 Funkcje biblioteczne

Programować w C czy C++ to w zasadzie żadna sztuka, wystarczy opanować tych kilkanaście instrukcji, wiedzieć jakie są operatory i jeszcze parę drobnych rzeczy. Po krótkim czasie nie będziesz musiał w trakcie programowania zaglądać do podręcznika. Jest jednak coś, co zawsze leżeć będzie na Twoim biurku: opis funkcji bibliotecznych.

Funkcje biblioteczne nie są częścią języka C i C++. Są to po prostu funkcje, które ktoś tam napisał, a okazały się tak dobre i tak często przydatne, że zrobiono z nich standardową bibliotekę. Biblioteka ta stała się tak popularna wśród programistów, że każdy producent kompilatora C++ musi ją także dostarczyć. Muszą być w niej funkcje, które wykonują pewne standardowe usługi i w dodatku poszczególne funkcje muszą nazywać się tak samo, jak analogiczne funkcje w innych wersjach kompilatora.

Przykładowo:

Kiedyś ktoś napisał funkcję, która zamienia litery w taki sposób, że jeśli wyślemy jej jako argument wielką literę 'W' to w odpowiedzi jako rezultat dostaniemy małą literę 'w'. Czyli funkcja z wielkich liter robi małe. Natomiast małym literom nie szkodzi. Zwraca je jako małe. Nie szkodzi też cyfrom, znakom specjalnym itd. Oto realizacja takiej funkcji. Nazywamy ją `tolower` (ang. – do niższej).

Dygresja :

> *Nazwa pochodzi od angielskiego określenia liter małych jako „lower case" (niższa kasetka) – wielkie litery nazywają się „upper case" (wyższa kasetka). Określenia te pochodzą od pracy zecerów dawniej składających teksty w drukarni ręcznie. Litery takie znajdowały się w dwóch kasetkach z przegródkami. Kasetka z małymi literami, jako częściej używanymi, leżała bezpośrednio przed zecerem, natomiast po litery wielkie musiał on sięgać wyżej.*

A oto jak wygląda nasza funkcja:

```
/**************************************************/
char tolower(char znak)
{
int roznica = 'a' - 'A' ;

     if( (znak >= 'A') && (znak <= 'Z') )
          return (znak - roznica) ;
     else
          return znak ;
}
/**************************************************/
```

Jak ta funkcja działa? Przysyłamy do niej kod znaku (jego reprezentację liczbową). Następuje sprawdzenie czy jest to znak z przedziału A - Z. Robimy to przez sprawdzenie czy kod ASCII przysłanego znaku jest większy lub równy kodowi znaku 'A' i równocześnie mniejszy lub równy kodowi znaku 'Z'.

Jeśli tak, to do kodu znaku dodajemy liczbę `roznica`. Skąd wiemy, że trzeba dodać właśnie tyle? Z tablicy kodów znaków ASCII. Z tablicy takiej łatwo znajdziemy, że

```
A - 65          a - 97
B - 66          b - 98
C - 67          c - 99
.............   ................
Z - 90          z - 122
```

Nie jest to przypadkiem, że różnica między kodem litery wielkiej, a kodem odpowiadającej jej litery małej jest zawsze stała. Dzięki temu, aby ze znaku ASCII reprezentującego literę wielką zrobić znak reprezentujący literę małą, wystarczy dodać do jego kodu tę właśnie różnicę)

```
32                    // czyli: 97 - 65
```

> We wszystkich innych powszechnie stosowanych kodach powinna również być zachowana zasada, że kolejne litery alfabetu reprezentowane są kolejnymi kodami liczbowymi.

Jeśli natomiast przysłany do funkcji znak nie jest z interesującego nas przedziału, to go poprostu zwracamy, bez zmian.

Ktoś kiedyś napisał taką funkcję. Okazała się tak przydatna, że teraz jest w bibliotece standardowej.

Załóżmy jednak, że interesuje Cię ewentualna zamiana liter małych na wielkie. Co zatem robimy? Nie, nie zabieramy się do pisania! Bierzemy do ręki opis funkcji bibliotecznych dostępnych w naszym kompilatorze.

W opisie tym spisane są wszystkie dostępne standardowe funkcje biblioteczne. Nie musisz ich znać, nie musisz ich nawet używać. Prawie wszystko możesz przecież sobie napisać samemu. Będzie to jednak wyważanie otwartych drzwi.

Uwaga dla programistów C

> Jeśli znasz standardowe funkcje biblioteczne z klasycznego C, to znajdziesz je wszystkie także i w C++, więc nie musisz się niczego uczyć.

W opisie funkcji bibliotecznych, funkcje takie są czasami uszeregowane alfabetycznie co do swojej nazwy, czasami podzielone na rozdziały według tego, do czego służą. Odnalezienie właściwej nie jest rzeczą trudną. W naszym wypadku łatwo dojdziemy, że obok funkcji `tolower` jest funkcja `toupper`, która zamienia literę małą na wielką. Tego właśnie szukaliśmy.

Jednak powiedzmy jasno: w opisie nie znajdziemy tekstu (ciała) funkcji, takiego jak to na przykład napisaliśmy dla funkcji `tolower`. Jest tam tylko opisane co ta funkcja robi, z jakimi argumentami należy ją wywoływać i jaki typ funkcja ta

zwraca. Natomiast nie mówi się jak jest to zrealizowane. Tym lepiej, bo komuś, kto chce posłużyć się daną funkcją, nie jest potrzebna znajomość wszystkich sprytnych sztuczek, którymi posłużył się piszący tę funkcję.

Dodatkowo przy opisie danej funkcji bibliotecznej znajdziemy też informację, gdzie znajduje się deklaracja tej funkcji. Po co ta informacja?

W zasadzie mówiliśmy już o tym kilkakrotnie. Pamiętasz zapewne — mówiliśmy, iż każda nazwa w języku C++ musi być zadeklarowana zanim zostanie użyta. W naszym wypadku `toupper` to też taka nazwa. Jest to nazwa funkcji – musi więc znaleźć się w programie deklaracja tej funkcji. Abyśmy nie musieli pisać jej sami (przepisując z opisu biblioteki) została ona już wpisana do pliku nagłówkowego o nazwie `ctype.h`

Wystarczy więc wstawić ten plik nagłówkowy do tekstu naszego programu za pomocą dyrektywy preprocesora

```
#include <ctype.h>
```

i sprawa jest załatwiona. Jeśli byśmy o tym zapomnieli to kompilator C++ nam tego nie podaruje i upomni się o nią.

(Kompilator klasycznego języka C by nam to podarował, ale ja osobiście taką pobłażliwość kompilatora C okupiłem już kilkoma godzinami poszukiwań błędów w programie. Dlatego wolę teraz ten bardziej krytyczny język C++).

Jak widać z funkcjami bibliotecznymi sprawa jest prosta: najpierw szukamy funkcji w opisie[†)], potem wstawiamy do programu plik nagłówkowy z jej deklaracją, a następnie posługujemy się tą funkcją w programie. Tak, jakby to była nasza własna funkcja.

Funkcja jest jednak zdefiniowana w pliku bibliotecznym. Plik ten trzeba dołączyć w czasie linkowania (czyli łączenia). Jest to zwykle bardzo proste, ale tu niestety nie mogę Ci nic pomóc. Jak to zrobić – znajdziesz w opisie swojego kompilatora.

Najczęściej jednak jest to tak zintegrowane z kompilatorem, iż nie trzeba wydawać specjalnych rozkazów.

Co chciałbym, abyś z tego paragrafu zapamiętał:

To mianowicie, że opis (leksykon) funkcji bibliotecznych będzie Twoim najlepszym przyjacielem. Trochę tak, jak „ściąga" na kolokwium.

Osobiście zawsze, jak tylko przyjdzie mi pracować na nowym typie komputera, łapczywie rzucam się na opisy funkcji bibliotecznych. Znajdują się tam bardzo różne ciekawe funkcje. Oprócz funkcji naprawdę standardowych takich jak np. obliczenie x do potęgi 17/23, znaleźć tam można też takie, które posłużą nam do narysowania na błękitnym ekranie białego kółka wypełnionego deseniem z czerwonych serduszek.

†) zwanym po angielsku: Reference Manual

6 Preprocesor

Preprocesor to jakby przednia straż kompilatora. Zanim kompilator przystępuje do akcji, tekst programu jest przeglądany przez preprocesor. Zwykle preprocesor wkracza do pracy sam, bez jakiejkolwiek inicjatywy z naszej strony. W zasadzie byłby dla nas niezauważalny i nieistotny, gdyby nie kilka przysług, które może nam oddać.

O tym, co ma zrobić dla nas preprocesor, decydujemy za pomocą tak zwanych ***dyrektyw preprocesora***. Dyrektywy takie są umieszczane przez nas w stosownych miejscach programu, a ich znakiem szczególnym jest to, że pierwszym nie-białym znakiem takiej linijki jest znak #. Przed tym znakiem mogą być w linijce tylko spacje i tabulatory.

Z niektórymi takimi dyrektywami już się oswoiliśmy. Na przykład wielokrotnie występowała już w naszych programach dyrektywa

```
#include <iostream.h>
```

Dyrektywy są żądaniami, by preprocesor wykonał jakąś akcję. Preprocesor podejmuje ją zwykle na nasze wyraźne żądanie. Jest jednak akcja, która zostanie wykonana przez niego samoczynnie. O tym mówi poniższy paragraf.

6.1 Na pomoc rodakom

Jeśli Twój komputer ma na klawiaturze znaki:

```
\   ^   [   ]   {   }   !
```

to możesz opuścić ten paragraf i zacząć czytać następny. Mówić tu będziemy o tym, jak preprocesor pomaga programistom, którzy takich znaków na klawiaturach nie mają.

Zatem, jeśli nie masz na klawiaturze tych znaków, to po pierwsze bardzo Ci współczuję i zastanawiam się jakim cudem dobrnąłeś w tej książce aż tutaj. Wiemy jak bardzo ważne i częste są te znaki w programowaniu w C++. Niestety, w niektórych krajach komputery mają w miejscu tych znaków swoje znaki narodowe. Przykładem niech będzie Dania, gdzie w tych miejscach są znaki typu:

```
Æ    Å    å
```

Pewnie nikt by się biednymi Duńczykami nie przejął, gdyby nie to, że twórca języka C++ Bjarne Stroustrup pochodzi z Danii.[†)] Także w innych krajach mogą być na klawiaturach jakieś inne znaki zamiast tych, o które nam chodzi. Co wtedy?

Wyjście jest takie: Nieobecne na klawiaturze znaki można w programie uzyskać zastępując je tak zwanymi *sekwencjami trzyznakowymi.* Są one tak dobrane, że odlegle przypominają znak, który mają symulować. Np. sekwencja `??(` oznacza to samo co znak `[`

Oto zebrane sekwencje i ich odpowiedniki:

```
??=      #        ??(      [
??/      \        ??)      ]
??'      ^        ??<      {
??!      |        ??>      }
```

Prosty program wygląda z użyciem tych symboli tak:

```
??=include <iostream.h>
main()
??<
     int i ;
     char tab??(30??) ;

     for( i = 0 ; i < 30 ; i ++)
     ??<
          tabl??(i??) = 'a' + i ;
          cout << "zaladowanie do elementu " << i
               << " wartosci " << tabl??(i??) << endl ;
     ??>
     cout << "Poszukiwanie litery k lub m ??/n" ;

     for(i = 0 ; i < 30 ; i ++)
     ??<
          if((tabl??(i??)=='k') ??!??! (tabl??(i??)=='m')
          ??<
               cout << "Znak " << tabl??(i??)
                    << "jest w elemencie " << i << endl ;
          ??>
     ??>
??>
```

†) Mieszka w USA i pracuje w AT&T.

Jeśli powyższy program wydaje Ci się mało czytelny, to nie ma rady - musisz pomyśleć o zmianie klawiatury Twojego komputera.

6.2 Dyrektywa #define

Ta dyrektywa preprocesora ma postać:

`#define` *wyraz* *ciąg znaków zastępujących go*

Dyrektywa ta[†)] powoduje, że w kompilowanym pliku każde następne wystąpienie słowa *wyraz* będzie zastępowane wyszczególnionym ciągiem znaków. Oczywiście nie musi być to koniecznie wyraz. Musi być to jednak grupa znaków bez białego znaku w środku. To natomiast, co jest ciągiem znaków zastępujących, może mieć w środku białe znaki.

Na przykład poniższy fragment

```
#define CZTERY 4
     // ...
     float tablica[CZTERY] ;
     i = CZTERY + 2 * CZTERY ;
     funkcja(CZTERY- 0.3) ;
     cout << "Mowilem ci to CZTERY razy " ;
```

zostanie przez preprocesor automatycznie zamieniony na

```
     // ...
     float tablica[4] ;
     i = 4 + 2 * 4 ;
     funkcja(4 - 0.3) ;
     cout << "Mowilem ci to CZTERY razy " ;
```

Zwyczajowo wyrazy zastępowane pisze się wielkimi literami po to, by w tekście programu przypomnieć sobie, że nie chodzi tu o nazwę obiektu, lecz o działanie dyrektywy `define`. Powtarzam – jest to tylko zwyczaj. Równie dobrze można pisać litery małe.

Zauważ, że dyrektywa `define` nie penetruje wnętrza stringów (czyli tzw. stałych tekstowych). Jest to oczywiście zupełnie logiczne. Wewnątrz stringów bowiem, zupełnie przypadkowo mogą się pojawić identyczne wyrazy jak ten zastępowany, ale w zupełnie innych znaczeniach i kontekstach. Lepiej więc, że `define` nie ma do nich dostępu.

Oto dalsze przykłady:

```
#define MAX_LICZ_PASAZER     250
#define LICZB_STEWARD   8
#define PASAZ_NA_STEWD (MAX_LICZ_PASAZER/ LICZ_STEWARD)
```

†) (czytaj: „defajn")

Jak widać następne dyrektywy `define` mogą korzystać z właśnie zdefiniowanych powyżej nazw.

Dyrektywą `define` można definiować zastępowanie dowolnego ciągu znaków. Przykładowo – po takiej dyrektywie

```
#define ZEGAREK    (while(!zajety) czas(); )
```

można w programie zastosować zapis

```
funkcja_a();
     i =15 * czynnik ;
     ZEGAREK ;
     x = 15 * log(17);
```

co odpowiada zapisowi

```
funkcja_a();
     i =15 * czynnik ;
     (while(!zajety) czas(); ) ;
     x = 15 * log(17);
```

Czyli dyrektywą `define` możemy sobie oszczędzić trochę pracy przy pisaniu długich, a często występujących instrukcji. Zauważ, że dyrektywa `define` nie ma na końcu średnika. To dlatego, że nie jest to normalna instrukcja programu, lecz dyrektywa dla preprocesora. Jeśli przez zapomnienie postawimy średnik na końcu, to zostanie on uznany jako jeden ze znaków zastępujących. Czyli w wypadku naszej definicji

```
#define CZTERY 4 ;
```

ten sam fragment wyglądałby następująco:

```
// ...
float tablica[4 ;] ;
i = 4 ; + 2 * 4 ; ;
funkcja(4; - 0.3) ;
cout << "Mowilem ci to CZTERY razy " ;
```

Oczywiście na skutek wstawienia tego średnika, znalazł się on w zupełnie nieodpowiednich dla siebie miejscach – wywołując w trakcie kompilacji komunikaty o błędach. Taki błąd jednak wykrywa się natychmiast, więc nie jest to groźne.

Bardzo długie dyrektywy

Może się zdarzyć, że ciąg znaków zastępujących jest długi i z tego powodu trudno umieścić dyrektywę w jednej linii. Nie ma problemu: gdy uznamy, że linia powinna się skończyć - umieszczamy na jej końcu znak \ (bekslesz) i kontynuujemy pisanie dyrektywy w linijce następnej. Tym sposobem dyrektywa może się ciągnąć przez wiele linijek.

```
#define HMI Hahn-Meitner Institut\
fuer Kernforschung\
W.Berlin 39, Glienickerstr 100
```

Zwracam uwagę, że chwyt ten stosować trzeba tylko wobec dyrektywy preprocesora. Zwykłą instrukcję programu można przecież swobodnie umieszczać w kilku linijkach. To znamy od dawna.

Definiowanie stałych

W języku C klasycznym dyrektywa ta tradycyjnie używana była do definiowania stałych. W języku C++ oprócz tego sposobu mamy inny, lepszy: mianowicie obiekty typu `const`.

Oto porównanie. Stary styl:

```
#define ROZDZIELCZOSC 8192
long widmo[ROZDZIELCZOSC] ;
```

Nowy styl:

```
const int rozdzielczosc = 8192 ;
long widmo[rozdzielczosc] ;
```

O tym, żeby nie używać `define` jako sposobu deklarowania stałych mówiliśmy już przy okazji definiowania obiektów typu `const`. (str. 47)

W skrócie to można streścić tak:

> Stosując definiowanie stałych jako obiekty typu `const` dajemy kompilatorowi większe szanse wykrycia naszych ewentualnych omyłek.

Innym poważnym zastosowaniem dyrektywy `define` było definiowanie tak zwanych *makrodefinicji*. To także nie wytrzymało próby czasu, o czym porozmawiamy za chwilę.

Zastosowaniem, które próbę czasu wytrzymało, jest między innymi definiowanie symboli dla kompilacji warunkowej. Także o tym będziemy mówili niebawem.

Na koniec jeszcze jedna uwaga. Dużo się napracowałem by wpoić Ci zasadę, że definicja to jest moment, gdy rezerwuje się dla jakiegoś obiektu miejsce w pamięci – czyli inaczej, jest to miejsce w programie, gdzie dany obiekt się rodzi. Tymczasem tutaj słowo `define` znaczy coś zupełnie innego. Tutaj tylko określamy, że jak napotka się jeden ciąg znaków, to należy go zamienić na inny ciąg znaków. Żaden obiekt tutaj nie powstaje. Zapytasz: gdzie tu konsekwencja?

Nie ma żadnej konsekwencji i przepraszam Cię za to. Na usprawiedliwienie dodam, że tamto mówiłem o kompilatorze. W tym rozdziale rozmawiamy o jego służącym – preprocesorze, który przychodzi jeszcze zanim zjawia się prawdziwy kompilator. Nie wymagajmy więc konsekwencji od jego głupszego służącego. Jak widać nieco innym językiem mówi się do preprocesora - czyli: proszę o pobłażliwość w linijkach zaczynających się od znaku # .

6.3 Dyrektywa #undef

Jeśli dyrektywą `define` określiliśmy jakąś nazwę, to owo polecenie (nie chce mi przejść przez gardło – ta: *definicja*) obowiązuje od momentu wystąpienia tej linii w programie, a ważne jest do końca pliku.

Czasem jednak może nam zależeć by preprocesor zapomniał o poleceniu wydanym mu dyrektywą `define`. W tym celu wystarczy użyć dyrektywy†)

`#undef` *wyraz*

Począwszy od tego miejsca w programie, przetwarzanie dalszych linijek będzie się odbywało tak, jakbyśmy poprzedniej dyrektywy

`#define` *wyraz* .....

nie wydawali. Uczciwie muszę przyznać, że jeszcze nigdy tej dyrektywy nie używałem.

6.4 Makrodefinicje

Podobnie jak w języku C, tak i tu w C++, dyrektywa `define` może służyć do tworzenia makrodefinicji.

Rozważmy taki przypadek:

```
#define KWADR(a)    ((a) * (a))
```

Jak to działa? Otóż przed przystąpieniem do właściwej kompilacji – preprocesor (czyli straż przednia kompilatora) – zamienia w tekście programu wszelkie wystąpienia wyrażenia

```
KWADR(parametr)
```

na wyrażenie

```
( (parametr) * (parametr) )
```

Po takiej zamianie przystępuje się do właściwej kompilacji. Inaczej mówiąc następujące linijki programu

```
a = KWADR(c) + KWADR(x) ;
cout << KWARD(m+5.4) ;
```

zamienią się automatycznie na linijki

```
a = ( (c) * (c) ) + ( (x) * ( x) ) ;
cout << ( (m + 5.4) * (m + 5.4) ) ;
```

Do czego może służyć taka makrodefinicja? Na przykład do tego, by zamiast stosować w linii skomplikowany zapis – uprościć go sobie.

†) To skrót od ang. undefine (czytaj: „andef")

W makrodefinicji może być też więcej parametrów

```
#define OBJECT(a,b,c) ( (a) * (b) * (c) )
```

Przypominam, że nie może być spacji ani – ogólniej – białych znaków w wyrażeniu (słowie) `OBJECT(a,b,c)`. Biały znak bowiem kończy określenie makrodefinicji, a zaczyna określenie tego, czym ma ona być zastąpiona. Czyli jakby otwiera jakby ciało tej „funkcji". (Naprawdę nazywamy to **rozwinięciem makrodefinicji**).

`Inline` contra makrodefinicja

Paragraf ten piszę między innymi dlatego, by obrzydzić Ci ochotę do używania makrodefinicji. Myślę, że nie przyjdzie mi to trudno.

Na pewno, drogi Czytelniku, pomyślałeś już o funkcjach typu `inline`, o których rozmawialiśmy w poprzednim rozdziale. Zawołasz pewnie: „To przecież to samo!" Nie, nie to samo. Prawie to samo, ale są różnice. To z powodu tych różnic szykuję czarną propagandę.

Najważniejsza rożnica to to, że makrodefinicja jest jakby tępym narzędziem, **mechanicznym zamienianiem jednego stringu na drugi.** Nie ma tu sprawdzania typów parametrów (jakby argumentów), nie ma sprawdzania zakresu ważności użytych nazw. Kompilator nie może nas ostrzec czy popełniliśmy jakiś błąd. Dlatego makrodefinicji nie radzę używać. Lepiej zastosować funkcje typu `inline`, która nam to wszystko zagwarantuje.

Drugi powód to **nieoczekiwane efekty uboczne**.

Rozważmy taki przypadek. Mamy naszą makrodefinicję

```
#define KWADR(a)    ( (a) * (a) )
```

i zastosujemy ją w takim wyrażeniu

```
int x = 4, p ;
p = KWADR(x++) ;
cout << "p = " << p << ", x teraz = " << x ;
```

W rezultacie wykonania tego fragmentu otrzymamy na ekranie

```
p = 20, x teraz = 6
```

Niezauważenie dla nas `x` zostało inkrementowane dwukrotnie. Wyniku spodziewaliśmy się także innego – przecież (4 * 4) = 16. Zatem dlaczego ? To proste. Łatwo to zrozumieć, gdy rozpiszemy sobie wyrażenie, w którym nastąpiła makrodefinicja. Wygląda ono wtedy tak:

```
p = ( (x++) * (x++) ) ;
```

Jak widać inkrementacja została wykonana dwukrotnie mimo, że zamierzaliśmy zrobić to jednokrotnie. Chciałbym zawołać teraz triumfalnie: „–A nie mówiłem?! Nie używajmy makrodefinicji wcale!". Tak jednak nie zawołam, gdyż są

Sytuacje kiedy makrodefinicja przydaje się†)

Zapytasz: „–Mimo, że nie sprawdza nam typu argumentów?" Nie tylko *mimo*, ale wręcz właśnie dlatego. Są sytuacje, gdy chcemy oszukać kompilator. Na przykład wtedy, gdy nie chcemy, by kompilator sprawdzał nam typ argumentów. Klasycznym przykładem jest tu makrodefinicja

```
#define MAX(a,b)    ( ((a) > (b)) ? (a) : (b) )
```

możemy z niej korzystać niezależnie czy porównujemy ze sobą dwie liczby czy dwa adresy, czy też znaki. Typ argumentów nie jest bowiem sprawdzany. Gdybyśmy jednak chcieli napisać to samo jako funkcję typu `inline`, to należałoby dokładnie określić typ argumentów. Wymagałoby to zapewne napisania kilku wersji takiej funkcji, zależnie od typu porównywanych argumentów.

Jeśli więc zdecydujemy się na posługiwanie się tą makrodefinicją, pamiętajmy, że tak samo jak poprzednia, może być ona źródłem wspomnianych efektów ubocznych.

Nawiasy

O jeszcze jednej rzeczy muszę wspomnieć: Czy zauważyłeś jak gęsto rozwinięcia makrodefinicji zaopatrywałem w nawiasy? Do tego stopnia, że wręcz trudno to było czytelne. Nie bez powodu. Weźmy taką makrodefinicję

```
#define WYR(a,b,c)  a * b + c          // ryzykowne !
```

Jeśli użyjemy tej makrodefinicji w taki sposób:

```
y = WYR(2, 1 + 6.5, 0) * 1000 ;
```

to w efekcie działania preprocesora linijka ta zamieni się na taką:

```
y = 2 * 1 + 6.5 + 0 * 1000 ;
```

Czyli zamiast obliczyć

```
(2 * (1+6.5) + 0 ) * 1000
```

obliczymy

```
(2 * 1) + 6.5 + (0 * 1000)
```

Aby się przed tym ustrzec, należy w naszej definicji zastosować nawiasy. Poprawnie powinna wyglądać tak:

```
#define WYR(a,b,c) ( (a) * (b) + (c) )
```

†) W najnowszych wersjach języka C++ nawet ta sytuacja odpada. Zamiast makrodefinicją lepiej posłużyć się narzędziem zwanym *szablonem funkcji.* O szablonach napisałem książkę p.t. „Pasja C++". Zajrzyj do niej po przeczytaniu „Symfonii".

6.5 Dyrektywy kompilacji warunkowej

Zdarza się, że chcielibyśmy, by pewne linijki programu pojawiały się w nim tylko wtedy, gdy tego zażądamy. Na przykład na etapie uruchamiania programu przydają się linijki wypisujące na ekranie wyniki pośrednie.

```
x = i * czynnik[r] + szereg(x0);
cout << "teraz x = " << x << endl ;      // pomocniczy wydruk
m = funkcja(x) ;
```

Potem jednak, gdy program działa poprawnie wydruki takie nie są już potrzebne. Teoretycznie można by więc je usunąć, no ale kto wie, czy kiedyś jeszcze przy robieniu jakiś modyfikacji nie przydadzą się.

Wyjściem jest oczywiście chwilowe ujęcie ich w znaki komentarza. Wyjście to nie jest dobre. Jeśli bowiem w programie mamy dużo takich wydruków kontrolnych rozsianych po różnych miejscach, to dużo się napracujemy ujmując je w komentarze.

Drugi powód jest jeszcze ważniejszy – jak pamiętasz komentarzy typu /*...*/ nie można zagnieżdżać. Jeśli chcemy instrukcję, (albo kilka instrukcji) ująć w taki komentarz, a komentarz typu /*...*/ już w tym fragmencie jakoś jest używany, to jesteśmy bezsilni. Kompilator, który nie pozwala na zagnieżdżanie komentarzy – próbę taką uzna za błąd.

Jest inny sposób. Wyjście to nazywa się *kompilacja warunkowa*. Polega to na tym, iż, w zależności od spełnienia pewnych warunków, określone linie programu są kompilowane lub nie. Realizujemy to za pomocą dyrektyw preprocesora. To on przygotowuje kompilatorowi materiał i to on określone linie programu może odrzucić z procesu kompilacji. Kompilacja odbędzie się tak, jakby te linijki nigdy w programie nie istniały.

Jak się to robi? Bardzo prosto. Otóż obszar kompilacji warunkowej ograniczamy linijkami będącymi odpowiednimi dyrektywami preprocesora.

Dyrektywa `#if`

```
#if warunek
        // linie kompilowane warunkowo
#endif
```

Jak widać przypomina to w pewnym sensie znaną nam instrukcję `if`. *Warunek* jest to stałe wyrażenie. Rozumiem to jako wyrażenie, w którym każdy element jest stały, a jego wartość jest już znana w czasie, gdy preprocesor pracuje nad tą linijką. Innymi słowy nie może tam wystąpić żaden z obiektów (np. zmiennych) występujących w programie. Warunek jest wtedy spełniony, gdy wyrażenie warunkowe jako całość ma wartość różną od zera („prawda"). Oto przykłady:

```
#define RODZAJ 2
// ...
cout << "To jest kompilowane zawsze " << endl ;      // ❶
#if (RODZAJ == 1)
    // ...                                            ❷
    cout << "To jest kompilowane, gdy "
             "warunek jest spelnony " << endl ;
```

```
#endif
cout <<"To znowu jest kompilowane zawsze "<<endl;    // ❸
```

Rzut oka wystarczy by stwierdzić, że warunek nie jest tu spełniony i linijka ❷ nie zostanie skompilowana. Po linijce ❶ nastąpi bezpośrednio linijka ❸.

Naszą dyrektywę

```
#endif
```

możemy też zapisać jako

```
#endif (RODZAJ == 1)
```

Postawiony tu warunek nic nie oznacza, a jednak bardzo się przydaje. W sytuacji, gdy parę`#if`'ów zagnieżdżonych jest jeden w drugim, bardzo ułatwia to rozeznanie, w którym miejscu kończy się jaki obszar.

Słyszałem jednak, że bywają kompilatory, które (wbrew regułom) nie tolerują powtórzenia takiego warunku. Można sobie jednak w tym wypadku radzić stawiając go po znaku komentarza.

```
#endif // (RODZAJ == 1)
```

Inne dyrektywy dla kompilacji warunkowej

```
#if warunek
          // instrukcje 1
#else
          // instrukcje 2
#endif
```

Jest dokładnie tak, jak się domyślasz: zależnie od spełnienia warunku albo do kompilacji wchodzą `instrukcje` 1, albo `instrukcje` 2.

Najczęściej taką wielowariantową kompilację warunkową stosujemy, gdy nasz program ma mieć wiele wariantów. Trzon jest ten sam, ale niektóre części są wymieniane zależnie od bieżącej wersji.

Na przykład: piszemy program sterujący wysuwaniem podwozia w samolocie. W różnych typach samolotu jest różny typ podwozia, jednak wszędzie zasada jest podobna, więc tylko niektóre funkcje trzeba wymienić na inne. Oczywiście można skopiować stary program i zmodyfikować robiąc osobny program dla jednego typu, osobny dla drugiego. Nie zawsze się to jednak opłaca. Jeśli bowiem pracując nad jedną wersją tego programu wpadniemy na jakiś dobry pomysł dotyczący tej części, która jest identyczna dla obu programów, to zamianę trzeba będzie wprowadzać dwukrotnie – raz w starym, a raz w nowym programie. Moja praktyka mówi mi, że takich (niekoniecznie od razu genialnych) przeróbek jest zwykle wiele. Opłaca się korzystać z dobrodziejstw kompilacji warunkowej.

Oto taki fragment:

```
// najpierw definujemy sobie dla wygody takie symbole
#define PODWOZIE_707    1
#define PODWOZIE_747    2
#define PODWOZIE_DC11   3
#define PODWOZIE_LIL    4
```

```
// tutaj definiujemy z którym typem mamy do czynienia
// w tym konkretnym wypadku
#define TYP_PODWOZIA PODWOZIE_747
//-----------------------------
int wystaw_kola(){
#if (TYP_PODWOZIA == PODWOZIE_707)
     cout << "Tak jest kapitanie wystawiam 707\n" ;
#elif (TYP_PODWOZIA == PODWOZIE_747)
     cout << "Tak jest kapitanie wystawiam 747\n" ;
#elif (TYP_PODWOZIA == PODWOZIE_DC11)
     cout << "Tak jest kapitanie wystawiam DC11\n" ;
#elif (TYP_PODWOZIA == PODWOZIE_LIL)
     cout << "Tak jest kapitanie wystawiam LIL\n" ;
#else
     cout << "To nigdy sie nie powinno zdarzyc \n" ;
     #error "zle zdefiniowana wersja programu !!!!"
#endif TYP_PODWOZIA
     return 1 ;
}
```

Ponieważ widać, że wersja programu zdefiniowana jest na typ podwozia `PODWOZIE_747`, zatem powyższy fragment zostanie kompilatorowi przedstawiony jako

```
int wystaw_kola()
{
     cout << "Tak jest kapitanie wystawiam 747\n" ;
     return 1 ;
}
```

W przykładzie tym zobaczyłeś nową dyrektywę

```
#elif
```

co jest jakby skrótem od `else if` i jest konstrukcją analogiczną do takiej właśnie kombinacji instrukcji.

Wybacz mi tę dyrektywę `#error`, o której do tej pory jeszcze nie wspominałem. Zdecydowałem się jednak wstawić ją tutaj na wypadek, gdybyś kiedyś do tego miejsca książki sięgnął szukając wzorca na zastosowanie kompilacji warunkowej różnych wersji swojego programu.

A swoją drogą to założę się, że domyślasz się jak działa dyrektywa `#error`. Jeśli nie, to i tak pomówimy o tym niebawem.

Warunek `#ifdef, #ifndef`

Dyrektywa kompilacji warunkowej

```
#ifdef nazwa
                    // rozumieć jako: if defined
     // ... instrukcje
#endif
```

nie sprawdza warunku, tylko sprawdza czy dana nazwa została zdefiniowana. To znaczy czy wystąpiła dyrektywa

```
#define nazwa ...
```

i nie unieważniliśmy jej dyrektywą

```
#undef nazwa
```

Jeśli tak, czyli nazwa jest preprocesorowi znana, to działanie jest takie, jakby chodziło tu o warunek i był on spełniony.

Analogicznie dyrektywa

```
#ifndef nazwa2
                    // rozumieć jako: if NOT defined
     // .... instrukcje
#endif
```

sprawdza czy nazwa została zdefiniowana i obowiązuje. Jeśli *nie* obowiązuje to tak, jakby w kompilacji warunkowej – warunek był spełniony.

Dyrektywy kompilacji warunkowej mogą być zagnieżdżane. Oto przykład:

```
#if (WERSJA == 1)
     #if (SZYBKOSC == 1)
                    // instrukcje (1)
     #else
                    // instrukcje (2)
     #endif
#else
     #if (SZYBKOSC == 1)
                    // instrukcje (3)
     #else
                    // instrukcje (4)
     #endif
#endif WERSJA
```

Jeśli w trakcie kompilowania tego fragmentu będą już w mocy następujące dyrektywy

```
#define WERSJA 7
#define SZYBKOSC 1
```

to w rezultacie w skład naszego skompilowanego programu wejdą *instrukcje (3)*.

6.6 Dyrektywa #error

Dyrektywa ta ma formę

```
#error string
```

a powoduje, że po napotkaniu jej kompilacja zostaje przerwana i wypisywany jest komunikat błędzie, którego częścią jest informacja umieszczona przez nas jako *string*

Oto kiedy się to przydaje:

```
#if (WERSJA == 1)
     // .....
#elif (WERSJA ==2)
     // .....
#else
     #error "Musi byc albo wersja 1 albo wersja 2!"
#endif
```

Jeśli zapomnieliśmy zdefiniować symbol `WERSJA`, albo jeśli zdefiniowaliśmy go tak, że ma on taką wartość, iż żaden ze sprawdzanych warunków nie jest spełniony, wówczas w preprocesor natknie się na naszą dyrektywę `#error`. Dyrektywa ta spowoduje przerwanie dalszej pracy i wypisanie w komunikacie o błędzie proponowanego tekstu:

```
Error directive: "Musi byc albo wersja 1 albo wersja 2!"
in function ...
```

6.7 Dyrektywa `#line`

Ma ona postać

`#line` *stała* "nazwa_pliku"

Proponowana nazwa_pliku nie jest obowiązkowa. Dyrektywa ta służy do oszukiwania kompilatora. Jeśli nakażemy mu

```
#line      128 "fikcja.c"
```

to od tej pory kompilator uzna, że jest to linijka 128 programu – mimo, że naprawdę jest to linijka numer 10 (na przykład). Dodatkowo kompilator będzie myślał, że kompiluje plik o nazwie "fikcja.c" Właściwą nazwę w tym momencie zapomina.

Dajmy przykład. Mamy plik TST.C z programem, o którym wiemy, że ma błędy w linijce 10-tej i 20-tej. (Wiemy to – bo na użytek tego przykładu specjalnie je tam popełniliśmy). Gdy kompilujemy taki program, to otrzymujemy informację mniej więcej takiej treści:

```
Plik TST.C ma bledy w linijce 10 i 20
```

Następuje teraz bliższe opisanie tych błędów. Robimy jednak taki eksperyment: Na początku programu dopisujemy jeszcze jedną linijką takiej treści

```
#line 500 "fikcja.c"
```

Dopisanie linijki spowoduje oczywiście, że wszystkie dalsze przesuną się o jedną linię w dół. Błędne są więc teraz linie 11 i 21. Spróbujemy teraz skompilować ten sam program. Otrzymamy znowu komunikat o błędzie, ale tym razem takiej mniej więcej treści:

```
Plik FIKCJA.C ma bledy w linijce 511 i 521
```

6.8 Wstawianie treści innych plików w tekst kompilowanego właśnie pliku

Dyrektywy preprocesora `#include` †)

```
#include <nazwa_pliku_A>
#include "nazwa_pliku_B"
```

powodują, że w tekst kompilowanego właśnie programu, w miejsce, w którym znajdują się takie dyrektywy, zostaje wstawiona treść innego pliku o wyszczególnionej nazwie. Zupełnie tak, jakbyśmy w tym miejscu w programie, będąc jeszcze w edytorze, sprowadzili w to miejsce treść rzeczonego pliku.

Dlaczego tak więc po prostu nie zrobić ?

- Pierwsza odpowiedź brzmi: z lenistwa.
- Po drugie: jeśli sprowadzanym plikiem jest plik nagłówkowy przy ewentualnych zmianach jakiejś deklaracji – wystarczy korekta w jednym tylko pliku.
- Po trzecie: – gdybyśmy chcieli do naszego programu włączyć wszystkie pliki nagłówkowe z deklaracjami funkcji bibliotecznych, to nasze pliki programowe rozrastałyby się ogromnie. Lepiej więc plik nagłówkowy wypożyczyć na chwilę, na samą okoliczność kompilacji.

A teraz o różnicy między przedstawionymi formami tej dyrektywy

Jeśli przy nazwie pliku użyliśmy cudzysłowu, to plik, który nakazujemy włączyć będzie najpierw poszukiwany w bieżącym katalogu, a jeśli tam nie zostanie znaleziony, to będzie szukany tak, jakby zamiast znaków cudzysłowu użyte były tam znaki < > .

Jeśli zaś przy nazwie plików są znaki < > to plik będzie poszukiwany w standardowym miejscu, gdzie znajdują się pliki zwykle włączane (np. biblioteczne). Miejsca poszukiwania plików włączanych tą dyrektywą są jednak zależne od implementacji, więc należy się zawsze upewnić jak w takim wypadku postępuje nasz kompilator.

Reguła jednak jest przeważnie taka :

> używając cudzysłowu włącza się pliki, które sami piszemy, natomiast znaki < > stosuje się włączając np. pliki nagłówkowe bibliotek. (Są one zwykle zgromadzone w jakimś specjalnym katalogu).

Dyrektywy `#include` mogą się zagnieżdżać. To znaczy, że gdy tą dyrektywą włączamy jakiś inny plik, to w nim może być dyrektywa `#include` włączająca jakiś jeszcze inny plik. Poziomów zagnieżdżenia może być wiele.

Jak zagwarantować sobie jednokrotne włączanie danego pliku ?

Może się zdarzyć, że dyrektywą `#include` włączamy do programu pliki A, B, C, X. Tymczasem w pliku B jest – o czym nie wiemy lub nie pamiętamy –

†) include – ang. = wstawiać, włączać, wcielać. (Czytaj: „inklud")

dyrektywa `#include` włączająca plik X. W związku z tym do kompilacji programu użyte zostaną wchodzące teraz w skład naszego programu pliki: A, B, X, C, X. Może to spowodować problemy, gdy w pliku X są fragmenty, które nigdy nie powinny pojawiać się w kompilacji dwukrotnie. (Np. definicje zmiennych lub definicje funkcji).

Jak się przed tym ustrzec? Jest prosty sposób: Użycie kompilacji warunkowej. Plik X powinien wyglądać na przykład tak:

```
#ifndef PLIK_X
#define PLIK_X

    // zwykła treść pliku

#endif
```

Jeśli ten plik zostanie włączony do kompilacji choć raz – zdefiniowana zostanie nazwa PLIK_X. Ewentualna powtórna próba włączenia tego pliku odbędzie się, gdy ta nazwa już jest zdefiniowana – czyli na mocy kompilacji warunkowej (dyrektywa `#ifndef`) nic z tego pliku po raz drugi kompilowane nie będzie.

6.9 Sklejacz czyli operator

To bardzo ciekawy operator. Jego działanie jest takie, że dokleja on jeden z argumentów (parametrów) makrodefinicji do stojącego po jego lewej stronie słowa.

Najlepiej zobaczyć to na przykładzie:

```
#define ST(rodzaj)      statecznik ## rodzaj

int ST(poziomy) ;
int ST(pionowy) ;
```

w rezultacie działania sklejacza taki fragment zostanie przetłumaczony przez preprocesor jako

```
int statecznikpoziomy ;
int statecznikpionowy ;
```

Zauważ, że spacje po jednej i drugiej stronie znaków ## zostały usunięte i nastąpiło rzeczywiście doklejenie do słowa `statecznik`, w rezultacie czego powstał jeden wyraz.

```
#define BOE(typ,co) boeing_ ## typ ## _ ## co ## _cena
```

w wypadku zapisu

```
cout << BOE(747, skrzydlo) ;
```

Jest on zastąpiony przez

```
cout << boeing_747_skrzydlo_cena ;
```

Jak widać dzięki temu operatorowi zaoszczędzić można pisania.

6.10 Dyrektywa pusta

Jest to dyrektywa składająca się z samego znaku

```
#
```

Taka dyrektywa nie ma żadnego działania. Jest przez preprocesor po prostu ignorowana.

6.11 Dyrektywy zależne od implementacji

Dyrektywy takie zaczynają się od słowa `pragma`, po którym następuje komenda charakterystyczna dla danego konkretnego kompilatora (preprocesora)

`#pragma` ***komenda***

Dzięki temu wprowadzona zostaje możliwość używania dyrektyw właściwych dla danego kompilatora, zatem szczegółowego opisu tych dyrektyw trzeba szukać w opisie kompilatora, którym się posługujemy.

Jeśli *komenda* jest danemu konkretnemu typowi kompilatora nieznana, napotkawszy ją w programie – ignoruje ją.

6.12 Nazwy predefiniowane

W trakcie pracy preprocesora oprócz nazw, które sami zdefiniowaliśmy, są jeszcze nazwy†), które definiuje dla siebie sam preprocesor. Oto one:

`__LINE__`

- ta nazwa kryje w sobie numer linijki pliku, nad którą preprocesor właśnie pracuje. Łatwo się domyślić, że jeśli kompilator wykrywa błąd i wypisuje komunikat o błędzie w linijce nr... to posługuje się właśnie tą nazwą.

`__NAME__`

- pod tą nazwą zapamiętana jest nazwa właśnie kompilowanego pliku

`__DATE__`

- pod tą nazwą zdefiniowany jest ciąg znaków odpowiadający dacie w momencie kompilacji. Mogą być dwie formy

†) Uwaga, w nazwach występują stojące obok siebie dwa znaki podkreślenia, które tu, w druku książki wyglądają niestety jak jedna długa kreska. Naprawdę jest wiec _ _ a nie __

```
"Mar 19 1995"
"Mar 3 1995"
```

zależnie od tego, czy numer dnia jest jedno- czy dwucyfrowy.

`__TIME__`

- ta nazwa kryje w sobie aktualny czas w momencie translacji. Ma on postać ciągu znaków (stringu) "hh:mm:ss". Na przykład

```
"15:45:08"
```

O istnieniu tych predefiniowanych nazw można się przekonać kompilując plik, w którym znajdą się następujące linijki.

```
cout << "Kompilacja tego pliku " << __FILE__ ;
cout << "\n (linijka "           << __LINE__
     << ") \n zaczela sie : "     << __DATE__
     << " o godzinie : "          << __TIME__ << endl;
```

Po uruchomieniu takiego fragmentu programu na ekranie zostanie wypisany tekst

```
Kompilacja tego pliku T.C
 (linijka 10)
 zaczela sie : Jun 04 1992 o godzinie : 00:45:50
```

Na wszelki wypadek podkreślę jeden fakt: Jeśli za 10 minut uruchomisz jeszcze raz ten sam program, to informacja o czasie będzie identyczna. `__TIME__` i `__DATE__` zawierają bowiem informacje, które dotyczą momentu czasowego samej kompilacji. W kompilacji zostają one tam zamrożone na zawsze.

Jeśli zaś chodzi Ci o wypisanie na ekranie **bieżącej** *daty, godziny i minuty, to realizuje się to za pomocą standardowych funkcji bibliotecznych, takich jak* `time()`, `localtime()` *itd. Ich deklaracje zebrane są w pliku nagłówkowym* `time.h`

Dodatkowo zdefiniowana jest jeszcze nazwa

```
__cplusplus
```

Jeśli kompilujesz kompilatorem C++ to zwykle może on na nasze życzenie zachowywać się tak, jakby był kompilatorem klasycznego C. Wówczas ta nazwa nie jest zdefiniowana.

Czasem się to przydaje. Najczęściej używałem tej możliwości w okresie przejściowym, kiedy nieśmiało przerabiałem moje programy z C na C++. Nie byłem wówczas pewny i chciałem zawsze mieć możliwość wycofania się. Dlatego używałem wówczas kompilacji warunkowej, gdzie warunkiem było zdefiniowanie lub niezdefiniowanie tej nazwy.

Do czego mogą się przydać takie predefiniowane nazwy

Nie przydają się często, ale rozważmy taki przypadek

Program nasz składa się z wielu plików, nad którymi pracują różni programiści dokonując ciągłych ulepszeń. Dajmy na to, że w skład programu wchodzi moduł obsługujący radar. Dostajemy go od kolegi już w postaci binarnej, gotowej do

zlinkowania z naszymi modułami. Może się czasem okazać przydatne, by w gotowym programie wiedzieć z którą wersją części „radarowej" mamy do czynienia.

Jak to zrobić? Na przykład wymagając od kolegi by umieścił w jego pliku funkcję `wersja_radaru()`, która na ekranie wypisze tekst informujący o numerze wersji. Przy okazji modyfikacji swojego modułu nasz kolega powinien zawsze zmienić treść tego tekstu – zatem na ekranie pojawiał się będzie nowy opis wersji. Niestety nasz kolega jest niechluj i mimo dokonania modyfikacji programu – często nie uaktualnia tekstu informującego o wersji.

Co robić? Jest wyjście. Wersję poznać możemy np. po nazwie pliku, w którym dane funkcje („radarowe") są umieszczone, a także po momencie kompilacji tego pliku.

Wystarczy, by kolega piszący ten moduł, umieścił w nim funkcję

```
void wersja()
{
     cout << cessna: " << __FILE__ << " "
          << __DATE__ << " " << __TIME__ << endl ;
}
```

Dzięki temu ile razy tę funkcję wywołamy, na ekranie pojawi się

```
cessna: crs71.c MAY 10 1992 15:03:47
```

Ważne, że od tej pory ta informacja nie jest zależna od dobrej woli kolegi piszącego ten plik. Bez względu na jego niechlujstwo zawsze mamy ślad w postaci prawdziwej nazwy jego pliku i tego, kiedy on ten plik kompilował.

7 Tablice

Jeśli masz do czynienia z grupą zmiennych (ogólniej mówiąc - obiektów), to możesz z nich zrobić tablicę. Tablica to ciąg obiektów tego samego typu, które zajmują ciągły obszar w pamięci.
Korzyść z tego jest taka, że zamiast nazywania każdej ze zmiennych osobno, wystarczy powiedzieć: odnoszę się do n-tego elementu tablicy.

Tablice są typem pochodnym. Znaczy to po prostu, że bierze się jakiś typ – dajmy na to `int`, i z elementów takiego typu buduje się tablicę. Jest to wtedy tablica elementów typu `int`.

Jeśli chcemy mieć 20 zmiennych typu `int`, to można z nich zbudować tablicę

```
int a[20] ;
```

Ta definicja rezerwuje w pamięci miejsce dla 20 liczb typu `int`.

Rozmiar tak definiowanej tablicy musi być stałą, znaną już w trakcie kompilacji. Kompilator bowiem musi już wtedy wiedzieć ile miejsca ma zarezerwować na daną tablicę.

Rozmiar ten nie może być więc na przykład ustalony dopiero w trakcie pracy programu.

```
cout << "Jaki chcesz rozmiar tablicy ? " ;
int rrr ;
cin >> rrr ;

int a[rrr] ;                          // Błąd !!!
```

Jest to błąd, bo wartość `rrr` nie jest jeszcze w czasie kompilacji znana. Znana jest dopiero w trakcie pracy programu.

(Jeśli rzeczywiście zachodzi konieczność takiej deklaracji – powinieneś pomyśleć o tzw. dynamicznej alokacji tablicy – str. 186)

A oto inne przykłady – będą to definicje różnych tablic. Definicje - czyli miejsca ich narodzin w programie. Definicja jest, jak wiadomo, także deklaracją, czyli

poinformowaniem kompilatora o typie danego obiektu. Stąd dla poniższych definicji podaję w komentarzu jak czyta się deklarację takiej tablicy:

```
char zdanie[80] ;              // zdanie jest tablicą
                               // 80 elementów typu char

float numer[9] ;               // numer jest tablicą
                               // 9 elementów typu float

unsigned long kanal[8192] ;    // kanal jest tablicą 8192
                               // elementów typu unsigned long

int *wskaz[20] ;               // wskaz jest tablicą 20 elementów
                               // będacych wskaźnikami (adresami)
                               // jakichś obiektów typu int
```

O adresach jeszcze dokładniej nie mówiliśmy, ale ten ostatni przykład przytoczyłem po to, by pokazać z jak różnych typów można zbudować tablice.

Tablice można tworzyć z:

- typów fundamentalnych (z wyjątkiem `void`),
- typów wyliczeniowych (`enum`),
- wskaźników,
- innych tablic ;

a także (o czym dowiesz się w przyszłości):

- z obiektów typu zdefiniowanego przez użytkownika (czyli klasy),
- ze wskaźników do pokazywania na składniki klasy.

7.1 Elementy tablicy

Na początku zajmiemy się prostymi tablicami tworzonymi z typów fundamentalnych.

Jeśli zdefiniujemy sobie taką tablicę:

```
int t[4] ;
```

to jest to tablica czterech elementów typu `int`. Poszczególne elementy tej tablicy to:

```
t[0]      t[1]      t[2]      t[3]
```

Jak widzisz

Numeracja elementów tablicy zaczyna się od zera. Jest to bardzo ważne i to trzeba zapamiętać.

W początkowym okresie będziesz się zapewne często mylił. Tym bardziej, że w niektórych językach programowania numeracja elementów tablicy zaczyna się od 1. Jeśli masz takie przyzwyczajenie, to będzie Ci trudniej.

Prawdę mówiąc nie ma specjalnej tragedii jeśli zapomnisz coś wpisać do elementu zerowego. Twoja strata. Gorzej, jeśli zapomnisz, że ostatnim, czwartym elementem naszej tablicy jest element `t[3]`, a nie element `t[4]`. Numerujemy przecież od zera! Element `t[4]` nie istnieje. Próba wpisania czegoś do elementu `t[4]` nie będzie sygnalizowana jako błąd, gdyż w językach C oraz C++ nie jest to sprawdzane.†)

Wpisanie czegoś do nieistniejącego elementu `t[4]` spowoduje zniszczenie w obszarze pamięci czegoś, co następuje bezpośrednio za tablicą. Co dokładnie jest niszczone – zależy to od implementacji. Dla zobrazowania powiem tylko, że zdarza się, iż przy takich definicjach:

```
int t[4] ;
int z ;
```

Podczas próby zapisu czegoś do nieistniejącego elementu `t[4]`

```
t[4] = 15 ;
```

zniszczona zostanie treść zmiennej `z`, bo akurat została umieszczona w pamięci bezpośrednio za tablicą `t`. Ponieważ typ zmiennej `z` zgadza się akurat z typem elementów tablicy `t`, dlatego możliwe jest też, że w zmiennej `z` znajdzie się liczba 15.

> Powtarzam jednak: to, co powiedziałem, zależne jest od implementacji. Ważny jest tutaj tylko pewnik: coś mimowolnie w pamięci zniszczyliśmy.

Należy zatem pamiętać, że tablica n-elementowa ma elementy o indeksach od 0 do n-1 (a nie do n!).

Tablicę można zapełnić treścią - na przykład za pomocą zwykłej operacji przypisania.

```
#include <iostream.h>
main()
{
int t[4] ;
     for( int i = 0 ; i < 4 ; i++)
                    t[i] = 100 * i ;                    // wpis
     cout << "Wydruk tresci tablicy : \n" ;
     for(i = 0 ; i < 4 ; i++)
     {
          cout << "Element nr : " << i
               << " ma wartosc " << t[i] << endl ;
     }
}
```

†) Nie traktuj tego jako wadę. Jeśli Ci to nie odpowiada, to w przyszłości będziesz potrafił to zmienić – definiując nowy rodzaj tablic – takich, które to sprawdzają. Ale zapłacisz za to czasem dostępu do elementu takiej tablicy. Tablica, która tego nie sprawdza – jest szybsza.

W rezultacie wykonania tego programu na ekranie pojawi się

```
Wydruk tresci tablicy :
Element nr : 0 ma wartosc 0
Element nr : 1 ma wartosc 100
Element nr : 2 ma wartosc 200
Element nr : 3 ma wartosc 300
```

Zauważ, iż zakres obu pętli `for` jest taki, że `i` przebiega od 0 do 3. To właśnie z powodów, o których wspominaliśmy.

Jeśli jesteś ciekaw, to spróbuj w drugiej pętli `for` – tej od wypisywania elementów na ekran – zmienić `i<4` na `i<=4`. Spowoduje to wypisanie na ekran piątego, nieistniejącego elementu o indeksie `t[4]`. Pobranie wartości z tego obszaru pamięci bezpośrednio za tablicą nie spowoduje katastrofy, jedynie wartość będzie bezsensowna. Tragedia byłaby dopiero wtedy, gdybyśmy chcieli coś w to miejsce wpisać.

Uwaga praktyczna:

Tu chciałbym Ci coś poradzić. Zwykle elementy tablicy czyta się lub wypisuje za pomocą pętli, na przykład pętli `for`. W naszym przypadku zapisaliśmy to tak:

```
for(i = 0 ; i < 4        ; i ++) ...    itd
```

czyli inaczej

```
for(i = 0 ; i < rozmiar  ; i++) ...
```

Mogliśmy równie dobrze napisać tak:

```
for(i = 0 ; i <= 3       ; i ++) ...
```

czyli inaczej

```
for(i= 0 ; i <= rozmiar-1  ; i++) ...
```

Podstawowa różnica jest tu, jak widać, w użyciu mocnej lub słabej nierówności. Jednak oba zapisy sprawiają, że pracujemy na elementach o indeksach 0-3, czyli oba zapisy są równoważne.

> Otóż radzę Ci byś zdecydował się na jeden typ zapisu i nigdy nie używał drugiego. Unikniesz wtedy omyłkowego adresowania elementu `t[rozmiar]` (który, jak wiemy, nie istnieje).

Osobiście preferuję ten pierwszy zapis (z silną nierównością), bo nie muszę odejmować jedynki od rozmiaru, a poza tym zapis jest krótszy.

7.2 Inicjalizacja tablic

Innym sposobem nadania wartości elementom tablicy jest inicjalizacja – nadanie wartości początkowych w momencie definicji tablicy (czyli w momencie jej narodzin).

Pamiętasz zapewne jak robiliśmy inicjalizacje w wypadku zwykłych typów

```
int liczba = 237 ;
float wspolczynnik = 0.372 ;
```

W przypadku tablicy trzeba nadać wartość początkową każdemu elementowi. Służy do tego tzw. inicjalizacja zbiorcza (ang. agregate initialization).

W naszym wypadku wygląda to tak:

```
int t[4] = { 17, 5, 4, 200 } ;
```

Jest to wygodny sposób, bo w jednej linijce załatwia inicjalizację wszystkich czterech elementów. Dzięki temu

`t[0]`	ma wartość	17
`t[1]`	ma wartość	5
`t[2]`	ma wartość	4
`t[3]`	ma wartość	200

Do znudzenia będę przypominał, że element `t[4]`, podobnie jak i element `t[107]` – nie istnieje. Gdybyś jednak w tym momencie zbiorczej inicjalizacji na liście ujętej klamrami { } umieścił o jedną lub kilka liczb za dużo, to tutaj kompilator zasygnalizuje Ci błąd. W inicjalizacji sprawdza się czy rozmiar tablicy nie jest przypadkiem przekroczony. Tylko przy inicjalizacji. Potem nie.

Możliwa jest też taka inicjalizacja naszej tablicy:

```
int t[4] = { 17, 5 };
```

Jak widać liczb jest za mało. Inicjalizacja taka spowoduje, że żądane wartości początkowe zostaną nadane tylko dwóm pierwszym elementom. Elementom `t[0]` i `t[1]`. **Pozostałe dwa elementy będą inicjalizowane zerami.**

Dla wygody istnieje też taki sposób definiowania i inicjalizacji tablic:

```
int r[] = { 2, 10, 15, 16, 3 } ;
```

Zauważ, że tutaj w kwadratowym nawiasie nie podaliśmy rozmiaru tablicy. Kompilator więc sam sobie przelicza ile to liczb podaliśmy w klamrze i w efekcie rezerwowana jest pamięć na te elementy. W naszym wypadku powstanie tablica pięcioelementowa.

7.3 Przekazywanie tablicy do funkcji

Załóżmy, że mamy tablicę z danymi pomiarowymi. Próbek pomiarowych jest dużo, bo aż 8192

```
long int widmo[8192] ;
```

Chcemy teraz napisać sobie funkcję, która treść każdego elementu tej tablicy pomnoży przez 3. Jak to pomnożyć, oczywiście wiesz – wystarczy napisać pętlę mnożącą przez 3 – po kolei wszystkie elementy tej tablicy od 0 do 8191. To jest proste. Jak jednak przesłać do funkcji tablicę?

Pamiętasz jak mówiliśmy o przesyłaniu argumentów do funkcji? Zwykle przesyła się przez wartość, czyli fotografowany jest każdy argument i jego zdjęcie

(kopia) przesyłana jest do funkcji. Tablicy jednak nie da się przesłać przez wartość. Można tak przesłać pojedyncze jej elementy, ale nie całość. To nie wynika z niedoskonałości języka. Po prostu - czy wyobrażasz sobie funkcję, która w wywołaniu dostaje 8192 argumenty? To tak, jakby fotografować 8192 razy. Samo pojedyncze wywołanie takiej funkcji trwałoby pewien znaczący czas, nie mówiąc już o trudnościach w realizacji takiego przesłania.

Zatem zasada jest taka, że:

> tablice przesyła się podając funkcji tylko adres początku tej tablicy.

Jeśli mamy funkcję

```
void funkcja (float ttt[] );
```

która spodziewa się jako argumentu: tablicy liczb typu `float`, to taką funkcję wywołujemy na przykład tak:

```
float tablica[] = { 7, 8.1, 4, 4.12 };

funkcja(tablica);                    // wywołanie funkcji
```

Przy nazwie tablicy w wywołaniu funkcji nie widzisz żadnych nawiasów kwadratowych.

Zapamiętaj sobie na zawsze (a najlepiej napisz sobie to na kartce i przylep nad biurkiem), że:
W języku C++ (tak jak i w C):

NAZWA TABLICY jest równocześnie ADRESEM ZEROWEGO JEJ ELEMENTU

Nie żartuję. Rzeczywiście wykuj to zdanie na pamięć, bo bardzo Ci to pomoże w swobodnym poruszaniu się po królestwie C++.

Jest jeszcze coś równie sympatycznego. Mianowicie wyrażenie

```
tablica + 3
```

jest adresem tego miejsca w pamięci, gdzie tkwi element o indeksie 3. Element o indeksie 3 to inaczej element

```
tablica[3]
```

adres takiego elementu to

```
&tablica[3]
```

Znak & (ampersand) jest jednoargumentowym operatorem oznaczającym uzyskiwanie adresu danego obiektu.

> Uwaga: nie mylmy tego operatora z dwuargumentowym operatorem & oznaczającym bitowy iloczyn logiczny.

A zatem poniższe dwa zapisy (wyrażenia) są równoważne

```
tablica + 3                &tablica[3]
```

Można je stosować wymiennie, co kto lubi.

Wróćmy jednak do rzeczy. W naszym wywołaniu funkcji napisaliśmy samą nazwę tablicy (bez klamer) więc do funkcji przesyłamy adres. Oto przykład programu, w którym do funkcji jako jeden z argumentów wysyłana jest tablica.

```
#include <iostream.h>
void potrojenie(int ile, long tablica[]);                    // ❶
/*****************************************************/
main()
{
const int rozmiar = 8192 ;                                   // ❷
long widmo[rozmiar] ;                                        // ❸

     // ----------nadanie wartości początkowej
     for(int i = 0 ; i < rozmiar ; i ++)
     {
          widmo[i] = i ;                                     // ❹
          if(i < 6)        // pokazanie pierwszych sześciu
               cout << "i= " << i << ") " << widmo[i]
               << endl ;
     }
     // ----------uwaga, wywołujemy funkcję !
     potrojenie(rozmiar, widmo) ;                            // ❺
     cout << "Po wywolaniu funkcji \n" ;
     for(i = 0 ; i < 4 ; i ++)
     {                                                       // ❻
          cout << "i= " << i << ") " << widmo[i] << endl;
     }
}
/*****************************************************/
void potrojenie (int ile, long t[])                          // ❼
{
     for(int i = 0 ; i < ile ; i++)
     {
          t[i] *= 3 ;                                        // ❽
     }
}
/*****************************************************/
```

W wyniku wykonania tego programu na ekranie pojawi się

```
i= 0) 0
i= 1) 1
i= 2) 2
i= 3) 3
i= 4) 4
i= 5) 5
Po wywolaniu funkcji
i= 0) 0
i= 1) 3
i= 2) 6
i= 3) 9
```

❶ Deklaracja funkcji, do której jako argument wysyła się tablicę. Zauważ jak deklaruje się argument formalny. Jest to jakby definicja tablicy typu `long`

o nieznanym rozmiarze. W związku z tym funkcja ta będzie się nadawała do pracy na dowolnej tablicy typu `long`, której elementy zamierza się potroić. Wewnątrz tej funkcji potrzebna jest nam także liczba elementów tablicy, których wartość liczbowa ma ulec potrojeniu. Dlatego wysyłamy sobie tę wartość jako argument. Musimy to zrobić, gdyż na podstawie nazwy tablicy (czyli adresu) nie da się określić ile dana tablica ma elementów.

❷ Rozmiar tablicy `widmo` definiujemy sobie w programie jako obiekt typu `int` z przydomkiem `const` – rozmiaru tej tablicy nie będziemy przecież w trakcie programu zmieniać. Dlaczego słowo `const` jest tu konieczne – okaże się za chwilę.

❸ Oto definicja tablicy. Jej rozmiar jest określony przez liczbę schowaną w powyższym obiekcie typu `int` z przydomkiem `const`. To `const` sprawia, że kompilator jest pewien stałości tej liczby już na etapie kompilacji. Jak pamiętamy rozmiar tak definiowanej tablicy musi być znany już na etapie kompilacji programu.

❹ Po definicji tablicy następuje pętla nadająca jej elementom wartości początkowe równe kolejnym liczbom naturalnym. Pierwsze 6 elementów tablicy wypisujemy na ekran.

❺ Wywołanie funkcji. Jako argumenty wysyłamy liczbę elementów, których zawartość należy potroić, oraz nazwę tablicy, na której ta operacja potrojenia ma się odbyć. Przypominam, że jeśli wysyłamy do funkcji adres tablicy, to tak, jakbyśmy wysyłali adres zerowego jej elementu.

❻ Po powrocie z funkcji wypisujemy wartości tych elementów na ekranie.

Co się dzieje wewnątrz funkcji:

❼ Odbieramy tam adres początku tablicy i służy on do zbudowania wewnątrz funkcji takiego aparatu obsługi tablicy `t[]`, że odniesienie się do elementu `t[3]` jest dokładnie tym samym, co odniesienie się do elementu `tablica[3]`. Innymi słowy nie pracujemy tu na żadnej kopii tablicy, tylko na oryginale.

Masz rację jeśli myślisz, że mętnie to tłumaczę. Niestety na razie nie mogę powiedzieć całej prawdy. Wszystko stanie się jasne w następnym rozdziale, kiedy to porozmawiamy o wskaźnikach.

❽ W naszej funkcji `potrojenie` przebiegamy po wszystkich elementach tablicy `t[]` z przedziału 0 - 8191 i mnożymy przez 3.
Przypominam że instrukcja

```
tab[i] *= 3 ;
```

to inaczej to samo co

```
tab[i] = tab[i] * 3 ;
```

Do funkcji przysłaliśmy argument określający liczbę elementów tablicy, na których należy przeprowadzić operację potrojenia. Musieliśmy to zrobić dlatego, że jedyne co wewnątrz funkcji wiadomo na temat przysłanej tablicy to to, jaki jest adres jej początku i to, iż jest ona typu `long`. Nic więcej. W szczególności nie wiadomo jaki ma rozmiar ta tablica.

Wysyłanie tylko jednego elementu

Do funkcji, której argumentem formalnym jest typ `int`

```
void fff(int x);
```

można wysłać argument będący jakimś elementem tablicy typu `int`. Nie jest to przesłanie tablicy. Odbywa się to identycznie, jak w wypadku zwykłej zmiennej `int`.

```
int m ;
int tabl[100] ;

     fff(m);                    // wysłanie do funkcji zwykłej zmiennej
     fff(tabl[38]) ;            // wysłanie do funkcji elementu nr 38
```

Takie wysłanie jednego elementu odbywa się oczywiście przez wartość.

Łatwo to zapamiętać tak:
Otóż wartością wyrażenia `tabl[38]` jest **liczba** (zapisana w tym elemencie tablicy). Natomiast wartością wyrażenia `tabl` jest **adres** tej tablicy. To dlatego mechanizm przesłania jest inny.

To (prawie) wszystko, co trzeba wiedzieć na temat tablic. Myślę, że nic w tym trudnego, bo z tablicami zetknąłeś się w innych językach programowania. Właściwie więc można by ten rozdział tutaj skończyć, gdyby nie pewien bardzo pożyteczny rodzaj tablic: tablice znakowe.

7.4 Tablice znakowe

Specjalnym rodzajem tablic są tablice do przechowywania znaków (np. liter). Dlatego też tablicom tym przyjrzymy się bliżej.

```
char zdanie[80] ;
```

Ta definicja określa, że `zdanie` jest tablicą 80 elementów będących znakami. W tablicy tej można umieścić tekst, dzięki temu, że każdy z jej elementów nadaje się do przechowywania reprezentacji liczbowej znaków alfanumerycznych. Na przykład – znaków zakodowanych kodem ASCII. Kod taki, jak zapewne wiesz, jest jednym ze sposobów kodowania liter, które w komputerze muszą być przecież przechowywane w postaci liczb. Jest kilka sposobów kodowania liter, my jednak zawsze mówić będziemy o kodzie ASCII.

Teksty w tablicach znakowych zwykło się przechowywać tak, że po ciągu liter (a właściwie ich kodów liczbowych) następuje znak o kodzie 0. Ten znak zwany jest NULL. Znak ten jest po to, by oznaczyć gdzie kończy się ciąg liter. Na taki ciąg liter zakończony znakiem NULL mówimy *string*.

Naszą tablicę `zdanie` można już w trakcie definicji zainicjalizować

```
char zadnie[80] = { "lot" } ;
```

Zauważ, że odbywa się to przez napisanie tekstu ujętego w cudzysłów.

W rezultacie w poszczególnych komórkach tablicy `zdanie` znajdą się następujące znaki

l	o	t	NULL							
0	1	2	3	4	5	...	...	...	78	79

Jak widzisz wpisane zostały trzy litery, a za nimi znak o kodzie NULL kończący string. Dalsze elementy tablicy nas nie interesują. Warto jednak pamiętać, że – w myśl omówionych niedawno zasad inicjalizacji zbiorczej tablic – nie wymienione elementy inicjalizuje się do końca zerami.

> Znak NULL został automatycznie dopisany po ostatniej literze `t` dzięki temu, że inicjalizowaliśmy tablicę ciągiem znaków ograniczonym cudzysłowem.

Jest to po naszej myśli, bo w językach C i C++ wszelkie funkcje biblioteczne pracujące na stringach opierają się na założeniu, że koniec stringu oznaczony jest znakiem NULL.

Na przykład:

> *Jeśli chcemy jakiś string wypisać na ekran, to wywołujemy standardową funkcję biblioteczną (`puts` – od: put string) i jako argument przekazujemy jej adres początku stringu. Funkcja ta będzie sukcesywnie wypisywała na ekran znaki zawarte w kolejnych komórkach pamięci począwszy od adresu podanego jako początek stringu. Akcja zakończy się dopiero po natknięciu się na komórkę ze znakiem NULL, czyli inaczej mówiąc z zapisanym tam bajtem 0.*

Jest też inny sposób inicjalizacji tablicy znaków

```
char zdanie[80] = { 'l', 'o', 't' };
```

Zauważ, że w klamrze pojawiły się pojedyncze litery ograniczone apostrofami. Zapis taki jest nie tylko równoważny trzem instrukcjom:

```
zdanie[0] = 'l' ;
zdanie[1] = 'o' ;
zdanie[2] = 't' ;
```

Ponieważ nie było tu cudzysłowu, więc kompilator nie dokończył tego znakiem NULL umieszczanym poprzednio w elemencie `zdanie[3]`. Zatem sprawa wydaje się ryzykowna – po ciągu liter nie nastąpił znak kończący string.

A jednak przypomnij sobie co mówiłem o inicjalizacji zbiorczej – czyli takiej, gdzie wartości początkowe umieszczone są w klamrze `{ }` . Otóż jeśli wartości początkowych jest mniej niż elementów tablicy, to reszta jest inicjalizowana zerami. Czyli pozostałe elementy tablicy `zdanie`, aż do elementu o indeksie 79, zostaną inicjalizowane zerami. W szczególności w elemencie `zdanie[3]` też znajdzie się zero. A teraz uważaj: znak NULL ma przecież też wartość 0 – zatem nie musimy się martwić, string w tablicy `zdanie` jest poprawnie zakończony.

Pułapka

Nie zawsze jednak życie jest tak piękne. Oto chcieliśmy być sprytniejsi i napisaliśmy taką definicję:

```
char zdanie[] = { 'l', 'o', 't' };
```

Pamiętamy bowiem, że jeśli nie podamy rozmiaru tablicy, to kompilator przeliczy sobie liczbę inicjalizatorów i taką zarezerwuje tablicę. W naszym przykładzie doliczy się trzech elementów. Zostanie więc zarezerwowane tablica 3 elementowa. W poszczególnych jej elementach znajdą się znaki `'l' 'o' 't'`, ale znaku NULL tam nie będzie. Nie było cudzysłowu więc kompilator nie sądził, że te znaki mają się składać na string. Dopisania reszty zerami nie będzie, bo żadnej reszty nie ma. Tablica ma tylko trzy elementy. Podkreślam jednak – nie jest to błąd. Możliwe, że litery te nie mają być używane jak ciąg znaków, (czyli string), lecz po prostu są to luźne litery, do innych celów. Traktować jako string ich jednak nie można.

Zwróć uwagę na tę deklarację:

```
char zadnie[] = { "lot" } ;
```

Tutaj kompilator obliczając ile elementów ma zarezerwować na tablicę doliczy się trzech liter, ale z faktu, że są one ujęte w cudzysłów wywnioskuje, że mają one być używane jako string, więc zarezerwuje jeszcze jeden dodatkowy element na znak NULL.

Jeśli nie wierzysz to zobacz:

```
#include <iostream.h>
main()
{
char napis1[] = { "Nocny lot"} ;
char napis2[] = {  'N', 'o', 'c', 'n', 'y',
                   ' ', 'l', 'o', 't' } ;
     cout << "rozmiar tablicy pierwszej : "
          << sizeof(napis1) << endl ;

     cout << "rozmiar tablicy drugiej : "
          << sizeof(napis2) << endl ;
}
```

W rezultacie wykonania tego programu na ekranie pojawi się

```
rozmiar tablicy pierwszej : 10
rozmiar tablicy drugiej : 9
```

Często używa się pojęcia **długość stringu** - jest to ilość liter należących do tego stringu. Pamiętajmy, że **rozmiar stringu** jest większy o 1, czyli o znak NULL.

A teraz pytanie kontrolne. Mamy takie dwie definicje tablic. Co o nich sądzisz – czy są poprawne?

```
char t1[1] = { "a" } ;
char t2[1] = { 'a' } ;
```

Nie, nie są. Pierwsza jest błędna. Tablica jest tu jednoelementowa, a chcemy wpisać do niej string złożony z litery `'a'` i znaku NULL. Na to potrzeba dwóch elementów tablicy. Zatem błąd.

W drugiej definicji inicjalizacja polega na wpisaniu do tablicy tylko jednej litery, a więc jest poprawna.

Przyglądaliśmy się przed chwilą jak wpisuje się stringi do tablic. Niestety: w podany sposób wpisywać można do tablicy znakowej tekst tylko w czasie inicjalizacji. Potem, próba wpisania do niej czegoś tym sposobem

```
zdanie[80] = "nocny lot";          // Błąd !
zdanie = "nocny lot" ;             // także błąd !
```

jest niepoprawna.

Jak zatem wpisywać string do tablic już istniejących?

Sprawa jest prosta. Trzeba napisać sobie funkcję, która się tym zajmie i podany string, litera po literze umieści w danej tablicy.

Zadanie to jest tak częste, że w standardowej bibliotece jest kilka funkcji realizujących to w różnych wariantach.

Przyjrzyjmy się takiej funkcji. Jako argumenty otrzymuje ona dwie tablice – tę, w której jest string źródłowy i tę, gdzie ma on później zostać skopiowany.

> *O tym, że tablica może być argumentem funkcji – wiemy już z poprzedniego paragrafu. Naprawdę do funkcji przesyłany jest tylko adres początku tablicy, a nie wszystkie jej elementy (których mogły by być tysiące).*

Jak ma wyglądać sam proces kopiowania? Jest to bardzo proste dzięki naszej zapobiegliwości. Kopiuje się znak po znaku elementy z tablicy źródłowej do tablicy docelowej tak długo, aż nie napotka się na znak NULL. Ten znak NULL, także należy przekopiować – jeśli wynik ma być poprawnym stringiem.

Spróbujmy zatem to napisać. Oczywiście posłużymy się pętlą.

```
void strcpy(char cel[], char zrodlo[])          // ❶
{
     for(int i = 0 ; ; i++)                     // ❷
     {
          cel[i] = zrodlo[i];                   // ❸
          if(cel[i] == NULL)break ;             // ❹
     }
}
```

❶ Jest to, jak czytamy, funkcja przyjmująca jako argumenty dwie tablice elementów typu `char` – czyli dwie tablice znakowe. Typem zwracanym jest typ `void`. Uczciwie przyznam, że mogłem wymyślić to lepiej, ale na razie niech będzie jak jest.

❷ Do przebiegnięcia po poszczególnych elementach tablicy służy nam pętla `for`. Pętlą ta zaczyna się od i = 0, bo jak wiadomo numeracja elementów tablic zaczyna się od 0. Dalej w instrukcji `for` widzimy puste miejsce – bo nie wiemy ile elementów tablicy będziemy przepisywać. Puste miejsce oznacza tutaj, że pętla jest nieskończona.

❸ To jest właściwa akcja przepisania poszczególnego znaku z tablicy `zrodlo` do tablicy `cel`

❹ Sprawdzenie czy właśnie przepisany znak nie jest czasem znakiem NULL. Jeśli tak, to pętlę przerywa się instrukcją `break`. Jeśli nie, to wykonujemy następny obieg pętli. Instrukcją `i++` zwiększa się zmienna `i`, którą używamy do indeksowania elementów tablicy, po czym kopiujemy dalej.

Inne sposoby zrobienia tego samego

A oto jak tę samą funkcję można zrealizować używając pętli `do-while`:

```
void strcpy(char cel[], char zrodlo[])
{
int i = 0 ;
     do{
          cel[i] = zrodlo[i];          // kopiowanie
     }while(cel[i++] != NULL) ;        // sprawdzenie i przesunięcie
}
```

W skrócie można treść tej instrukcji wypowiedzieć tak: Dotąd kopiuj elementy z tablicy do tablicy, dopóki (ang: while) – właśnie skopiowany element jest różny od NULL.

Przy okazji wykonuje się przesunięcia indeksu tablicy metodą postinkrementacji. Wyrażenie porównujące

```
cel[i] != NULL
```

jest jeszcze ze starą wartością `i`. Bezpośrednio po tym następuje inkrementacja czyli zwiększenie indeksu o 1 (**post**-inkrementacja).

Pamiętasz może, jak przy okazji omawiania operatora = mówiłem, że wartością wyrażenia przypisania (podstawienia) jest wartość będąca przedmiotem przypisania. Inaczej mówiąc wyrażenie

```
(i = 15)
```

nie tylko, że wykonuje przypisanie, ale jeszcze samo jako całość ma wartość 15. Podobnie wyrażenie

```
(cel[i] = zrodlo[i])
```

ma wartość równą kodowi ASCII kopiowanego właśnie znaku. Korzystając z tego możemy napisać naszą funkcję tak:

```
void strcpy(char cel[], char zrodlo[])
{
int i = 0 ;
     while(cel[i] = zrodlo[i])
     {
          i++ ;
     }
}
```

Przeczytać to można tak: Jeśli wartość wyrażenia kopiowania

```
(cel[i] = zrodlo[i])
```

jest różna od zera (czyli NULL) to inkrementuj indeks `i`. Teraz widzisz dlaczego znak NULL kończący string ma wartość właśnie 0

Sprytne, prawda? Nie mówimy:

- musisz skoczyć po gazetę i potem rozwiąż w niej krzyżówkę,

tylko

- jeśli gazeta, którą przyniosłeś ma jakiś tytuł inny niż NULL, to możesz rozwiązać w niej krzyżówkę.

Chcąc sprawdzić jaka jest wartość wyrażenia komputer musi to wyrażenie „obliczyć" , czyli w naszym wypadku wykonać kopiowanie znaku z tablicy `zrodlo` do tablicy `cel`. Za jednym zamachem robimy więc kilka rzeczy.

Dwie uwagi (tytułem przestrogi)

Po pierwsze:
Do wnętrza instrukcji kopiowania nie wstawiłem postinkrementacji indeksu `i`. To dlatego, że w wyrażeniu

```
cel[i] = zrodlo[i++] ;
```

nie mam pewności kiedy naprawdę nastąpiłaby ta postinkrementacja. Jest ryzyko, że na przykład po wyjęciu znaku z tablicy `zrodlo`, a przed wsadzeniem go do tablicy `cel`. Wtedy kopiowanie odbyłoby się na wzór takiej instrukcji:

```
cel[4] = zrodlo[3] ;
```

Jak to jest? Otóż kompilator nie gwarantuje kolejności obliczania zmiennej `i`, dlatego bezpieczniej inkrementację zrobić w głębi pętli `while`.

Druga uwaga:
Niektóre bardziej troskliwe kompilatory, gdy zobaczą wyrażenie

```
while(cel[i] = zrodlo[i])...
```

czyli inaczej

```
while(a = b)...
```

nie docenią naszego sprytu i pomyślą, że prawdopodobnie chodzi nam o porównanie dwóch elementów. Czyli, że mówimy: wykonuj daną pętlę dopóki `a` równa się `b`.

Kompilator taki przypuszcza, że pomyliliśmy porównanie (==) z przypisaniem (=) czyli, że zamiast dwóch znaków == wstawiliśmy tylko jeden =. Nie sygnalizuje błędu, jedynie ostrzega nas.

> Trzeba jednak przyznać, że najczęściej taki kompilator ma rację. Chwyt, który tutaj zastosowaliśmy nie stosuje się tak często. Za to bardzo często zapomina się przy operacji porównania o podwójnym znaku równości. Dlatego lepiej niech kompilator nam patrzy na ręce.

Wróćmy jednak do naszych baranów.

Tym sposobem napisaliśmy funkcję `strcpy`, która może nam się przydać bardzo często. Przykładowo spójrz na taki fragment programu:

```
//........
char start[] = { "taki sobie zwykly tekst" } ;
char meta[80] ;

strcpy(meta, start) ;
cout << meta ;
```

Fragment ten wykonuje kopiowanie stringu z tablicy `start` do tablicy `meta`. Następnie wypisuje się na ekran nową zawartość tej tablicy.

Przyjrzyjmy się definicji tablicy `start`. Nie ma w niej rozmiaru, zatem kompilator przeliczy sobie wszystkie znaki w tekście ograniczonym cudzysłowem, następnie doda jeden (na kończący znak NULL) i tyle komórek pamięci zarezerwuje na naszą tablicę.

Z tablicą `meta` jest inaczej. Tutaj nie inicjalizujemy jej, więc kompilator chce znać żądany rozmiar. Podajemy mu np. 80, bo spodziewamy się, że nasze teksty ładowane do tej tablicy nigdy nie będą dłuższe niż 79 znaków.

Ostrożnie !

Co by było, gdybyśmy w definicji tablicy `meta` zamiast `[80]` podali rozmiar `[5]` i wykonali program?

Pamiętasz zapewne, że w naszej funkcji `strcpy` kopiowanie odbywa się na oślep. Poszczególne znaki będą brane z tablicy `start` i wpisywane do kolejnych elementów do tablicy `meta`. Tak będzie aż do końca ciągu znaków, czyli do kończącego treść tablicy `start` znaku NULL. Jeśli tablica `meta` jest za krótka, bo ma tylko 5 elementów (ostatni to `meta[4]`) – to mimo wszystko dalej będzie odbywało się wpisywanie do nieistniejących elementów

```
meta[5], meta[6], ....
```

i tak dalej dopóki string ze `startu` się nie skończy.

Wiesz oczywiście co takie wpisywanie do nieistniejących elementów znaczy: Mimowolnie będą niszczone komórki pamięci znajdujące się zaraz za naszą tablicą `meta`. Tragedia. Tym większa tragedia, że nie musisz się zorientować od razu. Zależnie od tego co tam akurat było - błąd może ujawnić się o wiele później. Tym trudniej jest to wykryć.

Czy nie można zabezpieczyć się przed takimi ewentualnościami? Ostatecznie można, ale jest problem: otóż do funkcji `strcpy` przesyłamy tylko adres tablicy. Funkcja z deklaracji argumentów formalnych wie tylko, że ma do czynienia z tablicami znakowymi. Nie wie nic więcej. W szczególności nie wie jaki jest rozmiar tablicy. Nie można wewnątrz funkcji `strcpy` wstawić sobie takiego wyrażenia:

```
sizeof(cel)
```

Jeśli zatem chcemy się przed pomyłkami zabezpieczyć, to musimy przysłać do funkcji argument będący rozmiarem tablicy, albo – bardziej uniwersalnie – argument mówiący o tym, ile maksymalnie znaków życzymy sobie przekopiować.

Na przykład życzymy sobie przekopiować tylko 5 znaków. Jeśli string przeznaczony do kopiowania będzie bardzo krótki, np 3 literowy: `"ach"` to przekopiuje się cały. Jeśli będzie długi: `"Niesamowicie dlugie zdanie"`, to przekopiuje się tylko początek `"Niesa"`. Na końcu musi się oczywiście znaleźć bajt zerowy (NULL).

Moglibyśmy się teraz rzucić do pisania i wymyślić funkcję `strncpy`. Litera `n` w środku nazwy miałaby nam przypominać, że chodzi po przekopiowanie maksymalnie `n` znaków, (czyli tylko kawałka stringu) bez dodawania na końcu znaku NULL. Moglibyśmy tak zrobić, gdyby nie to, że te wszystkie wspaniałe funkcje dawno zostały napisane i zawarte są w standardowej bibliotece. Aby z nich skorzystać wystarczy włączyć do swojego programu plik nagłówkowy z ich deklaracjami. Plik nagłówkowy tej części standardowej biblioteki, która odpowiada za pracę ze stringami nazywa się `string.h` Potem, na etapie linkowania dołączana jest (najczęściej automatycznie) ta część biblioteki.

```
#include <iostream.h>
#include <string.h>
main()
{
char tekst[] = { "Uwaga, tarcza zostala przepalona !"} ;
char komunikat[120] ;

     strcpy(komunikat, tekst);
     cout << komunikat << endl ;

     strncpy(komunikat, "1234567890abcdef" , 9 );    // ❶
     cout << komunikat ;

     strcpy(komunikat, "--A ku-ku --!" ) ;
     cout << "\nNa koniec : "
          << komunikat << endl ;
}
```

Wykonanie tego programu objawi się na ekranie jako:

```
Uwaga, tarcza zostala przepalona !
123456789rcza zostala przepalona !
Na koniec : --A ku-ku --!
```

❶ Zauważ, że źródłem wcale nie musi być tablica definiowana przez nas. Może być to stała tekstowa (string) będący stałą dosłowną. Taki string przecież także musi być przechowywany gdzieś w pamięci komputera.

Dużo mówiliśmy tu na temat tablic znakowych. Nie dlatego, żeby były one jakieś specjalnie trudne, lecz dlatego, że bardzo często się ich używa. Nie ma tu w zasadzie żadnych cudów – string to po prostu znaki ustawione jeden za

drugim, a na ich końcu jest NULL, czyli bajt zerowy. To wszystko. Nazwa tego stringu (nazwa tablicy, gdzie on tkwi) jest równocześnie adresem tego miejsca w pamięci, gdzie string ten się zaczyna. Tam jest więc początek tego stringu. Koniec jest tam, gdzie jest znak NULL.

Jeśli chcemy wysłać string do funkcji, to wysyłamy tam adres jego początku, czyli samą jego nazwę (bez żadnych nawiasów kwadratowych). Dzięki temu funkcja dowiaduje się, gdzie w pamięci zaczyna się ten string. Gdzie się on kończy – funkcja może sprawdzić sobie sama szukając znaku NULL.

Do stringów jeszcze powrócimy w rozdziale o wskaźnikach.

7.5 Tablice wielowymiarowe

Tablice można tworzyć z bardzo różnych typów obiektów. Widzieliśmy już tablice z liczb całkowitych, zmiennoprzecinkowych, tablice znaków.

Mogą być także tablice, których elementami są inne tablice. Nazywamy je tablicami wielowymiarowymi. Oto przykład takiej tablicy:

```
int ddd[4][2] ;
```

Definicję tę, która jest przy okazji deklaracją czytamy tak: `ddd` jest tablicą 4 elementów, z których każdy jest dwuelementową tablicą liczb typu `int`.

Zauważ charakterystyczną notację. W innych językach często stosuje się zapis typu [i, j] – u nas jest [i][j]. Można myśleć o tym, jako o zwykłej odmienności notacji. Można też twierdzić, że zapis `ddd[i][j]` pozwala lepiej pamiętać, że `ddd` to tablica tablic. Czasem ten punkt widzenia się przydaje. Wkrótce się przekonasz.

Ewentualny błędny zapis `ddd[i, j]` kompilator zinterpretuje jako `ddd[j]`. (Przypominam jakie ma działanie operator ',' (przecinek) : wartością wyrażenia złożonego z kilku członów oddzielonych przecinkami jest wartość tego członu, który stoi najbardziej z prawej).

Poszczególne elementy naszej tablicy to:

```
ddd[0][0]      ddd[0][1],     ddd[1][0],     ddd[1][1],
ddd[2][0],     ddd[2][1],     ddd[3][0],     ddd[3][1]
```

Elementy takie umieszczane są kolejno w pamięci komputera tak, że najszybciej zmienia się najbardziej skrajny prawy indeks.

Zatem poniższa inicjalizacja zbiorcza:

```
int ddd[4][2] = { 10, 20, 30, 40, 50, 60, 70, 80 } ;
```

spowoduje, że elementom tej tablicy zostaną przypisane wartości początkowe tak, jakbyśmy to robili grupą instrukcji:

```
ddd[0][0] = 10 ;
ddd[0][1] = 20 ;
ddd[1][0] = 30 ;
ddd[1][1] = 40 ;
ddd[2][0] = 50 ;
ddd[2][1] = 60 ;
```

```
ddd[3][0] = 70 ;
ddd[3][1] = 80 ;
```

Można powiedzieć, że tablica ta ma 4 wiersze i 2 kolumny.

Oto przykład programu:

Załóżmy, że mamy nasze dane pomiarowe w postaci tablicy 8192 elementów. Takich pomiarów mamy cztery, dla czterech różnych próbek. Zapisane są te dane na dysku w postaci pliku. Uruchamiamy program i wczytujemy te dane do tablicy wielowymiarowej

```
long widmo[4][8192] ;
```

Jak to robimy, to na razie tajemnica, o szczegółach operacji z plikami porozmawiamy w osobnym rozdziale. Teraz tę akcję markujemy jedynie wywołaniem funkcji `wczytaj_dane`.

O co nam chodzi w tym programie: załóżmy, że między elementem tablicy o indeksie 500, a elementem o indeksie 540 zebrały się dane z interesującej nas reakcji. Chcemy więc zsumować wszystkie liczby zapisane od elementu o indeksie 500 do elementu o indeksie 540. Chcemy to zrobić dla każdego z czterech pomiarów

```
#include <iostream.h>
wczytaj_dane() ;
/**************************************************/
main()
{
long widmo[4][2048] ;
long suma ;
int i ;
      // tajemnicza funkcja, która wczyta z dysku cztery
      // zestawy wyników pomiarowych i dane te umieści w tablicy widmo
      wczytaj_dane() ;

      cout << "Jaki przedzial widma ma byc integrowany ?\n"
            << "podaj dwie liczby : " ;

      int pocz, koniec ;
      cin >> pocz >> koniec ;

                              // ---- pętla po 4 różnych próbkach
      for(int pomiar = 0 ; pomiar < 4 ; pomiar ++)      // ❶
      {
            suma = 0 ;
                              // ---- pętla integrująca dane jednej próbki
            for(i = pocz ; i <= koniec ; i++)           // ❷
            {
                  suma += widmo[pomiar][i] ;            // ❸
            }

            cout << "\nW probce "<< pomiar
                  << " miedzy kanalami "
                  << pocz << " a " << koniec <<" jest "
                  << suma << " zliczen" ;               // ❹
      }                       // koniec pętli po 4 pomiarach
```

```
}
/*************************************************/
wczytaj_dane()
{
        //... wczytywanie danych z dysku
}
```

Po wykonaniu programu (zależnie od odpowiedzi na pytania) możemy na ekranie otrzymać na przykład następujący tekst:

```
Jaki przedzial widma ma byc integrowany ?
podaj dwie liczby : 50 75
W probce 0 miedzy kanalami 50 a 75 jest 493 zliczen
W probce 1 miedzy kanalami 50 a 75 jest 392 zliczen
W probce 2 miedzy kanalami 50 a 75 jest 300 zliczen
W probce 3 miedzy kanalami 50 a 75 jest 172 zliczen
```

Uwagi

❶ Ponieważ w programie mamy wykonać identyczną akcję na 4 różnych próbkach dlatego wygodnie posłużyć się pętlą.

❷ W danych pomiarowych danej próbki sumujemy kolejne elementy (sąsiednie kanały). O tym, która część `widma` ma zostać poddana takiej operacji, zadecydowaliśmy odpowiadając na pytania o początek i koniec.

❸ Jest to moment sumowania elementów tablicy. Jeśli jeszcze nie oswoiłeś się z operatorem += to może wolisz taki, równoważny tamtemu zapis

```
suma = suma + widmo[pomiar][i] ;
```

❹ To jest moment wypisania wyniku na ekran.

Jak widzisz nie ma w tablicach dwu- i wielowymiarowych niczego nadzwyczajnego ani trudnego. Domyślasz się chyba też jak komputer oblicza sobie, gdzie w pamięci jest dany element tablicy.

Jak już wiemy, w pamięci elementy ustawiane są kolejno tak, że najczęściej zmienia się najbardziej skrajny indeks. Znaczy to, że w naszej tablicy

```
long widmo[4][2048] ;
```

element `widmo[1][3]` leży w stosunku do początku tablicy o tyle elementów dalej

```
(1 * 2048) +3
```

ogólniej, element `widmo[i][j]` z tablicy o liczbie kolumn 2048 leży o

```
(i * 2048) + j
```

elementów dalej niż początkowy. To bardzo ważny wniosek. Jak widzisz do orientacji w naszej tablicy kompilator potrzebuje znać liczbę jej kolumn, natomiast wcale nie musi używać liczby wierszy.

Ja bym sobie w takich stwierdzeniach nawet nie zawracał głowy wierszami i kolumnami. Można to prościej zapamiętać tak:

> Ten wymiar, który w definicji tablicy jest najbardziej z lewej

```
long widmo[4][2048] ;          // u nas to liczba 4
```

> nie jest potrzebny kompilatorowi do obliczania położenia (adresu) elementów tablicy.

Zwykle o takich sprawach nie musimy w ogóle wiedzieć, każemy zrobić operację na wybranym elemencie `[i][j]`, a jak sobie to komputer załatwia – to już jego sprawa. Jeśli jednak jest się tego świadomym, to łatwiej zrozumieć jak przekazuje się do funkcji tablice wielowymiarowe.

7.5.1 Przesyłanie tablic wielowymiarowych do funkcji

Ten tytuł może trochę mylić. Tablice jako całość nigdy nie są przekazywane. To między innymi dlatego, że w wypadku takiej tablicy:

```
long widmo[4][8192] ;
```

musielibyśmy do funkcji przesłać 4 * 8192 = ok. 32 tysiące liczb (a każda jest np. 4 bajtowa).

Wiemy już, że tablice przekazuje się w ten sposób, iż przesyłamy do funkcji tylko adres początku.

Jak pamiętamy – nazwa tablicy jest równocześnie adresem jej początku - więc do hipotetycznej funkcji `fun` przesyłamy ją tak:

```
fun(widmo) ;
```

A teraz przyjrzyjmy się jak ją (tablicę) odbieramy w funkcji.

Zanim jednak to pokażemy uświadomijmy sobie co funkcja musi o naszej tablicy wiedzieć

- po pierwsze – oczywiście typ elementów tej tablicy (czy `float`, czy `char` itd),
- po drugie – aby funkcja mogła łatwo obliczać sobie, gdzie w pamięci jest określony element tablicy – musi znać liczbę kolumn w tej tablicy. (Na przykład: gdzie w stosunku do początku tablicy jest element [1][15]).

Deklaracja takiej funkcji wygląda tak:

```
void fun(long t[][8192] ) ;
```

podana jest w niej liczba kolumn tej tablicy, którą funkcja będzie otrzymywać. Oczywiście deklaracja

```
void fun(long t[4][8192]) ;
```

jest tak samo dobra, ale z liczby wierszy (tutaj 4) funkcja nie korzysta.

Więcej wymiarów

Przez analogię łatwo chyba się domyślisz, że w wypadku funkcji otrzymującej tablicę trójwymiarową deklaracja takiej funkcji może wyglądać na przykład tak:

```
void fun_trzy(long m[][20][100] );
```

a czterowymiarowej

```
void fun_cztery(long x[][2][12][365]) ;
```

Tylko lewego skrajnego rozmiaru nie trzeba deklarować, bo i tak nie bierze on udziału w obliczaniu pozycji żądanego elementu w pamięci.

Cały ten paragraf nie służy po to, by Cię zniechęcić do podawania w deklaracji tego lewego skrajnego wymiaru. Jest po to, byś wiedział dlaczego deklaracja

```
void fff(long m[][][]);          // błąd
```

jest błędna, a deklaracja

```
void ggg(long z[]) ;
```

jest poprawna.

Na tym kończy się rozdział o tablicach, ale nie rozstajemy się z tą tematyką, bowiem następny rozdział mówił będzie o wskaźnikach, co mocno jest związane z tematyką tablic.

8 Wskaźniki

Zapytano kiedyś Amerykanów: „–Czy można, żyć bez Coca-Coli?". Amerykanie odpowiedzieli: „–Można, ale po co?"

Dokładnie to samo można powiedzieć o wskaźnikach. Ich istnienie nie jest konieczne, najlepszy dowód, że jest wiele języków programowania, w których wskaźników nie ma. Z drugiej jednak strony – jeśli języki C i C++ mają opinię tak sprawnych i władnych †), to jest to głównie za sprawą wskaźników.

Są ludzie, którzy nie lubią C. Moim zdaniem to dlatego, że nie czują się swobodnie w operowaniu wskaźnikami.

Jeśli doskonale znasz język C, a książkę tę czytasz, by poznać C++, to w zasadzie mógłbyś ten rozdział opuścić. Jeśli jednak wskaźniki znasz trochę gorzej, to zachęcam do przeczytania.

8.1 Wskaźniki mogą bardzo ułatwić życie

Do rzeczy: Wyobraź sobie, że masz w ręce bardzo gruby rozkład jazdy. Taki na 600 stron. Ktoś pyta Cię o pociąg z Paryża do Marsylii. Otwierasz więc rozkład jazdy i szukasz przewracając strony. Wreszcie po dłuższym czasie trafiasz na właściwą stronę i przebiegając palcem kolumny cyfr znajdujesz, to czego szukałeś. Dumnym głosem mówisz – 9:26 i zamykasz rozkład z trzaskiem.

Wtedy ten ktoś pyta: „–A następny?". Bierzesz znowu rozkład jazdy, przewracasz strony, trafiasz na właściwą, potem znowu przebiegasz kolumny i... jest! 13:50

Tylko, że teraz jesteś już sprytniejszy: po odpowiedzi na wszelki wypadek trzymasz palec wskazujący na znalezionej godzinie odjazdu. Jeśli Twój przy-

†) Chciałoby się powiedzieć po angielsku: *powerfull*

jaciel zapyta Cię teraz: „–A następny...?" - błyskawicznie przesuniesz palec o jedną kolumnę w prawo o odczytasz godzinę. Jeśli zapyta Cię „–A o której on jest w Lyonie?" wtedy przesuniesz palec kilka linijek w dół.

Podany przykład ilustruje życie ze wskaźnikami i bez. Co było wskaźnikiem w naszym wypadku? Był to Twój palec, ale nie tylko: także wiedza o tym, jak należy go przesunąć w wypadku pytania: „A następny ?"

Przetłumaczmy to na język pojęć, którymi operujemy w C++ :

Rozkład jazdy to nic innego jak olbrzymia (wielowymiarowa) tablica liczb. Godzina odjazdu pociągu z Paryża do Marsylii to po prostu element tej tablicy

```
pociag[Paryz][Marsylia][wyjazd_z_Paryza][0]
```

Ostatni indeks oznacza, o który pociąg w ciągu doby nam chodzi (w ciągu doby mogą bowiem odjeżdżać do Marsylii np. 4 takie pociągi).

Szukanie informacji o tym pociągu to obliczanie miejsca w tablicy, gdzie zapisana jest ta informacja. Pamiętasz zapewne z poprzedniego rozdziału, jak oblicza kompilator pozycję elementu w stosunku do początku tablicy. Dla tamtej dwuwymiarowej tablicy była to zależność

```
(i * 8192) + j
```

gdzie 8192 było liczbą kolumn tablicy. Tutaj, w naszym przykładzie z rozkładem jazdy, mamy dużo więcej wymiarów, więc i obliczanie będzie bardziej skomplikowane. W naszym obliczaniu będą aż 3 mnożenia, a to zajmuje trochę czasu.

Gdy wreszcie trafiliśmy w rozkładzie jazdy na właściwe miejsce, to odczytaliśmy je – jest to jakby odczytanie liczby będącej treścią elementu naszej tablicy.

Potem zamknęliśmy rozkład i wtedy nasz przyjaciel wygłosił wspomniane słowa: „A następny?"

Następny pociąg w naszej tablicy to

```
pociag[Paryz][Marsylia][wyjazd_z_Paryza][1]
```

Od nowa więc zajęliśmy się obliczaniem miejsca w pamięci, gdzie zapisana jest poszukiwana informacja. Znowu natrudziliśmy się wielce powtarzając ten proces, ale wreszcie znaleźliśmy właściwą informację i odczytaliśmy ją.

Wtedy, chcąc oszczędzić sobie dalszej ewentualnej pracy, zaznaczyliśmy sobie palcem to miejsce i dopiero wtedy przymknęliśmy książkę.

> Oznacza to – zdefiniowaliśmy wskaźnik (wskazujący palec prawej ręki) i podstawiliśmy do niego wartość początkową – adres odczytanego właśnie elementu tablicy (innymi słowy przystawiliśmy palec wskazujący tak, by pokazywał na właściwą liczbę na właściwej stronie)

Książkę przymknęliśmy, ale tak, że palec mimo wszystko tam tkwi. Kiedy nasz przyjaciel zapytałby nas o następny pociąg, od razu otworzylibyśmy rozkład na właściwej stronie. Błyskawiczny ruch palca w prawo i już wiemy.

Zapytanie o to, kiedy pociąg jest w Lyonie, jest próbą znalezienia elementu

```
pociag[Paryz][Marsylia][postoj_w_Lyonie][1]
```

Tutaj wystarczy wiedzieć ile stacji jest między Paryżem a Lyonem i o tyle linijek przesunąć palec w dół.

Jaki jest wniosek z tej dykteryjki? Taki, że posługiwanie się wskaźnikiem bardzo przyspiesza operacje na tablicach.

Może być więcej niż jeden wskaźnik pracujący na naszej tablicy. Możemy bowiem innym palcem zaznaczyć sobie informację o pociągach z Marsylii do Avignon. Jeśli nam nie wystarczy palców zorganizujemy sobie zakładki. Zakładek takich możemy mieć dowolnie dużo.

Jeśli po tym ktoś mi powie, że nie lubi wskaźników, to nie uwierzę. Mało rzeczy może tak uprościć życie.

8.2 Definiowanie wskaźników

Wskaźniki nie muszą koniecznie pokazywać na elementy tablicy. Można nimi posłużyć się wobec dowolnej zmiennej, wobec dowolnego obiektu. Zaczniemy od takich przykładów.

Przyjrzyjmy się poniższej definicji

```
int *w ;
```

Jest to definicja wskaźnika mogącego pokazywać na obiekty typu `int`.

Definicję taką odczytujemy jak zwykle od końca. Na widok gwiazdki mówimy: „jest wskaźnikiem do pokazywania na obiekty typu..."

A zatem czytamy na głos:

`w` – jest wskaźnikiem do pokazywania na obiekty typu – `int`.

Innymi słowy, w tym wskaźniku `w` możemy przechowywać adres jakiegoś obiektu typu `int`. Treścią wskaźnika jest informacja o tym, **gdzie** wskazywany obiekt się znajduje, a nie **co** w tamtym obiekcie się znajduje.

W definicji widzisz znak `'*'`, który informuje nas, że mamy tu do czynienia ze wskaźnikiem. Słowo `int` w tej definicji informuje nas, że wskaźnik ten ma służyć do pokazywania na obiekty typu `int`.

Nasz wskaźnik nazywa się `w`. Samo `w`, bez gwiazdki. Gwiazdka zaś, mówi tylko, że to coś o nazwie `w` jest wskaźnikiem. Słowo `int` mówi na jakie obiekty może on pokazywać.

Porównaj to zresztą z definicją tablicy

```
int t[5] ;
```

Tablica nazywa się `t`. Nawiasy `[]` mówią tylko, że to `t` jest tablicą. `int` – że elementów typu `int`.

Mogą być wskaźniki do obiektów różnych typów: np. `char`, `float` itd, (a nawet do obiektów, których typy sami sobie wymyślimy i zdefiniujemy).

```
char * wsk_do_znakow ;
float * wsk_do_floatow ;
```

Zwróć uwagę, że w definicji jest powiedziane, że wskaźnik pokazuje na obiekty. Referencja (przezwisko) nie jest obiektem, dlatego nie może być do niej wskaźników. Także nie może być wskaźników do pojedynczych bitów jakiegoś słowa - tzw. pól bitowych (patrz str. 323).

W naszej definicji stworzyliśmy sobie wskaźnik, ale nie pokazuje on jeszcze na nic rewelacyjnego. To tak, jakbyśmy na lekcję geografii wystrugali wskaźnik z drewna i położyli go na boku. Mimo, że wskaźnik leży na boku, to przecież jego koniec zawsze na coś pokazuje (celuje), ale z tego pokazywania jeszcze nie można się szczycić.

> Wskaźnik służący do pokazywania na obiekty jednego typu nie nadaje się do pokazywania na obiekty innego typu. †)

Konkretnie: wskaźnikiem do `int` nie można pokazywać na `float` czy string. Próba ustawienia wskaźnika na obiekt innego, niewłaściwego typu wywoła protest kompilatora.

8.3 Praca ze wskaźnikiem

Oto jak można nadać naszemu wskaźnikowi wartość początkową, czyli sprawić, by na coś pokazywał.

Używa się do tego jednoargumentowego operatora `&` (ampersand). To on oblicza adres obiektu, który stoi po prawej stronie przypisania.

```
int *w ;          // definicja wskaźnika do obiektów typu int
int k = 3 ;       // definicja zwykłego obiektu typu int z liczbą 3

      w = &k ;    // ustawienie wskaźnika na obiekt k
```

Kiedy wskaźnik już pokazuje na żądane miejsce – możemy odnieść się do tego obiektu, na który pokazuje.

Odnieść się do obiektu – rozumiem jako na przykład: odczytać jego zawartość, czy też coś zapisać do niego. Do takiej operacji służy jednoargumentowy operator odniesienia się `*` (gwiazdka).

„–Znowu gwiazdka?" – zapytasz pewnie – „nie za dużo tego?" Odpowiem jednak: - Jest w tym logika. Otóż zapamiętaj sobie złotą myśl:

> W języku C++ definicje obiektów zapisywane są tak, że zapis przypomina sposób, w jaki później dany obiekt występuje w wyrażeniach.

Definicja tablicy zawierała klamry

```
int t[5] ;
```

†) W rozdziale o dziedziczeniu dowiemy się, że od tej reguły jest chwalebny wyjątek.

dlatego, później w wyrażeniach klamry oznaczają pracę na tablicy

```
a = t[1] ;          // odczytanie pierwszego elementu tablicy.
```

Nie inaczej jest w wypadku wskaźnika. Jeśli mamy nasz wskaźnik i już ustawimy go na obiekt k, to treść pokazywanego obiektu odczytuje się jednoargumentowym operatorem *

```
int *w ;                          // definicja wskaźnika
int k = 3 ;                       // definicja zmiennej

w = &k ;                          // ustawienie wskaźnika
cout << "W obiekcie pokazywanym przez "
         "wskaznik w jest wartosc " << (*w) ;
```

Wykonanie tego fragmentu programu da nam na ekranie

```
W obiekcie pokazywanym przez wskaznik w jest wartosc 3
```

Słowem: gwiazdka kieruje nas do obiektu pokazywanego przez wskaźnik. Nasz wskaźnik pokazywał na zmienną k. Zatem od tej pory *w oznacza to samo co k. Skoro zawartością k była liczba 3, to ta wartość pojawiła się na ekranie.

Wniosek: po ustawieniu wskaźnika obie poniższe instrukcje są równoważne

```
cout << k ;                       // wypisz na ekran k
cout << *w ;                      // wypisz na ekran k
```

Przyjrzyjmy się teraz takiemu programowi:

```
#include <iostream.h>
main()
{
int zmienna = 8 , drugi = 4 ;                          // ❶
int *wskaznik ;                                        // ❷

     wskaznik = &zmienna ;                             // ❸

     // prosty wypis na ekran ;                        ❹
     cout << "zmienna = " << zmienna
          << "\n a odczytana przez wskaznik = "
          << *wskaznik << endl ;

     zmienna = 10 ;                                    // ❺
     cout << "zmienna = "<< zmienna
          << "\n a odczytana przez wskaznik = "
          << *wskaznik << endl ;

     *wskaznik = 200 ;                                 // ❻
     cout << "zmienna = " << zmienna
          << "\n a odczytana przez wskaznik = "
          << *wskaznik << endl ;

     wskaznik = &drugi ;                               // ❼
     cout << "zmienna = "<< zmienna
          << "\n a odczytana przez wskaznik = "
```

```
            << *wskaznik << endl ;
}
```

Po wykonaniu tego programu na ekranie pojawi się

```
zmienna = 8
 a odczytana przez wskaznik = 8
zmienna = 10
 a odczytana przez wskaznik = 10
zmienna = 200
 a odczytana przez wskaznik = 200
zmienna = 200
 a odczytana przez wskaznik = 4
```

❶ Tutaj definiujemy dwa najzwyklejsze obiekty typu `int`. Od razu inicjalizujemy je wartościami liczbowymi.

❷ Tutaj definiujemy wskaźnik. Definicję czyta się tak: `wskaznik` – jest wskaźnikiem (przepraszam !) do pokazywania na obiekty typu `int`.

❸ Tutaj sprawiamy, że wskaźnik zaczyna pokazywać na coś sensownego. Robimy to za pomocą operatora & – uzyskującego adres obiektu `zmienna`. Ten adres jest podstawiony do wskaźnika operatorem przypisania =
W zasadzie powinienem napisać linijkę ❷ i ❸ razem stosując skrócony zapis, jednak obiecałem sobie nie przerażać Cię. Taki skrócony zapis tych dwóch linijek wygląda tak:

```
int *wskaznik = &zmienna ;
```

❹ Wielokrotnie w trakcie tego programu wypisujemy na ekranie wartość tego, na co pokazuje wskaźnik. W uproszczeniu to po prostu

```
cout << *wskaznik ;
```

❺ Do `zmiennej` wpisujemy nową wartość. Kolejny wypis na ekran udowadnia, że wskaźnik cały czas pokazuje na ten obiekt i oczywiście zauważa zmianę.

❻ To jest bardzo ważna linijka. Skoro wyrażenie `*wskaznik` oznacza obiekt, na który pokazuje wskaźnik, to zapis

```
*wskaznik = 200 ;
```

oznacza: „do tego, na co pokazuje wskaźnik, wpisz liczbę 200". Ponieważ wskaźnik pokazywał na obiekt `zmienna`, więc do obiektu `zmienna` zostaje wpisana liczba 200. Chciałbym, żebyś docenił tę historyczną chwilę: Oto do obiektu można coś wpisać albo używając jego nazwy (`zmienna`), albo wskaźnika, który na ten obiekt pokazuje (`*wskaznik`) . Na dowód, że to działa – znowu wypisujemy to na ekran. Widzimy na ekranie dwukrotnie liczbę 200

❼ Wskaźnik nie pokazuje raz na zawsze na ten sam obiekt. Można go łatwo przestawić i od tej pory pokazuje na inny obiekt. Tutaj widzimy ustawienie wskaźnika tak, by pokazywał na obiekt `drugi`. (Robimy to, jak widać, wstawiając do wskaźnika adres zmiennej o nazwie `drugi`. Następny wydruk na ekranie pokazuje nam więc już liczby 200 i 4. Liczba dwieście jest treścią `zmiennej`, natomiast skoro wskaźnik pokazuje teraz na obiekt `drugi`, to wyrażenie `*wskaznik` odnosi się teraz do obiektu `drugi`.

Pomyślisz pewnie: „Co to za bzdurny program ! Po co do wpisania czy odczytania czegoś z obiektu używać wskaźnika, skoro łatwiej użyć nazwę tego obiektu." Masz rację, program jest rzeczywiście bzdurny, w zasadzie wskaźniki mogły by w nim w ogóle nie istnieć. Chciałem w nim jednak pokazać, że:

> Do danego obiektu można odtąd odnosić się na dwa sposoby: albo przez jego nazwę albo przez zorganizowanie wskaźnika, który na ten obiekt pokaże.

Mała dygresja o science-fiction

Kiedyś tłumaczyłem pracę na wskaźnikach mojemu kilkunastoletniemu przyjacielowi, miłośnikowi fantastyki naukowej. Kiedy doszliśmy do zapisu

```
a = *w ;
```

i próbowałem mu wytłumaczyć, że gwiazdka oznacza... przerwał mi i zdecydowanie powiedział kończąc jakąkolwiek dyskusję:

„To proste: gwiazdka oznacza, że lecimy gwiazdolotem do miejsca, na które pokazuje wskaźnik `w` i dopiero tamtym obiektem naprawdę się zajmujemy"

Możesz to, drogi Czytelniku uznać za dziecinne, ale takie mnemotechniczne skojarzenia bardzo pomagają w początkowej fazie rozumienia.

Drugim skojarzeniem mojego przyjaciela było, że jednoargumentowy operator & – (znaczek ten po angielsku zwany jest AMPERSAND) zaczyna się na literę a – tak samo jak słowo ADRES.

W sumie miał on więc takie regułki:

`* w`	–	**gwiazdolotem** w miejsce, na które pokazuje `w`
`& m`	–	**a, jak adres** obiektu `m`

Przypominam, że „gwiazdolot" występuje tylko w wyrażeniach. Zupełnym wyjątkiem jest gwiazdka w linijce definicji wskaźnika. Tam ma ona przypominać sposób w jaki wskaźnika się będzie potem używało.

8.4 L-wartość

Operacja odniesienia się do danego obiektu może być po prawej stronie znaku = (przypisania)

```
a = *wskaznik ;
```

ewentualnie po jego lewej stronie

```
*wskaznik = 55 ;
```

Jeśli coś może stać po lewej stronie znaku = (mówiąc mądrzej - po lewej stronie operatora przypisania), to takie „coś" nazywamy l-wartością. Nazwa łatwa do

zapamiętania, bo l – jak lewa strona. Po angielsku nazywa się to l-value[†], a mówię o tym dlatego, że jeśli kiedyś napiszesz przez pomyłkę instrukcję

```
10 = i ;                    // 10 is not a l-value !
```

To kompilator zaprotestuje i powie Ci (po angielsku), że instrukcja ta jest błędna jako, że stojące po lewej stronie 10 nie jest słynną l-value. Czyli, że liczba 10 nie nadaje się do postawienia po lewej stronie. Po prawej tak, ale nie po lewej

```
i = 10 ;                    // jest poprawne
```

Analogicznie wyrażenie, które może stać po prawej stronie nazywamy r-wartością (od angielskiego right – prawy). Jednak po prawej stronie to stać może już byle kto, więc być r-wartością to żaden honor. L-wartość - to jest coś wyjątkowego.

Oczywiście l-wartość jest także r-wartością. Ale nie odwrotnie.

Dlaczego to takie ważne, że wyrażenie

```
*wskaznik
```

jest l-wartością ?

Dlatego, że mamy wyrażeniem `*wskaznik` posługiwać się w takich samych sytuacjach jak wyrażeniem `zmienna`, które tą l-wartością jest.

Zatem jeśli `wskaznik` pokazuje na `zmienna`, to poniższe zapisy są równoważne

```
zmienna = 6 ;        *wskaznik = 6 ;        // jako l-wartości

int m ;
m = zmienna ;        m = *wskaznik ;        // jako r-wartości
```

8.5 Wskaźniki typu `void`

Wskaźnik – jak wielokrotnie już mówiliśmy – to adres jakiegoś miejsca w pamięci plus wiedza o tym, jakiego typu obiekt pokazuje się.

Czyli z definicji

```
int *a ;
```

wynika, że a to wskaźnik, którym można przechować adres jakiegoś obiektu, a ta wiedza to pewność, że to obiekt typu `int`.

Możemy jednak zdefiniować wskaźnik bez tej „wiedzy". Mówimy wtedy, że jest to wskaźnik typu `void`.

```
void *w ;
```

†) (czytaj: „el-velju")

„Wiedza" ta – przypominam – służy po to, by można było poprawnie wskazywane miejsce zinterpretować (rozkodować jako obiekt typu `int`, `float` *itd.) oraz po to, by móc w poprawny sposób wskaźnikiem poruszać po ewentualnych sąsiednich obiektach – gdy mamy je zebrane w tablicę.*

Teraz mamy wskaźnik typu `void`. Jasne jest, że skoro z tej „wiedzy" świadomie rezygnujemy, to automatycznie nasz wskaźnik nie może służyć do odczytania tego miejsca, na które pokazuje. Nie można też nim poruszać po sąsiadach.

Pytanie: Po co nam wobec tego taki upośledzony wskaźnik?

Po to, że czasami ta wiedza staje się niepotrzebnym balastem.

Wyobraź sobie taką sytuację: mamy następujące trzy wskaźniki

```
int *wi1, *wi2 ;
float *wf ;
```

W trakcie programu robiliśmy różne rzeczy, na przykład taką operację:

```
wi1 = wi2 ;
```

co oznacza: „a teraz niech wskaźnik `wi1` pokazuje na to samo, co wskaźnik `wi2`". Nie ma problemu. Jednak operacja

```
wf = wi2 ;        // błąd
```

będzie przez kompilator sygnalizowana jako błąd. Zaprotestuje on: „–Jak to, wskaźnikiem do pokazywania na obiekty typu `float` chcesz pokazać na to samo, na co pokazuje wskaźnik `wi1` – czyli na obiekt typu `int`? "

W zasadzie kompilator ma rację.

Możemy oczywiście sobie tak wskaźniki ustawić, ale wtedy trzeba wyraźnie kompilatorowi dać do zrozumienia, że nie robimy tego przez pomyłkę, tylko świadomie. Posłużyć się możemy w tym celu operacją rzutowania

```
wf = (float *) wi1 ;
```

co można przetłumaczyć jako: „Wiem, że `wi1` jest wskaźnikiem (do) innego typu, ale potraktuj go jako wskaźnik (do) typu `float` i mimo wszystko ustaw wskaźnik na to samo, na co pokazuje właśnie `wi1`.

Do tego jednak musi być rzutowanie (umieszczone u nas w nawiasie). Bez rzutowania kompilator uznaje to przypisanie za błędne.

A teraz uważaj: Jeśli wskaźnik stojący po lewej stronie naszego przypisania (podstawienia) byłby typu `void`, to mimo braku rzutowania kompilator by nie zaprotestował.

```
wv = wi1 ;
```

Zapis ten oznacza: niech wskaźnik typu `void` pokazuje na to samo miejsce w pamięci, na które właśnie pokazuje wskaźnik typu `int`.

Możemy więc wykonywać takie operacje:

```
// definicje kilku wskaźników
void *wv ;                    // typu void

char *wc ;
```

```
int  *wi ;
float *wf ;

// tutaj w programie ustawia się te wskaźniki na jakieś obiekty
....
// a teraz przypisania do wskaźnika typu void (bez konieczności
                                                  // rzutowania)
wv = wf ;
wv = wc ;
wv = wi;
```

Tu nieco wybiegnę w przyszłość:

Jest jeden warunek: nie można temu wskaźnikowi przypisać treści wskaźnika do obiektu z przydomkiem `const`. *To oczywiście dlatego, by zapobiec oszustwom. Po takim przypisaniu można by bezkarnie zmienić obiekt, który miał być przecież* `const`.

Wobec powyższego sformułujmy wniosek:

> Wskaźnik każdego (niestałego) typu można przypisać wskaźnikowi typu `void`

`void` ←——— dowolny wskaźnik (non–`const`) ;

W trakcie takiego przypisywania przekazywany jest adres, a „wiedza" jest zapominana – i to jest w porządku.

Natomiast nie da się odwrotnie.

To znaczy wskaźnika typu `void` nie można przypisać wskaźnikowi „prawdziwemu". Trzeba posłużyć się operatorem rzutowania. Dla powyżej zdefiniowanych wskaźników wygląda to tak:

```
wf = (float *) wv ;
wi = (int *) wv ;
wc = (char *) wv ;
```

Jak widać konieczny jest tu operator rzutowania, bo następuje przypisanie

```
float*   ←———— void*
int*     ←———— void*
char*    ←———— void*
```

Reasumując:

> Wiedzę o typie obiektu pokazywanego można ewentualnie niezauważenie zapomnieć. Nabyć tej wiedzy niezauważenie nie można. Trzeba to świadomie wyspecyfikować operacją rzutowania.

Uwaga dla programistów C

Jest to jedna z różnic między C++ a ANSII C. W ANSII C było tak, że niezależnie, po której stronie operatora przypisania stał wskaźnik typu `void` *nie trzeba było używać rzutowania. Zatem „wiedzę" nabywało się także niezauważenie. Było to jakby wyjście awaryjne dla umożliwienia łatwiejszego zapisu instrukcji rezerwujących pamięć. (Funkcjami w rodzaju* `malloc()` *– memory allocation). Ponieważ w C++ do tego celu służą dziecinne proste operatory* `new` *i* `delete`*, dlatego regułę można zaostrzyć.*

8.6 Cztery domeny zastosowania wskaźników

Wskaźniki stosuje się w sytuacjach, gdy chodzi nam o:

- – ulepszenie pracy z tablicami,
- – funkcje mogące zmieniać wartość przysyłanych do nich argumentów,
- – dostęp do specjalnych komórek pamięci,
- – rezerwację obszarów pamięci.

W dalszej części tego rozdziału przyjrzymy się przykładom z tych czterech dziedzin zastosowań.

8.7 Zastosowanie wskaźników wobec tablic

8.7.1 Ćwiczenia z mechaniki ruchu wskaźnika

Jeśli mamy następujące definicje:

```
int *wskaznik ;                    // definicja wskaźnika
int tablica[10] ;                  // definicja tablicy
```

to instrukcja

```
wskaznik = & tablica[n] ;          // ustawienie wskaźnika
```

powoduje, że wskaźnik ustawia się na elemencie tablicy o indeksie ***n***. Wskaźnik nasz jest, jak wiadomo, wskaźnikiem do pokazywania na obiekty typu `int`. Elementy naszej tablicy są właśnie typu `int`, więc wszystko się zgadza. W naszej ostatniej instrukcji po prostu wstawiamy do wskaźnika adres n-tego elementu tablicy.

Instrukcja

```
wskaznik = & tablica[0] ;
```

jest ustawieniem wskaźnika na zerowym elemencie, czyli na początku tablicy.

Ponieważ (już kiedyś mówiliśmy o tym, a Ty obiecałeś powiesić sobie to nad biurkiem) – nazwa tablicy jest równocześnie adresem jej początku, dlatego równie dobrze moglibyśmy tę ostatnią instrukcję napisać tak:

```
wskaznik = tablica ;
```

A teraz uważaj, będzie coś bardzo ciekawego: Jeśli ustawiliśmy wskaźnik na żądany element tablicy, np. tak:

```
wskaznik = &tablica[4] ;
```

to, aby go potem przesunąć tak, by pokazywał na następny element tej tablicy wystarczy do niego dodać 1

```
wskaznik = wskaznik + 1 ;
```

czyli krócej

```
wskaznik++ ;
```

To jest właśnie ta prostota przesunięcia w rozkładzie jazdy palca o jedną kratkę, by odczytać następny pociąg. Aby wskaźnik przesunąć o n elementów wystarczy instrukcja

```
wskaznik += n ;          // inaczej : wskaźnik = wskaźnik + n ;
```

Jest to bardzo ważny fakt:

> Dodanie do wskaźnika jakiejś liczby całkowitej powoduje, że pokazuje on tyleż elementów tablicy dalej. Niezależnie od tego jakie są te elementy.

Nie jest to takie trywialne, bo przecież mogą być tablice typu `float`, których elementy zajmują w pamięci większą przestrzeń niż np. typu `int`. A jednak wskaźnik jest na tyle inteligentny, że wie jak się ma przesunąć, aby przeskoczyć o zadaną liczbę elementów.

Skąd to wie? Ze swojej własnej definicji! Jest przecież wskaźnikiem do pokazywania na obiekty jakiegoś wybranego typu. Wiedząc jakim typem ma się zajmować – zna rozmiar jednego elementu i może przyjąć na to poprawkę.

Zobaczmy teraz na przykładzie, jak sprytne jest przesuwanie wskaźników

Poniższy program służy do wydruku adresu, na który pokazują wskaźniki. Interesuje nas tutaj tylko adres, na który wskaźnik pokazuje. Chwilowo nie zajmujemy się tym, co pod owym adresem się kryje.

```
#include <iostream.h>
main()
{
                                        // definicje dwóch tablic          ❶
int ti[6] ;                                     // jedna int
float tf[6] ;                                   //druga float
                                        //dwa wskaźniki                   ❷
int *wi ;                               // do pokazywania na obiekty int
float *wf ;                             // do pokazywania na obiekty float

     wi = &ti[0] ;              // inaczej wi = ti ;                      ❸
     wf = &tf[0] ;              // inaczej wf = tf ;                      ❹

     cout << "Oto jak przy inkrementacji wskaznikow\n "
          << "zmieniaja sie ukryte w nich adresy :\n" ;
     for(int i = 0 ; i < 6 ; i++, wi++, wf++)          // ❺
     {
          cout << "i= "<< i
               << ") wi = "
               << (unsigned long) wi                   // ❻
               << ", wf = "
               << (unsigned long) wf << endl ;         // ❼
     }
}
```

Po uruchomieniu takiego programu na komputerze klasy IBM PC/AT na ekranie pojawi się

```
Oto jak przy inkrementacji wskaznikow
 zmieniaja sie ukryte w nich adresy :
i= 0) wi = 1087246316, wf = 1087246292
i= 1) wi = 1087246318, wf = 1087246296
i= 2) wi = 1087246320, wf = 1087246300
i= 3) wi = 1087246322, wf = 1087246304
i= 4) wi = 1087246324, wf = 1087246308
i= 5) wi = 1087246326, wf = 1087246312
```

❶ Dwie definicje tablic. Tablica `ti` ma elementy typu `int`. Tablica `tf` ma elementy typu `float`.

❷ Definicje wskaźników. Wskaźnik `wi` może pokazywać na obiekty typu `int`, wskaźnik `wf` może pokazywać na obiekty typu `float`.

❸ i ❹ Nadanie wartości początkowych wskaźnikom. Po prostu wstawia się do nich adresy obiektu, na który mają pokazywać. Wskaźnik `wi` ma pokazywać na zerowy element tablicy `ti`, natomiast wskaźnik `wf` ma pokazywać na zerowy obiekt tablicy `tf`. Skoro mają pokazywać na początki tych tablic, to równie dobrze można użyć zapisu, który podany jest w komentarzu.

❺ Pętla. Nie ma w niej nic niezwykłego poza tym, że po każdym obiegu wykonywane są:

```
wi++ ;    wf++ ;
```

❻Wypisanie na ekran adresu, na który wskaźnik `wi` pokazuje. Adres to jakaś liczba. To, jak adresowane są komórki pamięci w danym komputerze, zależy od typu komputera. My tutaj przez operację rzutowania chcemy to wypisać tak, jakby to była wartość typu `unsigned long`.

❼Wypisanie tego samego o wskaźniku `wf`

Zauważ, że na ekranie liczby symbolizujące adresy zmieniają się z różnym skokiem mimo, że przecież dodawaliśmy do wskaźnika tylko jedynki. W tym właśnie objawia się inteligencja wskaźnika. Wie on jak naprawdę ma się zmienić po to, by wskazać na kolejny element tablicy.

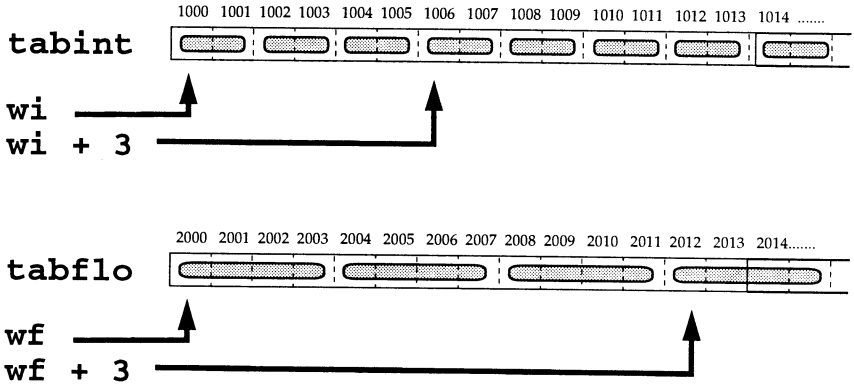
Skąd wskaźnik wie na jaki typ pokazuje? Stąd, że przecież go zdefiniowaliśmy jako: „wskaźnik służący do pokazywania na elementy typu `int`"

W miejscu wypisywania wskaźnika na ekranie widzisz

```
cout << (unsigned long) wi ;
```

przypominam, że operacja ujęta w nawias nazywa się rzutowaniem (casting). Mówiliśmy o niej na str 70. Przypomnę jednak, że oznacza ona mniej więcej coś takiego: „co prawda `wi` jest to wskaźnik, czyli w zasadzie adres, ale przeczytajmy go i zamieńmy na liczbę typu `unsigned long`" (i taką liczbę wydrukujmy na ekranie).

Przyjrzyjmy się poniższemu obrazkowi. (Liczby nad tablicami oznaczają przykładowe adresy poszczególnych komórek pamięci (bajtów) zajmowanych przez elementy tablic. Ze względów estetycznych wybrałem bardziej okrągłe liczby, a nie te, które pokazał nasz ostatni program).



Elementy tablicy typu `float` zajmują więcej miejsca w pamięci niż elementy typu `int`. Jednakże wskaźnik, który jest przeznaczony do pokazywania na dane typu `int` wie, jak ma się zmieniać przy przechodzeniu do sąsiedniego elementu tablicy. Podobnie wskaźnik przeznaczony do pokazywania na elementy typu `float` wie, jak zachować się przy przejściu do następnego elementu tablicy `float`.

Podobnie wyrażenie

```
(wf + 3)
```

daje w sumie adres trzeciego z kolei elementu tablicy za tym, na który właśnie wskazuje wskaźnik `wf`, a wyrażenie

```
(wf - 1)
```

jest adresem elementu o jeden wcześniejszego.[†)]

Wniosek stąd taki:

> Wskaźnik to nie tylko adres jakiegoś miejsca w pamięci. To także wiedza o tym, na jaki typ obiektu pokazuje. Wiedza ta wykorzystywana jest przy:
> a) interpretowaniu tego, na co wskaźnik pokazuje,
> b) ewentualnym poruszaniu wskaźnika

Przykład:

ad a) Wyobraź sobie bowiem, że wskaźnik pokazuje na jakiś bajt w pamięci[††)]. Jeśli wskaźnik jest typu `int` to wiadomo, że ten bajt i następny należy interpretować jako zakodowaną liczbę typu `int`. Jeśli zaś na ten sam bajt pamięci pokazywałby wskaźnik typu

†) W obu wypadkach sami musimy zadbać o legalność naszych poczynań. Jeśli `wf` pokazuje na początek tablicy, to wyrażenie `(wf - 1)` jest jakby adresem nieistniejącego elementu o indeksie: –1 (minus 1). O tych sprawach pomówimy za chwilę przy arytmetyce wskaźników.

††) Szesnastobitowego komputera IBM PC/AT

`float`, to wiadomo, że ten bajt i trzy następne (razem cztery) należy rozumieć jako zakodowaną liczbę typu `float`.

ad b) Kontynuując powyższy przykład: W przypadku tablicy typu `int`, aby przesunąć się do następnego elementu typu `int` - trzeba przeskoczyć o te dwa bajty. W przypadku tablicy typu `float` trzeba przeskoczyć rzeczone 4 bajty.

Wskaźniki są tak wspaniale pomyślane, że nie musimy o tych sprawach wiedzieć. Dodanie 1 do wskaźnika przesuwa go o właściwą ilość bajtów i pokazuje on na następny element.

Tak więc pamiętamy odtąd, że wyrażenie

```
(wi++)
```

oznacza przesunięcie wskaźnika do następnego elementu tablicy. Co ciekawsze – jeśli w przyszłości będziemy sobie wymyślali tablice z elementów mających po kilkaset bajtów, to i tak dodanie jedynki do wskaźnika służącego do pokazywania na tę tablicę przesunie poprawnie wskaźnik na następny element.

Dygresja:

Często mówiłem tutaj: wskaźnik „wie" – nadając wskaźnikowi jakąś osobowość i wolność działania. Oczywiście to nie wskaźnik „wie", tylko kompilator. Kompilator nasze wyrażenie `wi++` *zamienia na dodanie odpowiedniej liczby do obecnie tkwiącego w* `wi` *adresu. Czyli to kompilator jest tak mądry. Jednak – wybacz mi tę słabość - lepiej odpowiada mi interpretacja nadająca pewne cechy osobowości wskaźnikom. Życie w świecie takich wyobrażeń, gdzie wskaźnik to stary znajomy, którego przyzwyczajenia zna się na wylot, uprzyjemnia programowanie.*

8.7.2 Użycie wskaźnika w pracy z tablicą

Skoro już wiemy, jaki jest mechanizm przesuwania wskaźników – zobaczmy jak korzystać z tego, na co wskaźnik bieżąco pokazuje.

Zacznijmy od programu. Będzie to program na proste operacje na tablicach, jednak wykonywane będą one za pomocą wskaźników.

```
#include <iostream.h>
main()
{
int *wi ;
float *wf ;
int tabint[10] = { 0,1,2,3,4,5,6,7,8,9 };                    // ❶
float tabflo[10] ;                                            // ❷
                    // ustawienie wskaźnika
     wf = &tabflo[0] ;                                        // ❸

                    // załadowanie tablicy float wartościami początkow.
     for(int i = 0 ; i < 10 ; i++)
     {
          *(wf++) = i / 10.0 ;                   // tablica float ❹
     }

     cout << "Tresc tablic na poczatku\n" ;
```

```
        for(i = 0, wi = tabint, wf = tabflo                    // ❺
                          ; i < 10  ; i++)
        {
             cout << i << ") \t" << *wi << "\t\t\t\t"
                  << *wf << endl ;                             // ❻
             wi ++ ;                                           // ❼
             wf ++ ;
        }

        // nowe ustawienie wskaźników
        wi = &tabint[5] ;                                      // ❽
        wf = tabflo + 2 ;              // czyli wf = &tabflo[2] ; ❾

        // wpisanie do tablic kilku nowych wartości
        for(i = 0 ; i < 4 ; i++){
             *(wi++) = -222 ;                                  // ❿
             *(wf++) = -777.5 ;
        }

        cout << "Tresc tablic po wstawieniu nowych wartosci\n";
        wi = tabint ;
        wf = tabflo ;
        for(i = 0 ; i < 10 ; i++){
             cout << "tabint[" << i << "] = "
                  << *(wi++)
                  << " \t\ttabflo[" << i << "] = "
                  << *(wf++)
                  << endl ;
        }
}
```

❒ **Po wykonaniu tego programu na ekranie pojawi się:**

```
Tresc tablic na poczatku
0)   0                    0
1)   1                    0.1
2)   2                    0.2
3)   3                    0.3
4)   4                    0.4
5)   5                    0.5
6)   6                    0.6
7)   7                    0.7
8)   8                    0.8
9)   9                    0.9
Tresc tablic po wstawieniu nowych wartosci
tabint[0] = 0       tabflo[0] = 0
tabint[1] = 1       tabflo[1] = 0.1
tabint[2] = 2       tabflo[2] = -777.5
tabint[3] = 3       tabflo[3] = -777.5
tabint[4] = 4       tabflo[4] = -777.5
tabint[5] = -222        tabflo[5] = -777.5
tabint[6] = -222        tabflo[6] = 0.6
tabint[7] = -222        tabflo[7] = 0.7
tabint[8] = -222        tabflo[8] = 0.8
tabint[9] = 9       tabflo[9] = 0.9
```

Oto ciekawsze szczegóły programu:

❶ Definicja tablicy typu `int`. Od razu następuje inicjalizacja.

❷ Definicja tablicy typu `float`. Nie inicjalizujemy.

❸ Ustawienie wskaźnika na początkowy element tablicy.

❹ Po ustawieniu wskaźnika posługujemy się nim w pętli wpisującej do tablicy `tabflo` wartości początkowe. Wpisanie dzieje się za sprawą instrukcji

```
*(wf++) = i / 10.0 ;
```

co oznacza inaczej

```
*wf = i / 10.0 ;
wf ++ ;
```

Odbywa się więc tutaj wpisanie liczby do elementu tablicy pokazywanego właśnie przez wskaźnik `wf`, a następnie przesunięcie tego wskaźnika na element następny.

❺ Jest to pętla wypisująca na ekran treść tablic. Ponieważ chcemy to robić metodą przesuwania wskaźników, dlatego ustawiamy je na początkach tablic. Zauważ, że instrukcje ustawiające wskaźniki są wewnątrz instrukcji `for`. Dokładniej mówiąc w tej części, która jest wykonywana przed pierwszym obiegiem pętli. Zastosowaliśmy tu dla odmiany inny sposób ustawiania wskaźnika

```
wi = tabint ;
```

Sposób ten może nie jest tak od razu jasny, ale (pamiętasz?) umówiliśmy się, że przykleisz sobie nad biurkiem kartkę z napisem:

Nazwa tablicy jest równocześnie adresem jej zerowego elementu

To tłumaczy wszystko. Po prostu wyrażenia `(tabint)` oraz `(&tabint[0])` są równoważne.

❻ Wypisanie na ekran elementu pokazywanego przez wskaźnik to – jak wiemy – instrukcja typu

```
cout << *wi ;
```

❼ Przesunięcie wskaźników do następnego elementu. Mogliśmy to zrobić także zwięźlej w poprzedniej linijce operatorem postinkrementacji zapisując

```
cout << *(wi++) ;
```

❽ Zamierzamy w wybrane miejsca tablic wpisać nowe wartości. Skoro chcemy użyć do tego celu szybkiego sposobu za pomocą wskaźnika, dlatego musimy najpierw ustawić sobie wskaźnik. W wypadku `tabint` ustawiamy wskaźnik na elemencie `tabint[5]`.

❾ Natomiast w wypadku tablicy `tabflo` zacząć się to ma od elementu o indeksie 2. Zastosowaliśmy tu inny zapis, cały czas pamiętając, że przecież

```
wf = tabflo + 2 ;
```

to to samo co:

```
wf = &tabflo[2] ;
```

⑩ Wpisanie w te miejsca pamięci, gdzie pokazują wskaźniki. Przy wpisaniu od razu dokonujemy postinkrementacji wskaźników. Wpisujemy, jak widać, do kolejnych czterech elementów tablic. Następne linijki programu to po prostu pokazanie efektów pracy na ekranie.

Widzisz więc, że nie ma nic specjalnie trudnego w posługiwaniu się wskaźnikami. Zapytasz pewnie: „–A właściwie po co to wszystko, skoro da się to zrobić starym sposobem posługując się tablicami?" Masz rację. Mówiłem przecież, że bez wskaźników można żyć. Jest jednak zaleta. Posłużenie się wskaźnikiem da szybszy program. Pamiętasz naszą przypowieść o rozkładzie jazdy? To właśnie przecież robiliśmy tutaj. Na każde pytanie „–A następny?" odpowiadaliśmy instrukcją `wi++` i już byliśmy przy następnym elemencie. Będąc przy elemencie `tabint[5]` jednym skokiem znajdowaliśmy się przy elemencie `tabint[6]`, bez żmudnego liczenia adresu. Liczenie adresu przecież chwilę trwa.

Nazwa tablicy a wskaźnik

A teraz wróćmy jeszcze do sprawy omawianej w punkcie ❺. Skoro dwie poniższe instrukcje ustawiające wskaźnik na początku tablicy są równoważne

```
wskaznik = &tablica[0] ;
wskaznik = tablica ;
```

czyli, że wskaźnikowi można przypisać nazwę tablicy, to właściwie zachowuje się ona tak, jak wskaźnik. Istotnie – są ogromne podobieństwa. Zapis

```
tablica + 4
```

oznacza to samo co

```
&tablica[4]
```

Czy zatem zamiast naszej złotej regułki:

> Nazwa tablicy jest równocześnie **adresem** jej zerowego elementu

nie powinniśmy zastąpić regułką:

> Nazwa tablicy jest równocześnie **wskaźnikiem** do jej zerowego elementu

Tak, możemy to zrobić pod warunkiem, że będziemy pamiętać, iż to nie zwykły wskaźnik, tylko taki, którego nigdy nie będziemy przesuwać. (czyli: `const`)

Pokażmy tę różnicę jaśniej. Po instrukcji

```
wskaznik = tablica ;
```

`wskaznik` pokazuje na początek tablicy. O ile możemy postąpić tak:

```
wskaznik ++ ;
```

czyli przesunąć wskaźnik tak, by pokazywał na element następny, o tyle operacja

```
tablica ++ ;                    // błąd !!!
```

jest niedopuszczalna. A zatem nasza złota regułka powinna brzmieć tak:

> Nazwa tablicy jest jakby stałym wskaźnikiem do jej zerowego elementu.

Mimo tej bardzo mądrej konkluzji nie radzę napisanej przedtem złotej regułki zmieniać i zastępować nową. Pojęcie „adres" przeraża mniej niż pojęcie „stały wskaźnik". Przynajmniej na początku.

Oto dalsze konsekwencje. Skoro nazwa tablicy to właściwie wskaźnik, więc poniższe wyrażenia są równoważne

```
tablica[3]
*(tablica+3)
```

Wartością obu jest treść tego elementu tablicy, który ma indeks 3

Nie posądzaj mnie też, że w drugim z tych wyrażeń dokonuje się zabronionego przesuwania wskaźnika-nazwy tablicy. Dodanie +3 nie jest przesunięciem, tylko powiedzeniem, że chodzi nam o trzy pozycje dalej niż on teraz pokazuje.

> Oto ilustracja: Gdy znajomi pytają mnie, w którym miejscu na długiej ulicy Bülow Straße mieszkam, wówczas mówię, że tam, gdzie chińska restauracja, tylko trzy domy dalej. Wskaźnikiem jest tutaj „chińska restauracja". „Trzy domy dalej" – to operacja, którą przeprowadzamy w myśli, bez rujnowania chińskiej restauracji.

Inną różnicą między wskaźnikiem, a nazwą tablicy jest to, że wskaźnik jest jakimś obiektem w pamięci, więc można ustalić jego własny adres. Nie to, na co ten wskaźnik pokazuje, ale to, gdzie sam się mieści. Robi się to starym sposobem. Jeśli mamy wskaźnik

```
int *wskaznik ;                 // definicja wskaźnika
```

to jego adres jest wartością wyrażenia

```
&wskaznik
```

Natomiast nie można takiej operacji przeprowadzić w stosunku do nazwy tablicy.

Nazwa nie jest obiektem. Tablica tak, ale nazwa nie. **To też dobrze zapamiętać.** Ja mam na imię Jurek. Sam jestem obiektem (materialnym), natomiast moje imię obiektem nie jest. (Jeśli ktoś na mnie pokaże patykiem, to ten patyk-wskaźnik jest obiektem materialnym).

8.7.3 Arytmetyka wskaźników

Mówiliśmy już, że można dodawać i odejmować liczby całkowite do wskaźników i w ten sposób przesuwać je po wskazywanej tablicy. Jest tu jednak podobna sytuacja jak z odnoszeniem się do nich za pomocą konwencjonalnego - „tablicowego" zapisu: Nie jest sprawdzana legalność takiej operacji. To znaczy:

jeśli tablica ma tylko 10 elementów, a my wskaźnik aktualnie pokazujący na element `tablica[5]` przesuniemy o 80 elementów dalej, to wskaźnik będzie pokazywał na nieistniejący element `tablica[85]`. Możemy to zinterpretować tak: będzie on pokazywał na takie miejsce w pamięci, które zajmowałby element `tablica[85]`, gdybyśmy tylko zdefiniowali byli tak dużą tablicę. Ponieważ jednak tego nie zrobiliśmy –w miejscu tym są zupełnie inne dane. Jeśli mimo wszystko spróbujemy przeczytać to miejsce wskazywane przez wskaźnik, otrzymamy bezsensowny wynik, natomiast próba zapisania tam czegoś - zniszczy istniejącą tam legalnie inną daną.

O błędzie takim nie ostrzeże nas kompilator. Legalność pokazywania wskaźnikiem nie jest bowiem sprawdzana. Na dodatek, niekoniecznie od razu po zniszczeniu, program może wykazać błędne działanie. Dopóki nie korzystamy z tej zniszczonej danej – dotąd wszystko wydaje się w porządku.

Takie błędy są najtrudniejsze do znalezienia, bowiem objawiają się w programie czasem bardzo daleko od miejsca, w którym je spowodowaliśmy.

Oto moja rada:

> Jeśli program z zupełnie niewiadomych przyczyn błędnie działa czy zawiesza się, to zastanów się nad wskaźnikami. Najczęściej okaże się, że gdy wpisywałeś coś w miejsce pamięci pokazywane przez wskaźnik – pokazywał on na niewłaściwe miejsce lub (o zgrozo!) w ogóle zapomniałeś nadać wskaźnikowi jakąkolwiek wartość początkową. W rezultacie pokazywał on w przypadkowe miejsce, a Ty coś tam zapisałeś niszcząc coś nieznanego.

Oprócz tych operacji można też wskaźniki od siebie odejmować

Co z takiej operacji wynika? Zastanówmy się: jeśli mamy wskaźnik `wa`, który pokazuje na piętnasty element tablicy, oraz wskaźnik `wb`, który pokazuje na dziesiąty element tablicy, to jaka jest różnica między tymi wskaźnikami? Czyli jaka jest wartość wyrażenia

```
(wb - wa)
```

Zdrowy rozsądek podpowiada: 5 elementów. Rzeczywiście wartością tego wyrażenia jest liczba 5.

> Odjęcie od siebie dwóch wskaźników pokazujących na różne elementy tej samej tablicy daje w rezultacie liczbę dzielących je elementów tablicy. Liczba ta może być ze znakiem ujemnym lub dodatnim.

Oto prosty przykład:

```
#include <iostream.h>

main()
{
int tablica[15] ;                          // definicja tablicy
int *wsk_a, *wsk_b, *wsk_c ;

     wsk_a = &tablica[5] ;
     wsk_b = &tablica[10] ;
```

```
    wsk_c = &tablica[11] ;

    cout <<   "(wsk_b - wsk_a) = " << (wsk_b - wsk_a)
         << "\n(wsk_c - wsk_b) = " << (wsk_c - wsk_b)
         << "\n(wsk_a - wsk_c) = " << (wsk_a - wsk_c)
         << "\n(wsk_c - wsk_a) = " << (wsk_c - wsk_a) ;
}
```

W rezultacie jako wynik otrzymamy

```
(wsk_b - wsk_a) = 5
(wsk_c - wsk_b) = 1
(wsk_a - wsk_c) = -6
(wsk_c - wsk_a) = 6
```

Czy zauważyłeś, że wielokrotnie podkreślałem, iż muszą to być wskaźniki pokazujące na tę samą tablicę? Dlaczego to takie ważne?

Przykład z geografii

Na ścianie wisi mapa Europy. Jednym, drewnianym wskaźnikiem pokazujemy na tej mapie na Paryż, a drugim wskaźnikiem na Berlin. Jaka jest różnica tych wskaźników? (odległość między tymi wskaźnikami). Odpowiedź jest na przykład taka: 28 centymetrów, co przy uwzględnieniu podziałki mapy może oznaczać tyle-set kilometrów. Przyznasz, że ta odpowiedź ma sens.

A teraz dodatkowo na drugiej ścianie zawieszamy inną mapę, na przykład mapę nieba. Jednym drewnianym wskaźnikiem pokazujemy na Rzym, a drugim na gwiazdozbiór Oriona. Jak jest różnica tych wskaźników (odległość między tymi wskaźnikami)? To pytanie nie ma sensu. Mapy mają inne skale, wiszą na przypadkowych ścianach. Jeśli nawet weźmiemy taśmę mierniczą i zmierzymy tę odległość, to co ona będzie oznaczać?

Pamiętajmy więc -

> Odejmowanie wskaźników daje wynik, który można sensownie zinterpretować tylko wtedy, gdy te wskaźniki pokazują na elementy w tej samej tablicy.

Nie ma sensu mnożenie dwóch wskaźników (nawet pokazujących na tę samą tablicę) – no bo co by to miało oznaczać? Nie ma też sensu dzielenie itd.

Podsumujmy:

> Legalnymi operacjami arytmetycznymi na wskaźnikach są:
> 1) dodawanie i odejmowanie od nich liczba naturalnych – daje to przesuwanie wskaźników,
> 2) odejmowanie dwóch wskaźników pokazujących na tę samą tablicę.

8.7.4 Porównywanie wskaźników

Wskaźniki można ze sobą porównywać. Dla porównania dwóch wskaźników posługujemy się operatorami

```
==    !=    <    >    <=    >=
```

Jeśli mamy dwa wskaźniki

```
int *wsk1, *wsk2 ;
```

i zostały one już ustawione tak, że pokazują na jakieś obiekty, to równość tych wskaźników oznacza, że pokazują one na ten sam obiekt.

```
if((wsk1 == wsk2)
     cout<< "oba wskazniki pokazuja na to samo !" ;
```

Jeśli wskaźniki są różne to znaczy, że pokazują na różne obiekty.

```
if (wsk1 != wks2)
     cout << "wskazniki pokazuja na rozne obiekty" ;
```

Zwracam uwagę, że elementy tablicy to są też obiekty. Element piąty jest innym obiektem niż element szósty. Jeżeli dwa wskaźniki pokazują na jakieś elementy tej samej tablicy, to wskaźnik, który jest mniejszy pokazuje na obiekt o mniejszym indeksie.

```
#include <iostream.h>
main()
{
int tablica[5] ;
int *wsk_czer, *wsk_ziel ;
int i ;

     wsk_czer = &tablica[3] ;
     cout << "Mamy piecioelementowa tablice \n"
              "Wskaznik czerwony pokazuje na "
              "element o indeksie 3\n"
              "Na ktory element ma pokazywac "
              "wskaznik zielony ? (0-4) : ";
     cin >> i ;

     if(i < 0 || i > 4)
          cout <<
           "\nNie ma takiego elementu w tej tablicy !" ;
     else
     {
          wsk_ziel = &tablica[i] ;
          cout <<
           "\nZ przeprowadzonego porownania wskaznikow\n"
               "czerwonego z zielonym wynika, ze : \n" ;

          // właściwa akcja porównania————————————

          if(wsk_czer > wsk_ziel) {
               cout <<   "zielony pokazuje na element "
                         "blizej poczatku tablicy" ;
          }
          else if(wsk_czer < wsk_ziel){
               cout <<   "zielony pokazuje na element "
                         "o wyzszym indeksie" ;
          }else{    // czyli: wsk_czer == wsk_ziel
               cout <<   "zielony i czerwony pokazuja "
                         "na to samo\n" ;
```

```
            }
        }
}
```

Po wykonaniu tego programu i przykładowej odpowiedzi 4, na ekranie pojawi się tekst:

```
Mamy piecioelementowa tablice
Wskaznik czerwony pokazuje na element o indeksie 3
Na ktory element ma pokazywac wskaznik zielony ? (0-4) : 4
Z przeprowadzonego porownania wskaznikow
czerwonego z zielonym wynika, ze :
zielony pokazuje na element blizej konca tablicy
```

W przykładzie nie ma użycia operatorów <= i >=, ale ich znaczenie jest chyba oczywiste.

Warto znowu przypomnieć, że operacje

```
>    <    >=    <=
```

mają sens tylko dla wskaźników pokazujących na tę samą tablice. Powód jest dokładnie taki sam, jak przy arytmetyce wskaźników.

Jeśli tylko wskaźniki są tego samego typu – czyli inaczej: służą do pokazywania na obiekty tego samego typu (`int` albo `char` itd.) – i pokazują one na obiekty nie należące do tej samej tablicy, to mimo powyższego zastrzeżenia wolno nam je porównać. Wolno nam także porównać wskaźniki pokazujące właśnie na zupełnie odosobnione zmienne. Jest tylko kwestia jaki sens ma takie porównanie. A sens jakiś ma! Dowiadujemy się w ten sposób jak w pamięci komputera ulokowane są względem siebie te obiekty. Wynik i jego nasza interpretacja zależy tutaj od konkretnego kompilatora, którym się posługujemy.

Każdy wskaźnik można porównać z adresem 0 zwanym czasem NULL. Jest to przydatna właściwość, bo ustawienia wskaźnika na ten adres często służy nam by zaznaczyć, że wskaźnik nie pokazuje na nic sensownego. Wpisujemy tam świadomie 0 :

```
wsk = 0 ;                    // lub : wsk = NULL ;
```

Potem możemy to ewentualnie łatwo sprawdzić

```
if(wsk == 0)
    cout << "Wskaznik nie pokazuje na nic sensownego !" ;
```

to samo sprawdzenie wskaźnika można wykonać jako

```
if(wsk == NULL) ...
```

albo jeszcze prościej

```
if(! wsk) ...
```

8.8 Zastosowanie wskaźników w argumentach funkcji

Mówiliśmy, że są 4 domeny zastosowania wskaźników. Kolejną, którą się teraz zajmiemy, to wskaźniki jako argumenty funkcji. W zasadzie mówiliśmy już o tym trochę w rozdziale o funkcjach. Tutaj rozszerzymy ten temat, ale najpierw

Przypomnienie

Jeśli mamy funkcję z jednym argumentem

```
int funkcja(int argum);
```

i gdy wywołamy ją tak:

```
int a, x = 5 ;
a = funkcja(x) ;
```

to funkcja ta otrzymuje wówczas do pracy kopię zmiennej x (a nie oryginał). Jeżeli nawet w obrębie funkcji na tej kopii dokonujemy jakichś zmian, to w momencie opuszczania funkcji jest ona likwidowana. Funkcja więc nie może dokonać zmian zmiennej przysłanej do niej jako argument – przez wartość. (Czy pamiętasz jeszcze tę przypowieść z fotografią teściowej?).

Innymi słowy, gdybyśmy np. chcieli mieć funkcję, która przysłany do niej parametr zwiększy o 130, a jej treść (czyli definicję) napisali tak:

```
void funkcja(int foto)
{
     foto += 130 ;              // czyli: foto = foto + 130 ;
}
```

to wywołanie funkcji

```
int m = 10 ;
funkcja(m) ;
```

nie spowoduje jakiejkolwiek zmiany m.

Co się tu dzieje ?

Zmienna zostaje sfotografowana, a jej fotografia znajdzie się w podręcznym magazynku funkcji (na stosie). W trakcie działania funkcji do tej fotografii zostaje dodana liczba 130, więc obiekt na stosie ma teraz już wartość 140.

Ponieważ funkcja nie ma już więcej nic do roboty, więc kończy się ją uprzątając wszystkie śmieci ze stosu. Wtedy to nasza fotografia przestaje istnieć. Gdy wróciliśmy z funkcji patrzymy na prawdziwą zmienną m – jest nietknięta. Bawiliśmy się jej kopią, która została zniszczona.

Przypomnieliśmy tutaj mechanizm przesyłania argumentów przez wartość. Tak przesyłane są zwykłe obiekty. (O tablicach mówiliśmy już, że jest inaczej).

Co zatem zrobić, by funkcja mogła obiekt m zmienić ?

Najprostsze i najbardziej zalecane wyjście jest takie, by funkcja jako swój rezultat zwracała wartość, którą my świadomie wpiszemy do obiektu m:

Oto definicja takiej funkcji:

```
int fun2(int foto)
{
     foto += 130 ;
     return( foto ) ;
}
```

A tak tę funkcję wywołujemy w programie:

```
int m = 10 ;
m = fun2(m) ;
cout << m ;
```

Po wykonaniu tego fragmentu na ekranie pojawi się liczba 140.

Co się tu odbywa: fotografowana jest zmienna `m` i jej wartość (czyli liczba 10) jest umieszczana na stosie jako obiekt typu `int` o nazwie `foto`. Powstał więc lokalny obiekt automatyczny. Następnie do tego obiektu `foto` dodawane jest 130, co powoduje, że w rezultacie w obiekcie `foto` jest teraz wartość 140.

Teraz funkcja się kończy – instrukcją `return`. To, co stoi obok słowa `return`, jest to wyrażenie, którego wartość staje się rezultatem funkcji. W naszym wypadku to wyrażenie składa się tylko z jednej zmiennej. Jego wartość to 140. Ta właśnie liczba zamieniana jest na typ deklarowanego rezultatu zwracanego przez funkcję. U nas deklarowaliśmy, że funkcja zwraca wartość typu `int`, więc konwersja jest w zasadzie niepotrzebna. (Gdybyśmy mieli jednak 140.1 to nastąpiłoby obcięcie do `int` czyli wartość 140). Po konwersji wartość tę zapamiętuje się w jakimś tajemniczym miejscu. Zaczynamy sprzątać śmieci ze stosu. Likwidowany jest więc obiekt `foto`. Wracamy do miejsca w programie skąd wywołaliśmy funkcję. Widzimy tam, że rezultat funkcji ma zostać przypisany obiektowi `m`, w którym nadal tkwi wartość 10. Odszukujemy schowany w bezpiecznym miejscu rezultat funkcji (140) i wstawiamy go do obiektu `m`. W tym momencie odbyła się modyfikacja obiektu `m`.

Sposób jest bardzo dobry dlatego także, iż patrząc na zapis

```
m = fun2(m) ;
```

od razu widzimy, że do zmiennej `m` wpisuje się coś nowego, więc zmiana jego wartości nie jest dla nas niespodzianką. Takie względy są bardzo ważne przy analizie cudzych programów, a nawet swoich własnych, które przy kilkunastu tysiącach linii już trudno nam opanować.

Co jednak zrobić w wypadku, gdy funkcja ma zmienić więcej niż jeden obiekt?

Opisanego wyżej sposobu zastosować się nie da, dlatego, że funkcja za pomocą instrukcji `return` zwraca tylko jedną wartość. Tu właśnie przydają się wskaźniki. Otóż zamiast wysłać do funkcji kopię naszej zmiennej wysyłamy... Pewnie pomyślałeś „oryginał".

Nie, tego zrobić nie można. Wysłanie argumentów do funkcji jest jakby napisaniem do niej listu.

Weźmy taki obrazek: W łazience zepsuł się nam kran. Piszemy list do hydraulika. Nie możemy mu w liście przesłać tego kranu. Mamy dwa wyjścia:

- 1) Możemy mu przesłać fotografię tego zepsutego kranu. Wtedy hydraulik znajdzie u siebie w warsztacie na stosie rupieci taki sam kran (kopię naszego). Jeśli nawet sobie go naprawi, to nasz własny kran i tak będzie zepsuty.
- 2) Możemy mu w liście jednak przesłać – uwaga, uwaga! : ADRES tego naszego zepsutego kranu – nazwę ulicy numer domu, piętro i to, gdzie w mieszkaniu jest ten kran. Piszemy mu w liście, że ma naprawić kran będący pod takim adresem.

Oto jak to przełożyć na język programowania:

```
#include <iostream.h>
void hydraulik(int *wsk_do_kranu) ;                    // ❶
/*****************************************************/
main()
{
int kran = -1 ;                                        // ❷

     cout << "Stan techniczny kranu = "<< kran << endl ;
     hydraulik( &kran ) ;                              // ❸
     cout <<
          "Po wezwaniu hydraulika stan techniczny kranu = "
          << kran << endl ;                            // ❼
}
/*****************************************************/
void hydraulik(int *wsk_do_kranu)                      // ❹
{
     *wsk_do_kranu = 100 ;                   // akcja naprawiania ❺
}                                                      // ❻
```

Po wykonaniu tego programu na ekranie pojawia się:

```
Stan techniczny kranu = -1
Po wezwaniu hydraulika stan techniczny kranu = 100
```

Przyjrzyjmy się ciekawszym punktom

❶ Deklaracja funkcji `hydraulik`. Czytamy ją tak: `hydraulik` jest funkcją wywoływaną z jednym argumentem będącym wskaźnikiem do obiektu typu `int`. Funkcja ta zwraca typ `void`, czyli nic nie zwraca.

❷ Definicja obiektu typu `int` o nazwie `kran`. Wstawione tam od razu -1 oznacza, że kran jest bardzo zepsuty.

❸ Wywołanie funkcji `hydraulik`, której definicja jest w ❹. Prześledźmy co tu się zdarza. Otóż wzywamy hydraulika podając mu listownie adres zmiennej (operator jednoargumentowy `&` oznacza jak wiadomo „adres"). Co z tym adresem robi `hydraulik`?

❹ Widzimy, że przysłany adres służy `hydraulikowi` do inicjalizacji wskaźnika. Tak — bowiem `hydraulik` definiuje sobie (na stosie) wskaźnik do obiektu typu `int` i daje mu nazwę `wsk_do_kranu`. Odebrany od listonosza adres wstawia właśnie do tego wskaźnika. Odtąd więc jego prywatny (lepiej: lokalny) wskaźnik pokazuje na nasz kran.

❺ Do obiektu pokazywanego przez wskaźnik wstawia się liczbę 100. Wskaźnik pokazuje na nasz kran, więc to do naszego kranu wstawia się tę wartość 100. To jakby symbolizuje naprawę naszego kranu.

❻ Następnie opuszcza się funkcję, likwiduje się śmieci czyli zniszczony zostaje wskaźnik do naszego kranu – `hydraulik` podarł niepotrzebną mu już kartkę z adresem.

❼ Na dowód, że naprawa została dokonana naprawdę na naszym kranie, wypisujemy jego zawartość na ekran.
Gdybyśmy za chwilę wywołali funkcję `hydraulik` podając jej adres innego obiektu typu `int`, to zadziała ona na innym obiekcie. Jak w życiu: `hydraulik` naprawia jeszcze inny kran

```
int kurek = -10 ;
hydraulik(&kurek) ;
```

czyli wstawi on liczbę sto do obiektu `kurek` (bowiem adres tego obiektu właśnie wysłaliśmy).

Jeśli natomiast mamy całą baterię kurków

```
int bateria[15] ;
```

i chcemy naprawić czwarty kurek z tej baterii (tablicy), to wywołujemy

```
hydraulik( &bateria[4])
```

Chyba Cię ten zapis nie dziwi, oswoiłeś się zapewne z tym, że tak zapisuje się adres elementu tablicy.

Naprawa elementów od 4 do 8 tej baterii może zostać zrealizowana przez

```
for(int i = 4 ; i <= 8 ; i++){
     hydraluik( &bateria[i]);
}
```

8.8.1 Jeszcze raz o przesyłaniu tablic do funkcji

Jak pamiętasz, gdy do funkcji wysyła się tablicę jako całość, to wysyłane nie są kopie wszystkich jej elementów, ale po prostu jej adres. To też sprawia, że mając adres funkcja może pracować na oryginalnych jej elementach.

Jeśli jednak wysyłamy do funkcji jeden element tablicy, lub kilka - w każdym razie nie całość – to funkcja traktuje je jak zwykłe obiekty przesłane przez wartość. Jeśli chcemy wysłać je przez adres, to musimy powiedzieć to jasno – tak było właśnie przy wysyłaniu elementów baterii.

Porozmawiajmy teraz jednak o wysyłaniu tablicy jako całości

Odbywało się ono jakby według takiego schematu. Mając funkcję

```
void fun( int tab[] ) ;
```

oraz tablicę

```
int tablica[20] ;
```

wywołanie funkcji wyglądało tak:

```
fun(tablica);
```

Przesyłaliśmy do funkcji adres tablicy. (Złota regułka: nazwa tablicy jest adresem jej początku). Natomiast w poprzednim paragrafie zobaczyliśmy funkcję, która jest zdolna przyjąć jako argument – adres jakiegoś obiektu typu `int`.

Oto wywołanie funkcji `hydraulik` z argumentem będącym tablicą (całą):

```
hydraulik(tablica) ;
```

Jak widzisz zapis jest tu identyczny. Czy zatem `hydraulik` naprawi całą tablicę? Nie. On po prostu tego nie umie. Napisaliśmy funkcję hydraulik tak, że naprawia tylko obiekt o przysłanym adresie. Nazwa tablicy jest adresem jej zerowego elementu, więc naprawi on tylko ten zerowy element. Aby mógł naprawiać więcej musielibyśmy nauczyć go przesuwania tego wskaźnika.

Nie to jest tu jednak istotne. Chodzi o to, że tablicę można wysłać do funkcji jako tablicę, a odebrać na dwa sposoby:

- jako tablicę,
- jako wskaźnik.

8.8.2 Odbieranie tablicy jako wskaźnika

W rozdziale o tablicach mówiliśmy o tym, jak do funkcji wysłać tablicę. Przypominam, że tablicy nie wysyła się przez kopiowanie wszystkich elementów danej tablicy, bo może być ich potwornie dużo. Wysyła się do funkcji nazwę tej tablicy – która to nazwa, jak wiemy, jest przecież adresem jej początku.

Jednak skoro wysyłamy do funkcji ten adres, to możemy wobec tego odebrać go jako wskaźnik. Oto przykład kilku rodzajów funkcji:

```
#include <iostream.h>
/*****************************************************/
void funkcja_wska(int *wsk, int rozmiar) ;
void funkcja_tabl(int tab[], int rozmiar) ;
void funkcja_wsk2(int *wsk, int rozmiar) ;
/*****************************************************/
main()
{
int tafla[4] = { 5,10,15,20 } ;
     funkcja_tabl(tafla, 4);                    // ❶
     funkcja_wska(tafla, 4);                    // ❷
     funkcja_wsk2(tafla, 4);                    // ❸
}
/*****************************************************/
void funkcja_tabl(int tab[], int rozmiar)       // ❹
{
     cout << "\nWewnatrz funkcji funkcja_tabl \n" ;
     for (int i = 0 ; i < rozmiar ; i++)
          cout << tab[i] << "\t" ;
}
/*****************************************************/
void funkcja_wska(int *wsk, int rozmiar)        // ❺
{
     cout << "\nWewnatrz funkcji funkcja_wska \n" ;
     for (int i = 0 ; i < rozmiar ; i++)
```

```
        cout << *(wsk++) << "\t" ;                    // ❻
}
/*****************************************************/
void funkcja_wsk2(int *wsk, int rozmiar)
{
     cout << "\nWewnatrz funkcji funkcja_wsk2 \n" ;
     for (int i = 0 ; i < rozmiar ; i++)
          cout << wsk[i] << "\t" ;
}
```

Po wykonaniu tego programu na ekranie otrzymamy

```
Wewnatrz funkcji funkcja_tabl
5    10   15   20
Wewnatrz funkcji funkcja_wska
5    10   15   20
Wewnatrz funkcji funkcja_wsk2
5    10   15   20
```

Uwagi:

❶ Funkcję wywołujemy podając jej nazwę tablicy (czyli adres jej zerowego elementu). W definicji funkcji `funkcja_tabl` ❹ widzimy, że wysłany jej adres tablicy odebrany jest także jako tablica. Wewnątrz funkcji posługujemy się znanym zapisem „tablicowym". (Wiem, że oszukuję, ale cierpliwości!)

❷ Identycznie wyglądające wywołanie. Tym razem chodzi o inną funkcję ❺. Wewnątrz tej funkcji przysłany adres służy do inicjalizacji lokalnego wskaźnika `wsk`. Tak jakbyśmy wykonali taką instrukcję

```
int *wsk = tafla ;
```

Wskaźnikiem tym posługujemy się wewnątrz funkcji. W naszej funkcji zastosowaliśmy wyrażenie

```
*(wsk++)
```

które (przypominam) odpowiada złożeniu

`*wsk` *oraz* `wsk++`

czyli odczytaj coś, a potem przejdź do następnego elementu.

❸ Identyczne wywołanie. Tym razem to funkcja `funkcja_wsk2`. Jak widać z jej definicji odbiera ona tablicę w ten sam sposób, co funkcja powyżej. Najciekawsze jest to, że potem używa tablicy korzystając z notacji „tablicowej".

Jakie są wady i zalety tych typów funkcji? Czy lepiej odebrać jako tablicę, czy lepiej jako wskaźnik?

- Odebranie tablicy jako rzeczywiście tablicy – sprawia, że treść funkcji jest bardziej czytelna. Wskaźniki są genialnym narzędziem do zagmatwania zapisu.
- Odebranie tablicy jako adresu, którym inicjalizuje się wskaźnik – sprawia, że funkcja pracuje szybciej. Mówiliśmy już, że szybciej dociera się

do sąsiedniego elementu tablicy posługując się wskaźnikiem. (Pamiętasz jeszcze ten przykład z rozkładem jazdy?).

Aby znaleźć następny element tablicy wystarczy przesunąć tylko wskaźnik o 1 i gotowe. W wypadku tablicy – komputer musi żmudnie obliczyć położenie szukanego elementu tablicy. To trwa.

- Tablice wielowymiarowe łatwiej odbiera się w funkcji stosując notację wskaźnikową. Jest to sposób bardziej uniwersalny, bo rozmiary tablicy nie muszą być na stałe zaszyte w funkcji. Wystarczy jeśli funkcja dowie się o nich dopiero w momencie jej wywołania.

8.8.3 Argument formalny będący wskaźnikiem do obiektu `const`

Pamiętamy, że tablice do funkcji przysyła się nie tak, że funkcja otrzymuje kopie wszystkich elementów tablicy (np. w liczbie 8155) czyli nie przez wartość, ale tak, że funkcja otrzymuje adres tablicy. W rezultacie więc funkcja pracuje na oryginale tablicy i może dowolnie zmieniać jej elementy.

To bardzo wygodne, gdy funkcja naprawdę powinna zmieniać elementy tablicy – na przykład pomnożyć każdy przez 2.

Może być jednak sytuacja całkiem odwrotna. Czasem tablicę dajemy funkcji tylko po to, by ją sobie poczytała, ale nie chcemy, żeby w niej cokolwiek zmieniała. Jak się przed tym zabezpieczyć?

Tu właśnie przydaje się nam przydomek `const`, który może sprawić, że ze wskaźnika do obiektu zrobimy wskaźnik do **stałego** obiektu. Taki wskaźnik wskazuje na obiekty, ale nie pozwala na ich modyfikację.

Funkcja definiuje sobie na stosie wskaźnik do obiektu stałego. Otrzymany adres obiektu wstawia właśnie do takiego wskaźnika. Posługując się potem takim wskaźnikiem uniemożliwia sobie samej jakiekolwiek modyfikacje obiektu, na które on pokazuje.

```
#include <iostream.h>

// deklaracje funkcji ❶
void pokazywacz(const int *wsk, int ile);
void zmieniacz(int *wsk, int ile) ;
/*****************************************************/
main()
{
int tablica[4] = { 110,120,130,140} ;

     pokazywacz(tablica, 4);                          // ❷
     zmieniacz(tablica, 4);
     pokazywacz(tablica, 4);
     cout << "Dla potwierdzenia tablica[3] = "
          << tablica[3] ;
}
/*****************************************************/
void pokazywacz(const int *wsk, int ile)              // ❸
{
     cout << "Dziala pokazywacz " << endl ;
```

```
        for(int i = 0 ; i < ile ; i ++, wsk++)
        {
            // *wsk += 22 ;                          // błąd !   ❹
            cout << "element nr "<< i << " ma wartosc "
                << *wsk << endl;                          // ❺
        }
}
/*****************************************************/
void zmieniacz(int *wsk, int ile)                         // ❻
{
        cout << "Dziala zmieniacz " << endl ;

        for(int i = 0 ; i < ile ; i ++, wsk++)
        {
            *wsk += 500 ;                         // wolno nam !   ❼
            cout << "element nr " << i << " ma wartosc "
                << *wsk << endl;
        }
}
```

Po wykonaniu zobaczymy na ekranie

```
Dziala pokazywacz
element nr 0 ma wartosc 110
element nr 1 ma wartosc 120
element nr 2 ma wartosc 130
element nr 3 ma wartosc 140
Dziala zmieniacz
element nr 0 ma wartosc 610
element nr 1 ma wartosc 620
element nr 2 ma wartosc 630
element nr 3 ma wartosc 640
Dziala pokazywacz
element nr 0 ma wartosc 610
element nr 1 ma wartosc 620
element nr 2 ma wartosc 630
element nr 3 ma wartosc 640
Dla potwierdzenia tablica[3] = 640
```

Kilka uwag :

❶ Deklaracje funkcji. Są obowiązkowe, bo wywołania ich następują w programie wcześniej niż kompilator zobaczy ich definicje (są w tekście programu później). Gdyby nie były konieczne - i tak bym je napisał, taką już mam zasadę.

❷ Wywołanie funkcji `pokazywacz`. Wysyłamy tam tablicę. Czy nam się to podoba czy nie, funkcja dostaje jej adres i może nawet całą tablicę zniszczyć.

❸ Oto jak funkcja `pokazywacz` odbiera adres tablicy. Dostaje co prawda adres tablicy – czyli wszystkie uprawnienia, jednak funkcja definiuje wskaźnik z przydomkiem `const` i tam chowa adres tablicy. Równoważne to jest więc instrukcji

```
const int *wsk = tablica ;
```

> Jest to wskaźnik, który uznaje wskazywany obiekt za stały. Tym samym nie może takiego obiektu modyfikować.

Funkcja `pokazywacz` odbiera nazwę jako wskaźnik do **stałej** (a raczej do stałych). Innymi słowy oznacza to, że co prawda dostaliśmy w funkcji uprawnienia, ale obiecujemy z nich nie korzystać. Definiując właśnie taki wskaźnik, świadomie pozbawiamy się prawa do modyfikowania tablicy przysłanej jako argument.

> Sam oryginalny obiekt (tablica) nie jest obiektem stałym. Jednak za pomocą tak zdefiniowanego wskaźnika nie będziemy mogli go zmieniać. Innym, zwykłym wskaźnikiem oczywiście moglibyśmy.

❹ Ta linijka jest w komentarzu. Jest to próba modyfikacji obiektu wskazywanego przez wskaźnik. Jeśli chcesz się przekonać, jak kompilator strzeże obiektu wskazywanego przez nasz wskaźnik, to zlikwiduj ten komentarz. Już w czasie kompilacji otrzymasz informacje o błędzie.

❺ Odczytywać z obiektu wskazywanego przez taki wskaźnik oczywiście możemy. Nie jest to bowiem modyfikacja.

❻ To jest definicja funkcji `zmieniacz`. Adres użyty jest do inicjalizacji wskaźnika zwykłego typu, czyli nie-`const`. Tę sprawę już znamy – wskaźnikiem takim możemy dowolnie zmieniać pokazywane obiekty.

❼ Oto dowód na powyższe stwierdzenie. Modyfikacja elementów tablicy. Funkcja `zmieniacz` może swobodnie dodać liczbę 500 do elementów tablicy. O tym, że dzieje się to na oryginalnej tablicy, przekonuje nas wydruk jednego z nich w funkcji `main`.

Wskaźnik do obiektu stałego, to nie tylko złożona z dobrej woli obietnica. Są sytuacje, w których jest po prostu niezbędny. Jeśli mamy obiekt, który naprawdę jest stały, to nie można na niego pokazać innym wskaźnikiem jak tylko wskaźnikiem do stałej.

To w końcu zrozumiałe — inaczej moglibyśmy oszukiwać: co prawda sam obiekt jest stały, ale użyjemy tricku i zmienimy sobie jego zawartość posługując się wskaźnikiem. Tak się nie da. Kompilator nie pozwoli na danym stałym obiekcie ustawić żadnego wskaźnika, który z definicji nie obiecuje go nie zmieniać.

```
const int pojemnosc = 5 ;          // definicja stałego obiektu typu int
const int * staly_wsk ;            // definicja stałego wskaźnika
int *zwykly_wsk ;                  // definicja zwykłego

staly_wsk = & pojemnosc ;          // ustawienie wskaźnika na tym
                                   // obiekcie:
                                   // obiekt jest stały i wskaźnik też -
                                   // kompilator się zgadza

zwykly_wsk = & pojemnosc ;         // Błąd: obiekt jest stały, a wskaźnik
                                   // zwykły
                                   // - kompilator zaprotestuje !!
```

8.9 Zastosowanie wskaźników przy dostępie do konkretnych komórek pamięci

Trzecią domeną zastosowania wskaźników jest bezpośredni dostęp do specjalnie wybranych komórek pamięci. Chodzi tu o dostęp do komórki pamięci bez podawania jakiejkolwiek jej nazwy.

Dajmy na to, że w pamięci jest jakaś komórka o adresie 93952. Jest tam coś zupełnie szczególnego. (Np. komórka ta połączona jest zewnętrznie z miernikiem temperatury). Mamy zadanie wpisania tam jakiejś wartości lub odczytania jej. Komórka ta oczywiście nie ma nazwy.

Jak zatem się do niej odnosimy? Oczywiście za pomocą wskaźnika! Jak to zrobić konkretnie? Najpierw ustawiamy wskaźnik na żądaną komórkę wpisując do niego jej konkretny adres.

```
wsk = 93952 ;
```

Od tej pory posługujemy się już tym wskaźnikiem w znany sposób.

```
cout << "Obecna temperatura << *wsk ;
```

Niestety nie zawsze ustawienie wskaźnika na żądany adres jest tak proste – w różnych typach komputerów istnieją różne sposoby adresowania.

Dygresja dla wielbicieli IBM PC

W szczególności w komputerach klasy IBM PC jest to nieco skomplikowane. Jednak kompilatory dostarczają zwykle łatwych narzędzi do „zbudowania" konkretnego adresu. W kompilatorze Borland C++ mamy do dyspozycji makrodefinicję o nazwie `MK_FP` – (make far pointer). Użycie tej makrodefinicji rozwiązuje cały problem. Po szczegóły odsyłam do opisu kompilatora.

8.10 Rezerwacja obszarów pamięci

Czwartą domeną jest zastosowanie wskaźników przy rezerwowaniu jakichś obszarów pamięci.

Wiąże się z tym operator `new`, który tu właśnie będę reklamował.

> *Miłośnikom języka C podpowiem, że operator ten robi to samo, co znana im funkcja biblioteczna `malloc` (memory -allocation). O tej funkcji od dzisiaj należy zapomnieć, gdyż operator `new` robi to lepiej i łatwiej. (Na przykład w IBM PC uniezależnia nas od tak zwanego modelu pamięci).*

O co tutaj chodzi: W trakcie pisania programu nie zawsze wiadomo jak duże będą tablice, którymi chcemy się posługiwać. Powstaje pytanie: czy nie można by zrobić tak, że zaraz po starcie programu mówimy programowi jak wielka na być dana tablica?

Można. Do tego właśnie używa się operatora `new`. Natomiast problem, o którym mówimy nazywa się **dynamiczną alokacją (rezerwacją) tablic**.

Zanim pokażemy jak to zrobić, inny przykład kiedy operator `new` może się przydać:

Opracowujemy program kontroli lotów. Na ekranie obrazowane są samoloty lecące właśnie nad tym obszarem. Jeśli samolot wlatuje na nasze terytorium, pojawia się na brzegu mapy (ekranu) jako mały znaczek. Stopniowo przesuwa się w trakcie lotu, a kiedy opuszcza obszar – znika. Oczywiście te samolociki muszą istnieć już jakoś wcześniej w naszym programie. Tak jak musimy w tekście programu zdefiniować zmienną x jeśli mamy się nią kiedyś posługiwać. Musimy sobie więc gdzieś w programie napisać definicje obiektów reprezentujących te samoloty. Tylko ile ich napisać? 5, 10, 20? Na wszelki wypadek z zapasem definiujemy 25.

Niby „na wszelki wypadek", ale już w tym momencie ograniczyliśmy działanie programu od obsługi 25 samolotów – ale po co? Lepiej by było przecież niczego nie ograniczać.

W tym pomoże nam właśnie operator `new`. Jeśli dostaniemy – już w trakcie pracy programu – komunikat, że samolot wchodzi nad nasze terytorium, to dopiero wtedy zdefiniujemy nowy obiekt. Także nie ma problemu, gdy będą odbywać się pokazy lotnicze i w grę będzie wchodzić dodatkowych 100 obiektów. Będzie trzeba, to się je – już w trakcie pracy programu – zrobi operatorem `new`. Nie ma też problemu jeśli odbędzie się nalot dywanowy – proszę bardzo – nowe 4000 obiektów.

Wszystko to dzięki operatorowi `new` (i jego satelicie – operatorowi `delete` - likwidującemu potem te obiekty).

Inna sytuacja, kiedy mogą się jeszcze przydać `new` i `delete`.

Potrzebujemy wielkiej tablicy. Deklarujemy ją na przykład tak:

```
long tablica[4*8192] ;
```

a tu w trakcie linkowania dostajemy informację, że jest to błąd, ponieważ linker na tak wielkie tablice się nie zgadza. Łączna suma komórek z danymi nie może dla niego przekroczyć np. 64 KB i już. Co robić? Jest odpowiedź: Oszukać go za pomocą dynamicznej rezerwacji tablicy już w trakcie wykonywania programu.

8.10.1 Operatory `new` i `delete` albo Oratorium Stworzenie Świata.

Po takiej reklamie pora na przedstawienie. W rolach głównych wystąpią specjalne operatory `new` i `delete`[†]. Operator `new` zajmuje się kreacją, a `delete` unicestwianiem obiektów.

Do rzeczy: Jeśli mamy zdefiniowany np. taki wskaźnik:

```
char *wsk ;
```

†) new – ang: nowy (czytaj:„nju")
delete – ang: usuń (czytaj: „dilit")

to następująca instrukcja:

```
wsk = new char ;
```

powoduje utworzenie nowego obiektu typu `char`. **Nie ma on nazwy**, ale jego adres przekazywany jest wskaźnikowi `wsk`.
Z kolei instrukcja

```
delete wsk ;
```

powoduje likwidację tego obiektu. (Zakładam, że wskaźnik `wsk` nadal pokazywał na ten obiekt).
Inny przykład:

```
float *w ;
     w = new float[15] ;
```

Ostatnia instrukcja powoduje utworzenie piętnastoelementowej tablicy typu `float`. Tablica ta oczywiście nie ma nazwy, ale wskaźnik jest informowany o jej adresie. Kasowanie tej tablicy realizujemy instrukcją

```
delete [] w ;
```

Cechy obiektów stworzonych operatorem `new`

Cztery sprawy są tu bardzo ważne:

- Obiekty tak utworzone istnieją od momentu, gdy je utworzymy operatorem `new` do momentu, gdy je skasujemy operatorem `delete`. Inaczej mówiąc - **to my decydujemy o czasie ich życia**.
- Obiekt tak utworzony **nie ma nazwy**. Można nim operować tylko za pomocą wskaźników.
- Obiektów tych **nie obowiązują zwykłe zasady o zakresie ważności** – czyli to, w których miejscach programu są widzialne, a w których niewidzialne (mimo, że istnieją).
  Jeśli tylko jest w danym momencie dostępny choćby jeden wskaźnik, który na taki obiekt pokazuje, to mamy do tego obiektu dostęp.
- Tylko statyczne obiekty wstępnie inicjalizowane są zerami (o ile nie określiliśmy inaczej). Natomiast obiekty tworzone operatorem `new` nie są statyczne (wręcz przeciwnie – są dynamiczne!) dlatego **zaraz po utworzeniu tkwią w nich jeszcze śmieci**. Musimy sami zadbać o zapisanie tam sensownych wartości.

Oto przykład ilustrujący prostotę posługiwania się tym operatorem:

```
#include <iostream.h>
char * producent(void) ;                                  // ❶
/*****************************************************/
main()
{
char *w1, *w2, *w3, *w4 ;          // definicje wskaźników      ❷

     // tworzenie obiektów
     w1 = producent();                                    // ❸
```

```
     w2 = producent();
     w3 = producent();
     w4 = producent();

     *w1 = 'H' ;                                          // ❹
     *w2 = 'M' ;
     *w3 = 'I' ;

     cout <<"oto 3 znaki :" << *w1 << *w2 << *w3
          << "\noraz smiec w czwartym :" << *w4          // ❺
          << endl ;

     delete w1 ;          // kasowanie obiektow           ❻
     delete w2 ;
     delete w3 ;

     // *w1 = 'F' ;       // byłaby tragedia, bo obiekt
                          // już nie istnieje !!!         ❾
}
/*****************************************************/
char * producent(void)                                    // ❼
{
char *w ;
     cout << "Wlasnie produkuje obiekt \n";
     w = new char ;                                       // ❽
     return w ;
}
```

Po wykonaniu programu na ekranie pojawi się

```
Wlasnie produkuje obiekt
Wlasnie produkuje obiekt
Wlasnie produkuje obiekt
Wlasnie produkuje obiekt
oto 3 znaki :HMI
oraz smiec w czwartym : ¶
```

Uwagi do programu

❶ Deklaracja funkcji. Czytamy ją tak: `producent` jest funkcją wywoływaną bez żadnych argumentów, a która jako rezultat zwraca wskaźnik `(*)` do obiektu typu `char`.

❷ Definicja czterech wskaźników mogących pokazywać na obiekty typu `char`.

❸ Wywołanie funkcji `producent`, w której produkuje się obiekty typu `char`.

❼ Oto definicja funkcji `producent`. Jak widzisz to tutaj, w funkcji tworzymy obiekty. Wcale nie musieliśmy tutaj, wystarczyłoby w miejscu ❸ napisać instrukcje

```
w1 = new char ;
```

Jednak zrobiłem to celowo w funkcji - po to, by pokazać, że mimo, iż obiekty są tworzone wewnątrz funkcji, to jednak nie znikają po jej zakończeniu (jak to się zwykle dzieje z obiektami automatycznymi tworzonymi w funkcjach). Dlacze-

go? Otóż zwykłe obiekty definiowane wewnątrz funkcji są tworzone na stosie. Po zakończeniu pracy tej funkcji obiekty ze stosu są uprzątane.

❸ Zupełnie inaczej jest z obiektami tworzonymi operatorem `new`. Są one tworzone w obszarze pamięci, który przyznawany jest programowi do swobodnego używania. Obszar ten po angielsku nazywa się „free store" (swobodnie dostępny magazyn) lub heap (zapas). Trudno to dokładnie przetłumaczyć. Będę używał nazwy „zapas pamięci", bo najlepiej oddaje istotę problemu.

Zatem dzięki operatorowi `new` nasza funkcja właśnie tam, w dostępnym zapasie pamięci definiowała nowy obiekt, a informację o tym, w którym miejscu konkretnie (adres), przysłała jako rezultat funkcji.

Po zakończeniu działania funkcji nowy obiekt istnieje sobie nadal. Będzie istniał aż do momentu, gdy w dowolnym miejscu programu nie skasujemy go operatorem `delete`.

❹ Praca na nowych obiektach odbywa się tak, jak na zwykłych obiektach pokazywanych przez wskaźniki. Tu widzimy wpisanie czegoś do naszych trzech obiektów pokazywanych przez trzy wskaźniki. O czwartym obiekcie pokazywanym przez wskaźnik `w4` celowo zapominamy.

❺ Wypisujemy na ekran treść trzech obiektów - są tam oczywiście litery HMI. Wypisujemy też czwarty obiekt, do którego nic jeszcze nie wpisaliśmy, więc zawiera śmieci. Na ekranie pojawia się jakiś symbol będący w kodzie ASCII odpowiednikiem tkwiącego tam przypadkowego śmiecia. Jeśli uruchomisz ten program jeszcze raz, to śmieć będzie najprawdopodobniej inny. Tak było w moim wypadku. Jak śmieć to śmieć.

❻ Kasowanie obiektów. Tu właśnie zarezerwowane dla nich miejsce jest oddawane z powrotem do zapasu pamięci. Ewentualne następne wywołania funkcji `producent` mogą ten obszar znowu otrzymać.

❾ Skoro się oddało obszar pamięci z powrotem do zapasu, to go już nie ma. Nasz wskaźnik co prawda nadal pokazuje na to miejsce, ale tam może mieszkać już ktoś inny. Próba zapisania tam czegoś zniszczy tego ewentualnego nowego lokatora. Jak zwykle - tego typu błąd może objawić się o wiele później niż sam akt przestępstwa.

Obiektowi kreowanemu operatorem `new` można nadać wartość już w momencie stworzenia.

```
int * wsk ;
     wsk = new int(32);
```

Takie użycie operatora `new` sprawi, że w zapasie pamięci zostanie stworzony obiekt typu `int`, a do niego zostanie od razu wpisana liczba 32.

Można też sprawić, że obiekt stworzony zostanie w zapasie pamięci nie „byle gdzie", ale w określonym miejscu, na które pokazujemy wskaźnikiem. Jeśli mamy już wskaźnik `adr` ustawiony na konkretny adres, to wystarczy instrukcja o następującej składni:

```
wsk = adr new int ;
```

Jak widać – wystarczy bezpośrednio przed operatorem `new` umieścić upodobany adres (zapisany we wskaźniku `adr`).

8.10.2 Dynamiczna alokacja tablicy

Za pomocą operatora `new` można tworzyć (kreować) nie tylko pojedyncze obiekty, ale także i tablice.

```
int *tabptr ;
tabptr = new int[rozmiar] ;
```

gdzie *rozmiar* jest wyrażeniem typu `int`. W ten sposób stworzyliśmy nienazwaną tablicę elementów typu `int`. Wynikiem działania operatora `new` jest wskaźnik do początku tej tablicy. Podstawiamy go do naszego wskaźnika `tabptr`. Powyższe dwie linijki można napisać krócej jako:

```
int * tabptr = new int[rozmiar] ;
```

Zauważ, że rozmiar tablicy nie musi być stałą. Przypominam, że przy tradycyjnym sposobie definiowania tablic rozmiar musiałby być stałą znaną już w momencie kompilacji.

```
int tabliczka[15] ;
```

Operator `new` daje nam swobodę. Tablica definiowana jest dynamicznie, w trakcie wykonywania programu.

```
cout << "Ile elementow ma miec tablica ? \n" ;
int rozm ;
cin >> rozm ;
int *tabptr = new int[rozm] ;

//——— praca z tablicą ———————————
*tabptr = 44 ;            // wpisanie do zerowego elementu
tabptr[0] = 44 ;          // to samo inaczej

*(tabptr+3) = 100 ;       // wpisanie do elementu o indeksie 3
tabptr[3] = 100 ;         // to samo inaczej
```

Mówiliśmy kiedyś o tym, że zapis wskaźnikowy i tablicowy są w zasadzie wymienne, dlatego możemy do naszej tablicy stosować równie dobrze zapis „tablicowy". Oba zapisy pokazałem w powyższym przykładzie. Zapis „tablicowy" wydaje mi się łatwiejszy i bardziej naturalny.

Jednak uwaga: jeśli powiedzieliśmy, że wystarczy nam rozmiar 2 – to mamy tablicę tylko dwuelementową. Sami jesteśmy winni jeśli potem pracujemy na nieistniejącym elemencie czwartym i zdarzy się tragedia (strzał na oślep).

Aby zlikwidować tak wykreowaną **tablicę, stosujemy** operator `delete`.

```
delete [] tabptr ;
```

wynik działania operatora `delete` jest typu `void` (czyli nie zwracany jest żaden typ).

Za pomocą operatora `delete` kasuje się tylko obiekty stworzone operatorem `new`

Próba skasowania czegokolwiek innego może się okazać katastrofalna. Nie będzie jednak nieszczęścia jeśli zastosujemy operator `delete` w stosunku do wskaźnika pokazującego na adres zerowy (NULL) – takie sytuacje komputer sam rozpoznaje, bo żaden obiekt nie może mieć adresu 0.

Uwaga na pułapkę :

Łatwo się domyślić, że nie należy dwukrotnie kasować obiektu. Chodzi o sytuację, gdy obiekt stworzyliśmy operatorem `new`, potem skasowaliśmy go operatorem `delete`. Obiekt już nie istnieje. Tymczasem przez zapomnienie jeszcze raz bierzemy wskaźnik pokazujący na to miejsce w pamięci i wykonujemy kasowanie. Rezultat będzie niefortunny. Błąd nie musi ujawnić się od razu, dlatego trudniej go wykryć.

Ja radzę sobie w takich sytuacjach tak, że łącznie z kasowaniem obiektu, umieszczam instrukcję ustawiającą wskaźnik na NULL. Jak już wiemy ewentualne użycie przy kasowaniu adresu NULL nie jest katastrofalne.

```
int * wsk ;

     wsk = new int ;
     *wsk = 15 ;
     delete wsk ;
     wsk = NULL ;
     // ...
     delete wsk ;          // skoro NULL, to nie będzie tragedii
```

Druga pułapka przy tworzeniu obiektów operatorem `new`

Powiedzieliśmy, że obiekty tworzone za pomocą operatora `new` nie mają nazw. Pracujemy z nimi tylko za pośrednictwem wskaźników. Jest tu w związku z tym pułapka. Spójrz na ten program

```
#include <iostream.h>
main()
{
int *cze, *zol ;                    // def. 2 wskaźników          ❶

     cze = new int ;                // tworzymy obiekt A          ❷
     zol = new int ;                // tworzymy obiekt B

     *cze = 100 ;                   // ładujemy 100 do obiektu A  ❸
     *zol = 200 ;                   // ładujemy 200 do obiektu B

     cout << " Po wpisaniu : Na czerwonym = "<< *cze
          << " Na zoltym = " << * zol << endl ;

     cze = zol ;        // ←—— Niefortunna linijka !             ❹

     cout << " Po przelozeniu - Na czerwonym = "<< *cze
          << " Na zoltym = " << * zol << endl ;

     *cze = 5 ;
```

```
    *zol = 1 ;                                          // ❺

    cout << " Jakis wpis - Na czerwonym = "<< *cze
         << " Na zoltym = " << * zol << endl ;

    delete zol ;                                        // ❻
    // delete cze ;                    // Horror !
}
```

Po wykonaniu programu na ekranie zobaczymy

```
Po wpisaniu : Na czerwonym = 100 Na zoltym = 200
 Po przelozeniu - Na czerwonym = 200 Na zoltym = 200
 Jakis wpis - Na czerwonym = 1 Na zoltym = 1
```

Uwagi

❶ Najpierw definiujemy sobie 2 wskaźniki do obiektów typu `int`.

❷ Następnie operatorami `new` tworzymy dwa (nienazwane) obiekty typu `int`, a ich adresy wstawiamy do wskaźników: czerwonego `cze` i żółtego `zol`. Przypominam, że wartością wyrażenia

```
(new int)
```

jest adres nowowytworzonego obiektu.

❸ Do obiektów pokazywanych przez żółtego i czerwonego ładujemy 100 oraz 200.

❹ Ta linijka jest istotą naszego przykładu. Sprawia, że wskaźnik czerwony pokazuje odtąd na to samo, na co pokazuje wskaźnik żółty. Adres miejsca, na które pokazywał do tej pory wskaźnik czerwony, zostaje przez nieuwagę zniszczony. Od tej pory obiekt ten staje się dla nas niedostępny.

❺ Niezależnie, którym operatorem się posługujemy, pracujemy teraz na tym samym obiekcie – pokazywanym pierwotnie przez wskaźnik żółty.

❻ Nie potrzebujemy już obiektów więc kasujemy je. Najpierw ten pokazywany przez wskaźnik żółty.

Natomiast obiektu A pokazywanego pierwotnie przez wskaźnik czerwony nie można już nigdy skasować. Operator `delete` żąda bowiem pokazania mu wskaźnikiem, który to obiekt ma skasować. Tymczasem my tę informację straciliśmy w linijce ❹. Teraz na ten obiekt nie pokazuje **żaden** wskaźnik.

Gdybyśmy usunęli z tej linijki komentarz, to odbyłoby się tutaj katastrofalne, powtórne kasowanie obiektu już raz skasowanego. (Obiektu B).

Opisaną sytuację można przyrównać do takiego obrazka:

> Mały i psotny Jaś uwielbia przekłuwać szpilką napełnione gazem baloniki. Bierze wobec tego z domu dwa sznurki o kolorach żółtym i czerwonym ❶, idzie do ulicznego sprzedawcy w parku i kupuje dwa nowe (new) baloniki (obiekty typu `int`) ❷. Sprzedawca i Jaś przywiązują baloniki do sznurków. Baloniki są nierozróżnialne, bo mają ten sam kolor (nie mają nazw), ale Jaś ma do nich dostęp za pomocą sznurków – czerwonego i żółtego.

Jaś mógłby od razu swoje baloniki przekłuć, ale chce jeszcze się nimi chwilę pobawić. Maluje na nich liczby 100 i 200. ❸

Potem wpada na nierozważny pomysł: odwiązuje czerwony wskaźnik od balonika z liczbą 100 i dowiązuje go do balonika z liczbą 200. ❹ Balonik z liczbą 200 jest teraz na dwóch sznurkach: czerwonym i żółtym.
–A ten balonik z liczbą 100? No cóż, szybuje teraz ku przestworzom. Nieostrożny Jaś stracił go na zawsze. Nie może go już teraz przekłuć. Ostał mu się jeno sznur (czerwony i żółty) z jednym balonikiem.

Ciągnijmy dalej tę opowieść:

Jaś wyjmuje z kieszeni igłę i przekłuwa to, co jest na końcu sznurka żółtego. Balonik z hukiem pęka (`delete`) ❻. Uparty lub roztargniony Jaś chce teraz przekłuć to, co jest na końcu sznurka czerwonego. Tylko że drugi sznurek był przywiązany do tego samego balonika (właśnie przekłutego). Jaś dźga igłą w powietrze i w miejscu, gdzie do tej pory był właśnie przekłuty balonik, trafia na oko Małgosi. Horror !

Utrata kontaktu z obiektem uniemożliwia na skasowanie go. Jeśli jest to tylko jeden taki obiekt, to mała strata, niech sobie będzie nieskasowany. Jeśli jednak jest to sytuacja w programie, gdzie tworzy się i kasuje wiele obiektów (np. pętla, lub wielokrotnie wywoływana funkcja), wtedy nieskasowanych obiektów będzie bardzo dużo. Wyczerpie to przyznany nam obszar zapasu pamięci (free store) i już żadnych nowych obiektów nie będziemy mogli w naszym programie tworzyć.

8.10.3 Zapas pamięci to nie jest studnia bez dna

Może się okazać, że w pewnym momencie nasz obszar dostępnej pamięci się wyczerpie. Wówczas próba utworzenia nowego obiektu (np. typu `float`) za pomocą wyrażenia

```
(new float)
```

da nam w rezultacie nie adres do tego obiektu, ale 0 czyli NULL.

Jeśli więc w programie zamierzamy kreować dużo obiektów korzystając z zapasu pamięci – to musimy się spodziewać, że pamięć się w końcu wyczerpie. Trzeba się z tym liczyć i po prostu sprawdzać czy operacja się powiodła.

```
float * wsk ;
     wsk = new float[8192] ;          // kreacja tablicy o 8192
                                      // elementach typu float
     if(!wsk)                   // ←——czyli if(wsk == NULL)...
     {
          error("pamiec sie wyczerpala") ;
     }
```

Jest także inny sposób. W bibliotece standardowej C++ są do dyspozycji funkcje, które pozwalają nam zareagować na brak pamięci w zapasie. Po prostu możemy określić, która z naszych (własnych) funkcji ma się uruchomić w takim awaryjnym wypadku.

Oto przykładowy program:

```
#include <iostream.h>
#include <stdlib.h>                                          // ❶
#include <new.h>                                             // ❷
/*****************************************************/
void funkcja_alarmowa() ;                                    // ❸
long k ;                                                     // ❹
/*****************************************************/
main()
{
     set_new_handler(funkcja_alarmowa);                      // ❺

     for(k = 0 ; ; k++ )
     {
          new int ;                          // tworzenie obiektu  ❻
     }
}
/*****************************************************/
void funkcja_alarmowa()
{
     cout << "\n zabraklo pamieci przy k = "
          << k << " !\n" ;
     exit(1) ;                                               // ❼
}
```

Po wykonaniu na ekranie pojawi się na przykład taki napis

```
zabraklo pamieci przy k = 36972 !
```

Oczywiście liczba ta zależy od bieżącej sytuacji w pamięci komputera. Czasem mógł nam przydzielić większy zapas pamięci, czasem bardzo mały. Zależeć to może na przykład od tego, czy akurat w pamięci rezydują jakieś inne programy.

Kilka uwag

❶ Nowy plik nagłówkowy. Dotyczy on tej części biblioteki standardowej, gdzie jest funkcja `exit` kończąca działanie programu.

❷ W tym pliku nagłówkowym jest deklaracja funkcji `set_new_handler`, którą właśnie tu reklamuję.

❸ Deklaracja funkcji `funkcja_alarmowa`. Ta funkcja zostanie oddelegowana do zadziałania w momencie, gdy wyczerpie się zapas pamięci (free store).

❹ Obiekt typu `long` o nazwie `k` definiuję jako globalny. To po to, by był dostępny także w naszej `funkcji_alarmowej`.

❺ Tu jest właśnie moment poinformowania kompilatora kogo delegujemy do reakcji na wyczerpanie się pamięci. Inaczej mówimy – jest to instalacja nowego programu obsługi. Jak widać robi się to bardzo prosto – wywołując biblioteczną funkcję `set_new_handler` (ang: ustaw jako nowy program obsługi). W na-

wiasie jako argument jest nazwa funkcji. W jednym z następnych paragrafów dowiesz się, że nazwa funkcji jest jej adresem w pamięci. Zatem to ten adres właśnie podajemy funkcji bibliotecznej.

❻ Nieskończona pętla kreująca (tworząca) obiekty typu `int`. Wyniku operacji `new` nie podstawiamy nigdzie, bo obiekty te naprawdę nie są nam w tym przykładzie potrzebne - chodzi tu tylko o to, by całkowicie wyczerpać zapas pamięci.

❼ W momencie, gdy ten cel osiągniemy (wyczerpanie zapasu), automatycznie zostaje uruchomiona wyznaczona przez nas `funkcja_alarmowa`. Funkcja ta wypisze na ekranie wartość zmiennej globalnej `k` – w ten sposób będziemy wiedzieli przy którym z kolei obiekcie zapas się wyczerpał. (Funkcja ma dostęp do `k`, bo jest ono globalne - zdefiniowane poza wszelkimi funkcjami).

8.10.4 Porównanie starych i nowych sposobów

Jakiś czas temu Jerzy Stuhr zaśpiewał w Opolu kabaretową piosenkę „Śpiewać każdy może, trochę lepiej lub gorzej...", a kiedy skończył wzruszył ramionami i powiedział „Wielkie mecyje zaśpiewać w Opolu... "

To samo zapewne pomyśleli teraz programiści klasycznego C: w zasadzie operatory `new` i `delete` to przecież to samo, co dawne funkcje biblioteczne:

```
malloc()        new      // memory allocation
free()          delete   // free allocated memory
```

Czyli – „wielkie mecyje tworzyć obiekty!" Rzeczywiście. Problem tylko w słowach: „trochę lepiej lub gorzej". Wspomniane funkcje biblioteczne klasycznego C są nadal dostępne w C++, więc możesz sobie je dalej używać.

Co jednak przemawia za używaniem `new` i `delete` ?

Bardzo ważna cecha: Otóż jeśli kiedyś zdefiniujemy sobie nasz własny typ obiektu, (a będziemy to robić w następnych rozdziałach) to w momencie stwarzania (kreacji) pojedynczego obiektu danego typu – automatycznie może zostać wywołana nasza specjalna funkcja zwana konstruktorem. (O szczegółach mówić będziemy w osobnym rozdziale, str. 336). Przy zastosowaniu `malloc` tej akcji wywołania konstruktora oczywiście nie będzie.

Natomiast przy kasowaniu obiektu operatorem `delete` automatycznie wykona się inna nasza funkcja zwana destruktorem. Tego w klasycznym C nie mieliśmy.

Zapomnij więc o tamtych staromodnych sposobach.

Dodatkowo:

`malloc` zwraca wskaźnik do `void`, czyli wskaźnik do czegoś nieokreślonego. Natomiast `new` jest bezpieczniejszy, bo jako rezultat zwraca wskaźnik do typu, który właśnie stwarza. Dzięki temu nie możemy omyłkowo stworzyć operatorem `new` np. obiektu `float` a jego adres przypisać wskaźnikowi do `int`. Kompilator od razu nas uratuje przed nieszczęściem sygnalizując błąd.

Inny powód:

Gdy pracuję na komputerze klasy IBM PC z kompilatorem Borland C++

to lubię operatory `delete` *i* `new` *za to, że uniezależniają mnie od przyjętego tak zwanego modelu pamięci. (Zainteresowani wiedzą o czym mówię). Przy starym sposobie jeśli model pamięci był „small" to rezerwacje pamięci robiłem funkcjami bibliotecznymi:*

```
malloc(), free()
```

a jeśli był model „huge" to funkcjami:

```
farmalloc(),  farfree()
```

Dzięki operatorom `new` – `delete` *w ogóle nie muszę myśleć o tych modelach pamięci.*

Poznaliśmy już pobieżnie dziedziny, gdzie wskaźniki mogą się przydać. Porozmawiajmy teraz o samych wskaźnikach.

8.11 Stałe wskaźniki

Mówiliśmy niedawno o wskaźnikach do obiektów stałych. Są to wskaźniki, które pokazywanego obiektu nie mogą zmieniać. Traktują go jako obiekt stały. Sam obiekt, na który pokazują nie musi być rzeczywiście stały. Ważne jest to, że wskaźnik tak go traktuje.

Są jeszcze inne wskaźniki z przydomkiem `const`. Czy widziałeś kiedyś zwiedzając nieznane miasto stojący na ulicy wielki plan miasta, a na nim czerwoną strzałkę z napisem „TU STOISZ" ? Ta strzałka to właśnie stały wskaźnik. Na tej mapie pokazuje ona zawsze w to miejsce. Możesz pokazywać na tej mapie różne obiekty, różnymi wskaźnikami, ale nie tą strzałką. Jest ona dobrze przyklejona w obawie przed dowcipnisiami.

Ta strzałka to właśnie stały wskaźnik. Oto definicja takiego obiektu i ustawienie go obiekt:

```
int zoo ;
int * const wskaz = &zoo ;
```

> Stały wskaźnik to taki wskaźnik, który ustawia się raz i już od tej pory nigdy nie można go zmienić. Obrazowo można powiedzieć, że stały wskaźnik to **wskaźnik nieruchomy**. Zamrożony zostaje adres, który w nim jest zapisany.

Pomyślisz pewnie: „–O co ta cała sprawa? – bierzemy zwykły wskaźnik ustawiamy na jakiś obiekt i po prostu nigdy go nie przesuwamy!". Rzeczywiście, masz rację. Tyle, że za pomocą tego słówka `const` **zabezpieczamy się** przed ewentualnym nieuważnym przesunięciem wskaźnika. Gdybyśmy go chcieli przesunąć, to kompilator zasygnalizuje błąd. Gdyby nasz kolega z zespołu pracujący nad inną częścią programu i nie znający tej – próbował przez nieu-

wagę wskanik poruszyć – kompilator zaprotestuje dla naszego wspólnego dobra.

Przyjrzyjmy się bliżej powyższej definicji tego wskaźnika. Definicję tę czytamy od nazwy i posuwamy się w lewo:

`wskaz` jest to stały (`const`) wskaźnik (`*`) pokazujący na obiekty typu `int`.

Uwaga:

> Ponieważ jest to stały wskaźnik, należy już w trakcie definicji inicjalizować go, czyli nadać mu wartość początkową. (Po prostu ustawić go na jakiś adres). Można to zrobić tylko teraz albo nigdy.

Już linijkę później próba nadania mu jakiejś wartości będzie uznana za pogwałcenie zasady, że wskaźnik jest stały (nieruchomy). Nawet gdybyśmy chcieli wpisać do niego ten sam adres, który już ma. Nie można i koniec !

W naszym przykładzie wskaźnik jest inicjalizowany adresem obiektu `zoo`.

8.12 Stałe wskaźniki, a wskaźniki do stałych

Jaka jest zasadnicza różnica między wskaźnikami stałymi a wskaźnikami do stałych?

- Stały wskaźnik to taki, który **zawsze pokazuje na to samo**. Nie można nim poruszyć.
- Wskaźnik do stałego obiektu to taki wskaźnik, który **pokazywany obiekt uznaje za stały.** Nie może go więc modyfikować.

Te dwa typy wskaźników można ze sobą ożenić. Mamy wtedy stały (nieruchomy) wskaźnik do stałego (niezmiennego) obiektu. W definicji wystąpi dwa razy słowo `const`

```
const float * const p ;
```

definicję taką czytamy (znowu od prawej do lewej): `p` jest stałym (`const`) wskaźnikiem (`*`) pokazującym na obiekt typu `float` będący stałą (`const`).

Uściślijmy: będący stałą dla tego wskaźnika. Inny, zwykły wskaźnik pokazujący na ten sam obiekt może go zmieniać.

A oto przykłady użycia: najpierw przykład na stały (nieruchomy) wskaźnik:

```
int  a = 5 ,
     b = 100 ;
int *wa ;                          // zwyłky wskaźnik
int * const st_wsk = &a ;          // nieruchomy wskaźnik

wa = &a ;                          // ustaw wskaźnik na zmienną a
*wa = 1 ;                          // załadowanie 1 do zmiennej a
*st_wsk = 2 ;                      // załadowanie 2 do zmiennej a

//————————teraz próbujemy ruszyć oba wskaźniki

wa = &b ;                          // przestaw wskaźnik by pokazywał na zmienną b
```

```
st_wsk = & b;                    // błąd - bo to jest nieruchomy wskaźnik !!!
                                 // jest na zawsze ustawiony na zmienną a
```

A oto przykład ze wskaźnikiem pokazującym na stałą:

```
int x[4] = { 0, 1, 2, 3 } ;
int tmp ;
int *w ;                         // zwykły wskaźnik
const int * wsk_od _st ;         // wskaźnik do obiektu stałego. Nie musi
                                 // on być od razu ustawiany. Można
                                 // nim nawet potem poruszać

w = x ;                          // ustawienie obu wskaźników na początek tablicy
wsk_do_st = x ;

tmp = *w ;                       // odczytanie zerowego elementu tablicy
tmp = *wsk_do st ;               // jak wyżej

// ————przesunięcie obu wskaźników na następny element tablicy
w++ ;
wsk_do_st ++ ;          // poruszać nim wolno

// ————będziemy tam wpisywać
*w = 0 ;                         // wpisanie 0 do elementu x[1]
*wsk_do_st = 0 ;                 // ←——BŁĄD ! Ten wskaźnik traktuje
                                 // to, na co pokazuje, jako obiekt stały.
                                 // Za pomocą TEGO wskaźnika obiektu
                                 // modyfikowac nie wolno
```

A oto przykład na stały (**nieruchomy**) wskaźnik do **stałego** obiektu:

```
int  m = 6,
     n = 4,
     tmp ;
const int * const w = &m ;
                 // ponieważ jest to stały (nieruchomy) wskaźnik to musimy go
                 // od razu zainicjalizować adresem na jaki ma pokazywać

tmp = *w ;       // odczytanie wartości z obiektu pokazywanego
*w = 15 ;        // ←——BŁĄD ! - zapisać tam nie możemy. Wskaźnik
                 // traktuje przecież swoj obiekt jako stały.

w = &n ;         // ←——BŁĄD ! wskaźnik jest na dodatek nieruchomy,
                 // nie można nim pokazać na obiekt inny niż m
```

8.13 Strzał na oślep – Wskaźnik zawsze pokazuje na coś

Jedną z pułapek, w którą często się wpada jest zapomnienie nadania wskaźnikowi wartości początkowej. Inaczej mówiąc zapominamy go ustawić, by na coś pokazywał. To jest źle powiedziane, albowiem wskaźnik zawsze pokazuje na coś, nawet jeśli to coś nie jest niczym zamierzonym. Tak samo, jak drewniany wskaźnik do mapy pokazuje na coś nawet wtedy, gdy leży odłożony na boku. Powiedziałbym nawet drastyczniej: celuje na coś, jak leżący na boku pistolet.

Jeśli więc zapomnimy ustawić wskaźnik, to próba odczytania tego miejsca, na jakie pokazuje, da bezsensowne i przypadkowe rezultaty. Gorzej z zapisem. To tak, jakby leżący na boku pistolet nagle wystrzelił. Zapisując coś do takiego (przypadkowo pokazywanego) miejsca niszczymy je, a to jest zwykle fatalne w skutkach, chociaż błąd może objawić się dużo później.

Dla ciekawości podam, że wskaźnik, który jest zdefiniowany jako obiekt statyczny (to znaczy albo jest globalny, albo co prawda lokalny, ale za to z przydomkiem `static`) taki wskaźnik pierwotnie pokazuje na adres zerowy NULL[†]. Z takim wskaźnikiem pokazującym na NULL nie ma specjalnego ryzyka, bo operacje na takim szczególnym adresie komputer łatwo wykryje i ostrzeże nas.

Jednakże wskaźniki, które definiujemy jako obiekty automatyczne pokazują na całkowicie przypadkowe adresy.

> *Łatwo sobie ten fakt uprzytomnić. Przypomnijmy: Obiekty automatyczne (a wskaźnik to także* **obiekt** *– wszystko jedno czy w komputerze, czy wystrugany z drewna), zatem obiekty automatyczne są przecież tworzone na stosie, a zasada jest taka, że tworzonych na stosie obiektów komputer dla nas nie inicjalizuje. Są tam śmieci, dopóki o inicjalizację nie zatroszczy się sam programista.*

Dobra rada:

> Jeśli Twój program z niewiadomych przyczyn zawiesza komputer (IBM PC) lub powoduje tzw. crash programu (VAX) – czyli komputer odmawia dalszej pracy z tym programem wyrzucając nam na ekran – że tak powiem – protokół z sekcji zwłok programu (postmortem dump) to jest ogromne prawdopodobieństwo, że winne są jakieś nieustawione wskaźniki.

Powracając do naszej analogii – to wystrzeliliśmy z leżącego na boku pistoletu. Kula trafiła kogoś na oślep.

Oto jak prosto zrobić taki błąd:

```
void fun()
{
float a ;
float *x, *m ;              // def wskaźników bez nadania wart początkowej

     m = &a ;               // teraz ustawiamy wskaźnik m
     *m = 10.7 ;            // poprawne wpisanie liczby do obiektu a
     // -------- wskaźnika x nie ustawiliśmy na nic
     *x = 15.4 ;      // ←—— tragedia !
}
```

Ten wskaźnik `x` nie był ustawiony na nic sensownego, więc pokazywał na coś przypadkowego. Tutaj więc coś w pamięci komputera niszczymy.

†) Pamiętamy, że obiekty statyczne wstępnie inicjalizowane są zerami chyba, że sami je inicjalizujemy inną wartością

8.14 Sposoby ustawiania wskaźników

Skoro wskaźniki przed ich pierwszym użyciem powinny być ustawione - jak to zrobić?

Niektóre sposoby już poznaliśmy. Zbierzmy jednak wszystkie najważniejsze.

- Wskaźnik można ustawić tak, by pokazywał na jakiś obiekt wstawiając do niego adres wybranego obiektu

```
wsk = & obiekt ;
```

- Wskaźnik można ustawić również na to samo, na co pokazuje już inny wskaźnik. Jest to zwykła operacja przypisania wskaźników

```
wsk = inny_wskaznik ;
```

- Wskaźnik ustawia się na początek jakiejś tablicy podstawiając do niego jej adres. Skoro wiemy, że nazwa tablicy jest równocześnie adresem jej zerowego elementu, zatem w zapisie niepotrzebny jest operator &. Piszemy po prostu

```
wsk = tablica ;
```

- Wskaźnik może pokazywać także na funkcję. O wskaźnikach do funkcji będziemy mówić za chwilę (str. 206). Tam też dowiemy się, że nazwa funkcji to także jej adres, zatem i tu zbędny jest operator &

```
wskf = funkcja ;
```

- Operator `new` zwraca adres właśnie stworzonego nowego obiektu. Taki adres natychmiast wpisujemy do wskaźnika. Od tej pory wskaźnik pokazuje na ten nowy obiekt.

```
float *wsk ;
wsk = new float ;
```

- Wskaźnik można ustawić też tak, by pokazywał na jakieś konkretne miejsce w pamięci. Na przykład na jakiś adres znany nam z książki, choćby instrukcji obsługi jakiegoś układu sprzęgającego (interface). Tutaj trudniej podać przykład, bo bardzo zależy to od metody adresowania stosowanej w danym typie komputera.
  Weźmy jednak sytuację najprostszą – komórki w komputerze numerowane są po prostu od 0 w górę. Jeśli wówczas chcemy pokazać na adres 3007307 to wskaźnik ustawia się po prostu instrukcją:

```
wsk = 3007307 ;
```

Jeśli nasz komputer to komputer klasy IBM PC, a nasz kompilator to Borland C++, to do takiego ustawiania wskaźnika służy specjalna makrodefinicja MK_FP. Aby ustawić wskaźnik na takie miejsce w pamięci o współrzędnych: Segment `0xffa`, Offset = `0xfa` wykonujemy instrukcję

```
wsk = MK_FP(0xffa, 0xfa) ;
```

Nie przejmuj się jeśli tego ostatniego nie rozumiesz. To nie jest C++, tylko prywatne sprawy komputera IBM PC.

❖ Jeśli wskaźnik ma pokazywać na ciąg znaków (string) – można go ustawić w ten sposób.

```
wsk = "taki napis" ;
```

Ten sposób dopuszczalny jest tylko dla stringów. Nie jest to *kopiowanie* stringu. Tekst ten (string) istnieje przecież gdzieś w pamięci, (tam złożył go kompilator), a tą instrukcją ustawiamy tylko wskaźnik na to nieznane miejsce. Oczywiście nie można tego sposobu zastosować do tablic liczb.

```
int *wskint = { 1,2,3,4 } ;          // błąd !!!
```

8.15 Tablice wskaźników

Jak pamiętamy, tablica to ciąg zmiennych tego samego typu zajmujących ciągły obszar w pamięci. Jeśli mogą być tablice zawierające zmienne typu `int`, `float`, `char` itd. – to dlaczego nie miałoby być tablic, których elementami są wskaźniki, czyli adresy różnych miejsc w pamięci. Adresy to w końcu też jakieś liczby. Można je przechowywać w tablicach.

Tablica wskaźników do `float`

Oto przykład tablicy do przechowywania pięciu wskaźników. Wszystkie te wskaźniki służą do pokazywania na obiekty typu `float`

```
float *tabwsk[5] ;
```

Przeczytajmy tę definicję: Zaczynamy od nazwy

```
tabwsk
```

(i posuwamy się w prawo, bo operator `[]` jest mocniejszy od operatora `*` – czytamy więc:)

`[5]` – jest tablicą pięcioelementową

(teraz w lewo i napotykamy gwiazdkę)

`*` – wskaźników mogących pokazywać na obiekty typu `float`.

Tę samą definicję można by napisać tak:

```
float *(tabwsk[5]) ;
```

Użycie nawiasu okrągłego pokazuje wyraźniej kolejność (sposób) czytania definicji.

Tablica wskaźników do stringów

A oto definicja innej tablicy wskaźników. Jej elementami są wskaźniki mogące pokazywać na stringi.

```
char *miasta[6] ;
```

Wskaźniki te można ustawić tak, by pokazywały na jakieś stringi.

```
char *nazwy[6] = {
                "Krakow", "Berlin", "Paryz", "Oslo",
                "Los Angeles", "Compostella"
} ;
```

Elementami tej tablicy nie są – jak można by przypuszczać – stringi z nazwami miast. Są tam adresy tych miejsc w pamięci, gdzie kompilator umieścił sobie te stringi. Podobna konstrukcja z liczbami byłaby błędna:

```
int *wskint[4] = { 10,11,12,13 } ;     // !!!!
```

znaczyłoby to bowiem, że chcemy by wskaźnik będący pierwszym elementem tablicy pokazywał na adres 10 itd.

Dlaczego w wypadku stringów błędu nie ma? Na tej samej zasadzie, dla której konstrukcja

```
char *w = {"abcde"} ;
```

jest poprawna. Gdzieś w pamięci kompilator musiał umieścić sobie ten string.[†)] W momencie, gdy dochodzi do definicji i inicjalizacji wskaźnika, podstawia on adres tego stringu do wskaźnika.

Oto krótki program, w którym posługujemy się tablicą wskaźników:

```
#include <iostream.h>
/*****************************************************/
main()
{
char *stacja[] = {
          "Wansee", "Nikolassee", "Grunewald",
          "Westkreuz", "Charlotenburg",
          "Savigny Platz", "Zoologischer Garten" };
char *www[3] ;
int i ;

     for(i = 0 ; i < 7 ; i++){
          cout << "Stacja: "<< stacja[i] << endl ;

     }
     www[0] = stacja[2] ;
     www[1] = stacja[5] ;
     www[2] = "Taki tekst" ;

     cout << "Oto 3 elementy tablicy : \n"
          << www[0] << ", "
          << www[1] << ", "
          << www[2] << endl ;
}
```

†) Mimo, że nie żądaliśmy tego specjalnie – stringi są przechowywane jako obiekty statyczne.

8.16 Wariacje na temat stringów

O stringach (ciągach znaków) mówiliśmy już – jednak obecnie (kiedy już mamy za sobą wskaźniki) do sprawy tej wracamy raz jeszcze. Teraz jednak jesteśmy mądrzejsi. Wiemy na przykład, że jeśli do funkcji wysyłamy string (czyli tablicę znakową), to można w tej funkcji odebrać tę tablicę jako

- rzeczywiście tablicę

```
chr tab[]
```

lub

- jako wskaźnik do obiektów typu `char`

```
char * wsk
```

Posłużenie się wskaźnikiem da w efekcie funkcję, która może wykonywać się szybciej. Z drugiej strony jednak funkcja ze wskaźnikiem jest na pierwszy rzut oka mniej czytelna. Tak to zwykle bywa: coś za coś!

Oto przykład dwóch funkcji:

Obie drukują string przysłany do nich, ale żeby było ciekawiej drukują go tak, że po kolejnych znakach wstawiają pauzy. Zatem tekst `tornado` wyglądał będzie na ekranie tak:

```
t-o-r-n-a-d-o-
```

Oto program, w którym mamy realizację „tablicową" i „wskaźnikową" takiej funkcji. Funkcje te nazywają się odpowiednio `przedzielacz_tabl` oraz `przedzielacz_wsk`

```
#include <iostream.h>
void przedzielacz_tabl(char tab[]) ;
void przedzielacz_wsk(char *w) ;
/*****************************************************/
main()
{
char ostrzezenie[80] = { "Alarm trzeciego stopnia " } ;

     cout << "\n wersja tablicowa \n" ;
     przedzielacz_tabl(ostrzezenie);                    // ❶

     cout << "\n wersja wskaznikowa \n" ;
     przedzielacz_wsk(ostrzezenie);                     // ❶
}
/*****************************************************/
void przedzielacz_tabl(char tab[])                      // ❷
{
int i = 0 ;

     while(tab[i])
     {
          cout << tab[i++] << "-" ;
     }
}
/*****************************************************/
```

```
void przedzielacz_wsk(char *w)                                    // ❸
{
     while(*w)
     {
          cout << *(w++) << "-" ;
     }
}
/*****************************************************/
```

Po wykonaniu programu na ekranie zobaczymy

```
wersja tablicowa
A-l-a-r-m- -t-r-z-e-c-i-e-g-o- -s-t-o-p-n-i-a- -
 wersja wskaznikowa
A-l-a-r-m- -t-r-z-e-c-i-e-g-o- -s-t-o-p-n-i-a- -
```

❶ Dwa miejsca, gdzie te funkcje się wywołuje. Jak widać sposób przesłania tablicy znakowej jest identyczny w obu wypadkach.

❷ Realizacja tablicowa funkcji. Nie ma tu nic nadzwyczajnego. Definiujemy lokalny obiekt `i` po to, by mieć indeks do elementów tablicy. Potem następuje pętla `while`.

Najpierw sprawdza się co jest w elemencie `tab[i]`. Jeśli jest to cokolwiek innego niż znak NULL (bajt zerowy) kończący string, to następuje wypisanie tego znaku, a potem kreseczki -. Przy okazji wykonuje się postinkrementacja indeksu. To znaczy już po wydobyciu znaku z tablicy, indeks `i` jest zwiększany o 1.

Zauważ, że w tej funkcji dwukrotnie musi być liczona pozycja (adres) danego elementu tablicy w pamięci.

❸ Realizacja „wskaźnikowa" jest sprytniejsza. Przysłana do funkcji tablica znakowa (czyli właściwie adres jej początku) służy do inicjalizacji lokalnego wskaźnika. Pokazuje on na początek stringu.

Wewnątrz funkcji mamy znów pętlę `while`. Wykonuje się ona dotąd, dopóki znak wskazywany przez wskaźnik — czyli `(*w)` — jest różny od NULL. Wypis na ekran to znowu wydobycie z pamięci elementu, na który wskaźnik pokazuje. Przy okazji robiona jest postinkrementacja wskaźnika, czyli po spełnieniu swojej roli przesuwa się on na następną literę stringu.

Zauważ, że ani razu nie liczy się tu żmudnie pozycji adresu danej litery w pamięci. Bierzemy po prostu to, na co wskaźnik już pokazuje. Przejście do następnej litery jest tylko przesunięciem wskaźnika na sąsiada.

A oto jak wyglądałaby funkcja kopiująca string z tablicy do tablicy. W rozdziale o tablicach widzieliśmy realizację „tablicową". Teraz zobaczmy jak to będzie wyglądało w przypadku posłużenia się wskaźnikami.

Oto krótki program:

```
#include <iostream.h>
char * strcpy(char *cel, char *zrodlo) ;
/*****************************************************/
main()
```

```
{
char poziom[] = { "Poziom szumu w normie" } ;
char komunikat[80] ;

     strcpy(komunikat, poziom) ;                    // ❶
     cout << poziom << endl ;
     cout << komunikat << endl ;
}
/*****************************************************/
char * strcpy(char *cel, char *zrodlo)              // ❷
{
char *poczatek = cel ;                              // ❸

     while(*(cel++) = *(zrodlo++));                 // ❹
     return poczatek ;                              // ❺
}
/*****************************************************/
```

Po wykonaniu program drukuje

```
Poziom szumu w normie
Poziom szumu w normie
```

aby udowodnić, że istotnie skopiował string z jednej tablicy do drugiej.

Kilka wyjaśnień

❶ Wywołanie funkcji celem skopiowania stringu z tablicy `poziom` do tablicy `komunikat`.

❷ Definicja naszej funkcji. Czytamy ją: `strcpy` jest funkcją wywoływaną z dwoma argumentami - pierwszym: wskaźnikiem do tablicy znakowej `(char*)` i – drugim: wskaźnikiem do tablicy znakowej `(char*)`. Funkcja ta ma zwracać wskaźnik do tablicy znakowej `(char*)`

Jak widzimy w ❶ nasza funkcja została wywołana z dwoma argumentami aktualnymi: tablicami `poziom` i `komunikat`. Wysłano do funkcji nazwy tablic – czyli inaczej adresy ich początków.

Tymczasem funkcja `strcpy` definiuje sobie lokalne wskaźniki `zrodlo` i `cel`. Wskaźniki te są inicjalizowane przysłanymi adresami tablic. Pokazują więc one wstępnie na ich początki.

❸ Nie pytaj mnie teraz dlaczego, ale czasem okazuje się przydatne, by taka funkcja zwracała wskaźnik do tej tablicy, do której string zostaje wpisywany. Teraz jeszcze na tę tablicę pokazuje wskaźnik `cel`. Ponieważ jednak zamierzamy nim za chwilę poruszać, dlatego zapamiętujemy go sobie we wskaźniku `poczatek`. Później instrukcją `return` ❺ zwrócimy właśnie tę zapamiętaną wartość.

❹ Praca całej funkcji polega na wielokrotnym wykonywaniu wyrażenia

```
*cel = *zródlo
```

czyli kopiowaniu znaku pokazywanego wskaźnikiem `zrodlo` w miejsce pokazywane wskaźnikiem `cel`.

Przy okazji po robieniu tej akcji każemy oba te wskaźniki przesunąć na następne pozycje (następny element tablicy znakowej). To: „przy okazji" – to właśnie postinkrementacja zaznaczona za pomocą znaków ++ w wyrażeniu

```
( *(cel++) = *(zrodlo++) )
```

wyrażenie to jest przypisaniem, a więc jako całość ma wartość równą wartości przypisywanej. Czyli w naszym programie najpierw wartością tego wyrażenia będzie kod litery `'P'`, potem kod litery `'o'`, potem liter `'z'`, `'i'`, `'o'` i tak dalej, aż do znaku kończącego każdy string czyli NULL. Wtedy wartość tego wyrażenia będzie zero. Ponieważ wyrażenie to tkwi jako warunek pętli `while`, zatem wtedy właśnie pętla ta zostanie przerwana.

> A co jest właściwą treścią pętli? : NIC! Zauważ, że zaraz za warunkiem sprawdzanym przez `while` jest średnik, czyli pętla jest pusta. Mimo to jednak wykonuje dla nas pracę. Jak to się dzieje?

Mówiliśmy już kiedyś o tym, przypomnijmy jednak. Otóż pętla `while` zawsze najpierw sprawdza sobie warunek. Jeśli będzie on spełniony, to ewentualnie wykona treść pętli. Jednakże my jako warunek daliśmy jej skomplikowane wyrażenie, którego wynik (wartość) ma ostatecznie zadecydować czy robić pętlę czy nie. U nas to wyrażenie to jest przypisaniem - kopiowaniem znaku. Wyrażenie to musi zostać więc najpierw obliczone. Wartością przypisania jest kod kopiowanego znaku i on właśnie decyduje czy wykonać obieg pętli czy nie. Wielka decyzja to nie jest - jeśli tylko ten znak jest inny niż NULL, to warunek pętli jest spełniony.

Pętla `while` wie już czy ma wykonać swą właściwą treść czy nie. Treścią pętli jest średnik – czyli instrukcja pusta – ale to nic nie szkodzi, bo kopiowanie odbyło się przecież już w chwili sprawdzenia.

Uwaga:

> *Jeśli masz troskliwy kompilator, to w tym miejscu otrzymasz ostrzeżenie, że możliwe, iż wykonujesz tu niepoprawne przypisanie. Świadczy to dobrze o kompilatorze. Spodziewa on się tu najczęściej* **porównań** *typu*
>
> ```
> while(a == b) ...
> ```
>
> *natomiast u nas jest tylko jeden znak = oznaczający* **przypisanie**. *Kompilator na wszelki wypadek ostrzega nas wiedząc, że sytuacje z przypisaniem są tu raczej rzadkością. Powinniśmy jeszcze raz się upewnić czy naprawdę chcemy przypisywać, a nie porównywać. My jednak naprawdę chcemy tu przypisywać, więc na to ostrzeżenie nie reagujemy.*

Zauważ jeszcze jedną ciekawą rzecz. String, jak wiadomo, ma na końcu znak NULL. Poprawne skopiowanie stringu w inne miejsce wymaga oczywiście skopiowania także i tego ważnego znaku. Czy nasza funkcja to robi?

Załóżmy, że skopiowaliśmy ostatnią literę napisu źródłowego. Dzięki postinkrementacji wskaźnik `zrodlo` przeskoczył na następny znak i pokazuje na znak NULL. Pętla `while` przystępuje do ponownego obliczenia wyrażenia będącego warunkiem. Następuje więc kolejne przypisanie - czyli skopiowanie znaku NULL. Wartością tego wyrażenia przypisania jest teraz NULL, a więc w tym

momencie pętla przerywa się. Jednak znak NULL został już właśnie skopiowany.

Wróćmy do sprawy tego, co zwraca nasza funkcja `strcpy`.

Funkcja zwraca jakąś wartość, ale ostatecznie mogłaby też nic nie zwracać i być typu `void`. Byłoby to bez wpływu na kopiowanie stringu. Zwracana jest jednak wartość typu

```
char *
```

czyli wskaźnik do obiektu typu znakowego. Na przykład do stringu. Po co tak?

Jest taka tendencja w większości funkcji bibliotecznych zajmujących się stringami, aby funkcja zwróciła wskaźnik do miejsca, gdzie odbyło się kopiowanie. Co przez to zyskujemy?

Otóż teraz, aby po procesie kopiowania wykonać jakąś operację na tablicy `komunikat` – (np. wypisanie na ekran) – zamiast pisać 2 instrukcje

```
strcpy(komunikat, poziom);
cout << komunikat ;
```

wystarczy napisać

```
cout << (strcpy(komunikat, poziom) );
```

Jest to instrukcja „wypisz". Co ma zostać wypisane? To ukryte jest w wyrażeniu w nawiasie. Najpierw więc musi zostać obliczone wyrażenie czyli wykonywana jest funkcja `strcpy`. Zwracana przez nią wartość (wskaźnik do tablicy komunikat) jest właśnie wartością wyrażenia. Natomiast `cout` widząc wskaźnik typu `char*` rozumie to jako żądanie wypisania stringu znajdującego się pod wskazywanym adresem. A wszystko dlatego, że funkcja `strcpy` coś zwraca. Sprytne, prawda?

Oczywiście tak jest nie tylko w wypadku wypisywania na ekran. Jeśli np. mamy

```
char pierwszy[80] = { "hurra" }
char drugi [80] ;
char trzeci [80] ;
```

to zapis

```
strcpy(trzeci, strcpy(drugi, pierwszy) );
```

odpowiada temu samemu co

```
strcpy(drugi, pierwszy) ;
strcpy(trzeci, drugi);
```

W obu przypadkach najpierw string z tablicy `pierwszy` kopiowany jest do tablicy `drugi`, a następnie z tablicy `drugi` do tablicy `trzeci`. W rezultacie we wszystkich trzech tablicach będzie napis „hurra".

Mimo, że naszą funkcję wyposażyliśmy w typ zwracany `char*`, możemy ją używać na dwa powyższe sposoby. Nikt bowiem nie każe nam korzystać

z rezultatu zwracanego przez funkcję. Udajemy po prostu, że funkcja zwraca `void`.

Inne pożyteczne funkcje

Rozważmy teraz następujący przypadek

```
char stara[80] = { "zdanie trzecie" } ;
char nowa[80] ;

cout << (strcpy(nowa, stara+1) ) ;
```

Pytanie: Co zostanie wypisane na ekran?

Czyli inaczej – co będzie treścią tablicy `nowa`?

Zauważmy, że wysyłamy do funkcji `strcpy` nie tablicę `stara`, ale wyrażenie `(stara+1)`. Co jest wartością tego wyrażenia? To proste. Pamiętasz przecież naszą koronną zasadę, że nazwa tablicy jest wskaźnikiem do jej zerowego elementu? A pamiętasz, że dodanie liczby całkowitej `n` do wskaźnika powoduje, że rezultat pokazuje o `n` elementów dalej? Słowem wyrażenie `(stara+1)` jest adresem nie zerowego elementu tablicy, tylko pierwszego. Wskaźnik `stara` pokazywał na literę `'z'`, a wyrażenie `(stara+1)` pokazuje na literę `'d'`. Ten właśnie adres posyłany jest do funkcji `strcpy`.

Czy to jakoś przeszkadza funkcji `strcpy`? Skądże! Funkcja ta, niezależnie co jej przyślemy, rozpoczyna kopiowanie od tego miejsca w pamięci aż do napotkania NULL.

W sumie zatem do tablicy nowa zostanie skopiowany string

```
"danie trzecie"
```

bowiem pierwszą literę `'z'` przeskoczyliśmy.

`strcat`

Oto funkcja, która dopisze do jednego stringu drugi. Przykładowo jeśli przed operacją mieliśmy dwa stringi:

```
"Najpierw to "
"a teraz tamto"
```

po operacji będziemy mieli

```
"Najpierw to a teraz tamto"
```

Takie łączenie nazywa się konkatenacją, stąd nazwa funkcji `strcat`

Znowu prosty program:

```
#include <iostream.h>
char* strcat(char *cel, char *zrodlo) ;
/*****************************************************/
main()
{
char co[] = { "urzadzen sterowych" } ;
char komunikat[80] = { "Alarm :" };
```

```
        strcat(komunikat, co) ;
        cout << "po dopisaniu = "
             << komunikat << endl ;                    // ❶
        cout << (strcat(komunikat, ", o godz 17:12") ) ;//❷
}
/*****************************************************/
char * strcat(char *cel, char *zrodlo)                // ❸
{
char *poczatek = cel ;

        // przesunięcie napisu na koniec stringu
        while(*(cel++) );                             // ❹

        // teraz pokazuje o 1 znak za NULL
        cel--;                                        // ❺

        // to juz braliśmy przy strcpy
        while(*(cel++) = *(zrodlo++));                // ❻

        return poczatek ;
}
```

Po wykonaniu tego programu na ekranie pojawi się

```
po dopisaniu = Alarm :urzadzen sterowych
Alarm :urzadzen sterowych, o godz 17:12
```

Oto ciekawsze punkty programu:

❸ Jest to funkcja przyjmująca jako argumenty dwa wskaźniki do tablic znakowych. Zwraca także wskaźnik do tablicy znakowej. Z poprzednich stron wiemy już dlaczego.

❹ Aby dopisać coś do stringu powinniśmy wskaźnik pokazujący na jego początek przesunąć tak, by pokazał na koniec, czyli na kończący string znak NULL. Robimy to właśnie tą instrukcją.

❺ W poprzedniej instrukcji znaleźliśmy znak NULL, ale ponieważ instrukcja zawierała postinkrementację, więc wskaźnik `cel` pokazuje teraz na następny znak za znakiem NULL. Jest tam jakiś śmieć. Musimy się więc cofnąć wskaźnikiem tak, by pokazywał na znak NULL

❻ Jest to identyczny proces kopiowania jak w funkcji `strcpy`. Znaki zostają przepisywane tak, że pierwszy z nich zniszczy (zatrze) znak NULL kończący string „Alarm :". Kolejne znaki będą dopisywane począwszy od tego miejsca. W rezultacie otrzymamy wydłużony string, a na końcu oczywiście znajdzie się znak NULL.

❶ Na dowód, że to prawda wypisujemy to na ekranie.

❷ Do tablicy `komunikat` możemy dopisać jeszcze dalszy string. Tym razem nie jest on treścią tablicy, ale wstawiliśmy go bezpośrednio jako argument ograniczony cudzysłowami:

```
strcat(komunikat , "abc");
```

Czy można tak? Można – Kompilator bowiem bez naszego specjalnego żądania umieścił ten string gdzieś w pamięci komputera. (Stringi ujęte w cudzysłowy traktowane są tak, jakby były `static` – mają gdzieś swoje określone miejsce w pamięci – nawet jeśli tego miejsca nie znamy). Do funkcji zostanie więc wysłany adres tego specjalnego miejsca w pamięci, gdzie mieści się nasz string.

Nie można jednak zrobić następującej operacji

```
strcat("Uwaga :", "abc") ;                    // straszny błąd !
```

Oba stringi są wtedy gdzieś w pamięci. Z drugim wszystko jest w porządku, natomiast do tego pierwszego chcemy coś dopisać (ten drugi). Tymczasem na pierwszy string zarezerwowano 7+1 znaków i ani jeden więcej. Dalej teren nie należy już do nas. Dopisywanie coś do tego stringu będzie więc niszczeniem czegoś, co jest bezpośrednio za tym stringiem. To ma zwykle fatalne skutki.

8.17 Wskaźniki do funkcji

Jak wiemy wskaźnikiem można pokazywać na różne obiekty. Okazuje się, że logiczne jest także pokazanie na funkcję. To tak, jakbyśmy powiedzieli: a teraz masz wykonać *tę* funkcję.

Ostatecznie wskaźnik zawiera adres, więc czemu nie miałby to być adres tego miejsca w pamięci, gdzie zaczyna się kod będący instrukcjami żądanej funkcji. Oto przykład definicji takiego wskaźnika:

```
int (*wfun)() ;
```

Jak się taką deklarację czyta?

> Zaczynamy od nazwy. Następnie poruszamy się (o ile można) w prawo dlatego, że w po prawej stronie mogą stać tylko operatory `()` lub `[]` – a jak wiemy są one najsilniejsze z możliwych.
> Potem, gdy już się nie da w prawo (bo napotkaliśmy nawias zamykający), poruszamy się w lewo. Jeśli odczytaliśmy wszystko w obrębie danego nawiasu wychodzimy na zewnątrz niego i znowu zaczynamy w prawo.

A zatem naszą pierwszą definicję wskaźnika przeczytamy tak:

```
wfun
```

w prawo się nie da, bo jest nawias zamykający, więc idziemy w lewo -

```
(*wfun)
```

- **jest wskaźnikiem** – (załatwiliśmy całe wnętrze nawiasu ,więc wychodzimy na zewnątrz i poruszamy się w prawo, gdzie stoi bardzo mocny operator wywołania funkcji -

```
(*wfun)()
```

- **do funkcji wywoływanej bez żadnych argumentów** (nawias był pusty) – teraz już w lewo – **a zwracającą**

```
int (*wfun)()
```

wartość typu `int`

Bardzo ważne były tu nawiasy. Gdybyśmy je opuścili i napisali ostatnią definicję tak:

```
int *wf() ;
```

to byłaby to deklaracja funkcji (a nie wskaźnika do funkcji). Wedle powyższej reguły czytamy ten zapis tak:

```
wf()
```

`wf` jest funkcją wywoływaną bez żadnych argumentów, a zwracającą

```
* wf()
```

wskaźnik do

```
int * wf()
```

typu `int`

A zatem coś zupełnie innego. To dlatego, że nawiasy są silniejsze niż gwiazdka. Zobaczmy czym prędzej jak stosuje się to w praktyce. Oto przykład prostego programu:

```
#include <iostream.h>
int pierwsza();
int druga();                                                    // ❶
/*****************************************************/
main()
{
int i ;
int (*wskaz_fun)() ;                                            // ❷
     cout << "Na ktora funkcje ma pokazac wskaznik ?\n"
              "pierwsza -\t1 \nczy druga - \t2 \n"
              " napisz numer : ";
     cin >> i ;                                                 // ❸
     switch(i){
          case 1 :
               wskaz_fun = pierwsza ;                           // ❹
               break ;
          case 2 :
               wskaz_fun = druga ;
               break ;
          default :
               wskaz_fun = NULL;                                // ❺
               break ;
     }

     cout << "Wedlug rozkazu ! \n" ;

     if(wskaz_fun)                  // if not NULL              ❻
     {
          for(i = 0 ; i < 3 ; i++){
               (*wskaz_fun)() ;                                 // ❼
          }
     }
```

```
}
/*****************************************************/
int pierwsza()
{
     cout << "funkcja pierwsza ! \n" ;
     return 9 ;
}
/*****************************************************/
int druga()
{
     cout << "funkcja druga !\n" ;
     return 106 ;
}
```

Po wykonaniu na ekranie zobaczymy na przykład następujący wydruk

```
Na ktora funkcje ma pokazac wskaznik ?
pierwsza -    1
czy druga -   2
 napisz numer : 2
Wedlug rozkazu !
funkcja druga !
funkcja druga !
funkcja druga !
```

Kilka słów o tym programie:

❶ Deklaracje dwóch funkcji. Czytamy: funkcja `pierwsza` jest funkcją wywoływaną bez żadnych argumentów, która jako rezultat zwraca wartość typu `int`. Funkcja `druga` – tak samo.

❷ Definicja wskaźnika mogącego pokazywać na te wyżej zadeklarowane funkcje. To dlatego, że w myśl definicji (czytamy:) `wskaz_fun` jest wskaźnikiem do pokazywania na funkcje wywoływane bez żadnych argumentów, a zwracające jako rezultat wartość typu `int`.

❸ Pytamy użytkownika, którą funkcję chce wykonywać.

❹ Zależnie od odpowiedzi (stąd instrukcja `switch`) ustawiamy wskaźnik tak, by pokazywał na żądaną funkcję. W praktyce polega to na wpisaniu do niego adresu danej funkcji.
I oto natknęliśmy się na inną ważną zasadę:

Nazwa funkcji jest inaczej jej adresem w pamięci

Dzięki temu tak prosto operuje się wskaźnikiem do funkcji. Ustawienie go to po prostu instrukcja

wskaźnik = nazwa_funkcji ;

Zauważ, że nie ma tu żadnych nawiasów towarzyszących zwykle nazwie funkcji. Nawiasy takie rozumiemy jako „wywołaj funkcję o tej nazwie". Tym-

czasem my wcale nie chcemy jeszcze jej wywołać. Na razie tylko **o niej** mówimy. Umawiamy się ze wskaźnikiem, że ma na tę funkcję pokazywać.

Gdybyśmy zapomnieli i w omawianej linijce postawili nawiasy

```
wskaz_fun = pierwsza() ;                    // błąd !
```

Funkcja zrozumiałaby to jako zachętę do pracy: „Co? mówią o mnie z nawiasami ? – to znaczy, że mam ruszyć do akcji! ". Inaczej mówiąc ta linijka oznaczałaby coś takiego: wykonaj funkcję `pierwsza`, a jej rezultat wstaw do wskaźnika `wskaz_fun`.

Ryzyko jednak nie jest takie duże, bo kompilator nas sprawdzi i najprawdopodobniej do tego błędu nie dopuści. Wie on, że funkcja `pierwsza` ma zwrócić wartość typu `int`, a takiej wartości nie można przypisać (podstawić) do `wskaz_fun`, który spodziewa się adresu funkcji. Kompilator widząc tę niezgodność zaprotestuje.

> Zapamiętaj: Gdy mówisz komuś o funkcji – to używasz samej nazwy bez nawiasu i argumentów. Nawiasy są operatorem wywołania tej funkcji

❺ Jak wiemy do każdego typu wskaźnika możemy podstawić NULL. Nie oznacza to żadnej szczególnej funkcji, ale po prostu wygodnie się potem obecność NULL'a sprawdza.

❻ Jeśli nie dostaliśmy od użytkownika żadnej sensownej odpowiedzi – to do wskaźnika wstawiliśmy NULL. Oczywiście nie możemy nawet próbować uruchomić funkcji o takim adresie. Dlatego sprawdzamy: jeśli jest we wskaźniku coś innego niż NULL, to wtedy uruchomimy tak pokazaną funkcje. Jeśli NULL – to przeskakujemy ten fragment.

❼ Nie przerażaj się tą instrukcją. Porównaj dwie instrukcje:

```
pierwsza() ; // czyli to samo co: (pierwsza)() ;
```

oraz

```
(*wskaz_fun)() ;
```

Z dotychczasowych rozmów o wskaźnikach pamiętasz już, że zapis

**wskaźnik*

oznacza – „to, na co wskaźnik pokazuje". W naszym wypadku pokazuje on np. na funkcję `pierwsza`. Dlatego powyższe dwa zapisy są równoważne. Te dwa (w tym wypadku puste) nawiasy to oczywiście sygnał, że chcemy by funkcja wystartowała.

8.17.1 Ćwiczenia z definiowania wskaźników do funkcji

Nie ma nic trudnego we wskaźnikach do funkcji. Jeśli jednak początkujący programiści się ich boją, to powodem jest moim zdaniem zapis. Z tymi gwiazdkami i nawiasami trzeba się oswoić.

Uważam, że swobodę operowania wskaźnikami do funkcji nabyć można tylko wtedy, gdy umie się czytać ich definicje i takie definicje samemu pisać. Dlatego proponuję: skoro już wiemy jak czytać deklaracje (i definicje) wskaźników do funkcji, to

Spróbujmy sami napisać definicję wskaźnika do pokazywania na określony typ funkcji

Wiem, że jest to trochę nudne, ale obiecuję Ci, że jeśli nauczysz się teraz czytania i zapisu definicji wskaźników (także do funkcji) - to będziesz się zawsze czuł pewnie przy programowaniu w C++.

Wyobraźmy sobie, że mamy gdzieś funkcję

```
int muzyka() ;
```

która odgrywa melodyjkę. Chcemy teraz zdefiniować wskaźnik mogący pokazywać na taką funkcję. Wskaźnik ma się nazywać na przykład `www`.

Piszemy więc na środku linijki nazwę `www` i będziemy ją obudowywać dookoła.

Mówimy więc `www`

```
www
```

jest wskaźnikiem

```
(*www)
```

służącym do pokazywania na funkcję

```
(*www)()
```

zwracającą wartość typu `int`

```
int (*www)() ;
```

Gotowe! (czyli triumfalny średnik na końcu).

Jeśli się już taki wskaźnik ma, to na naszą funkcję `muzyka` ustawia się go choćby taką prostą instrukcją:

```
www = muzyka ;
```

A teraz inny wskaźnik. Ma się on nadawać do pokazywania na funkcję

```
float dzielenie(int, int);
```

Funkcja ta na przykład dzieli dwie liczby całkowite i zwraca nam rezultat dzielenia. Zbudujmy więc wskaźnik do niej. Niech się on nazywa `ddd`. Zatem mówimy `ddd` i zapisujemy

```
ddd
```

jest wskaźnikiem

```
(*ddd)
```

do funkcji wywoływanej z dwoma argumentami typu `int`

```
(*ddd)(int, int)
```

a zwracającej wartość typu `float`

```
float (*ddd)(int, int) ;
```

Gotowe!

Nie drzyj jeszcze tej książki! Wiem, że to wszystko jest suche, nudne i formalne, ale mam dla Ciebie teraz prezent. Podam Ci sposób...

Jak nie rozumiejąc niczego, napisać sobie definicję wskaźnika mogącego pokazywać na daną funkcję

Dajmy na to, że chodzi o wskaźnik mogący pokazać na taką funkcję

```
float funkcyjka(int, char);
```

Czyli na funkcję wywoływaną z dwoma argumentami (typu `int` oraz `char`), a zwracającą w rezultacie wykonania typ `float`.

Mój sposób polega na tym, że bierzemy deklarację tej funkcji i w tej deklaracji nazwę funkcji – zastępujemy ujętą w nawias nazwą wskaźnika z gwiazdką z przodu. Czyli dokonujemy takiej zamiany:

nazwa_funkcji ⟶ (*nazwa_wskaźnika)

W rezultacie otrzymujemy definicję wskaźnika mogącego pokazywać na daną funkcję. Jeśli tak zrobimy z naszą deklaracją, wówczas w rezultacie otrzymujemy zapis

```
float (*nazwa_wskaznika)(int, char);
```

który jest dokładnie tym, o co nam chodziło. Jest to poszukiwana definicja wskaźnika. Żeby się przekonać przeczytamy ten zapis.

`nazwa_wskaznika`	
`(*nazwa_wskaznika)`	*- jest wskaźnikiem*
`(*nazwa_wskaznika)(...)`	*- do funkcji*
`(*nazwa_wskaznika)(int, char)`	*- wywoływanej z dwoma argumentami*
`float (*nazwa_wskaznika)(int, char)`	*- a zwracającej typ* `float`

Sprytne, prawda? Można to zrobić zupełnie bezmyślnie! No, może prawie bezmyślnie. To dlatego, że potrzebna jest umiejętność przeczytania tego, cośmy zdefiniowali. Tak dla kontroli.

Podam Ci teraz bardzo ważną rzecz - sposób jak czytać skomplikowane deklaracje

Otóż zasada jest taka, że:

1) Zaczynamy czytanie od wygłoszenia nazwy, której deklarację czytamy.

2) Następnie od tej nazwy posuwamy się w prawo. W prawo dlatego, że tam mogą stać najmocniejsze operatory. Operator wywołania funkcji lub indeksowania tablicy. (Te operatory, jak pamiętamy z tablicy priorytetów, mają jeden za najwyższych priorytetów). To, co napotkamy tam, odczytujemy na głos.

3) Jeśli w prawo już nic nie ma, lub natkniemy się na zamykający nawias - wówczas zaczynamy czytanie w lewo. Czytanie w lewo kontynuujemy dotąd, dokąd wszystkiego nie przeczytamy, lub gdy nie natkniemy się na zamykający nas nawias.

4) Jeśli napotkamy taki nawias, to wychodzimy na zewnątrz i - będąc już na zewnątrz tego nawiasu znowu zaczynamy w prawo czyli wracamy do punktu 2)

Procedurę przeprowadzamy dopóki nie przeczytamy wszystkiego w tej deklaracji.

Jak "czytać" ?

Bardzo prosto. Jeśli na naszej drodze napotkamy znaczek

- `*` (gwiazdkę) - czytamy: jest wskaźnikiem mogącym pokazywać na...
- `(typ1, typ2)` - czytamy: jest funkcją wywoływaną z argumentami typ1, typ2 (tu czytamy typy będące w nawiasie), a zwracającą jako rezultat...
- `[n]` - czytamy: jest n-elementową tablicą

Dygresja:

Ze względu na fleksję w języku polskim - nie jest to do końca tak eleganckie. Czasem zamiast „jest wskaźnikiem" lepiej by było powiedzieć „będący wskaźnikiem". Sądzę jednak, że szybko się nauczysz jak to wypowiadać zgrabnie.

Wypróbujmy ten sposób czytając to, co bezmyślnie wyprodukowaliśmy niedawno, czyli nasz zapis

```
float (*amazonka)(int, char)
```

Jak widzisz teraz samą nazwę zmieniłem, by nas nie sugerowała. Czytamy:

Według punktu 1) mówimy na głos:

> amazonka

Według punktu 2) chcemy poruszać się w prawo, ale się nie da, bo od razu napotykamy ograniczający nas nawias. Zatem przechodzimy do punktu 3)

Według punktu 3) idziemy w lewo i napotykamy gwiazdkę, co oznacza „jest wskaźnikiem mogącym pokazywać na...". Mówimy to na głos i wobec tego nasza dotychczasowa wypowiedź to:

> amazonka jest wskaźnikiem mogącym pokazywać na

Dalej w lewo się nie da, bo napotykamy na zamykający nas nawias. Wychodzimy więc na zewnątrz tego nawiasu i zaczynamy znowu posuwać się w prawo. W prawo napotykamy zapis `(int, char)`, co upoważnia nas do powiedzenia

na głos: „funkcja wywoływana z 2 argumentami (typu `int` i char), a zwracająca jako rezultat..."

Tu polonista dostaje zawału serca, a spod kroplówki dochodzi jego jęk o tym, że poprawnie po polsku ma być:

> amazonka jest wskaźnikiem mogącym pokazywać na funkcję wywoływaną z dwoma argumentami (typu `int` i `char`), a zwracającą jako rezultat...

W prawo jest już tylko średnik, więc zmieniamy kierunek i czytamy w lewo. Tam jest tylko typ `float`, co czytamy na głos. To już koniec, więc w sumie powiedzieliśmy na głos:

> amazonka jest wskaźnikiem mogącym pokazywać na funkcję wywoływaną z dwoma argumentami (typu `int` i `char`), a zwracającą jako rezultat typ `float`.

Jest to dokładnie taki wskaźnik, o jaki nam chodziło. Zatem bezmyślny sposób okazuje się doskonały. O jego geniuszu przekonaliśmy się tylko dzięki temu, że nauczyliśmy się deklaracje czytać.

Oczywiście już chyba rozumiesz, że zastosowałem podstęp, bo przecież - gdy umiesz już czytać deklaracje - sposób nie jest bezmyślny.

Pisanie deklaracji jest tylko trochę trudniejsze niż ich czytanie. Różnica polega na tym, że najpierw mówimy na głos a potem to, co powiedzieliśmy, zapisujemy.

Jednak wydaje mi się, że łatwiej odczytywać niż zapisywać, więc lepiej użyć tego automatycznego sposobu do zapisu, a potem przeczytać by sprawdzić. Gdy nauczysz się już dobrze czytać, czyli zrozumiesz istotę tego czytania, to tym samym będziesz już umiał zapisywać.

Do zagadnienia tego wrócimy jeszcze na stronie 395.

Bardzo rzadko się zdarza by definicje wskaźników do funkcji były bardziej skomplikowane. Tak więc raczej „dla hecy" przytoczę taki wskaźnik – spróbuj go najpierw sam odczytać

```
int ( * (*fw)(int, char*) )[2] ;
```

Poddajesz się? – No to spróbujmy razem. Zaczynamy od środka, gdzie jest nazwa, a potem w prawo ile się da, potem w lewo ile się da, a jak się już nie da, to wychodzimy na zewnątrz nawiasu i kontynuujemy. Zatem

```
fw
```

(w prawo się już nie da, więc w lewo)

```
*fw
```

jest wskaźnikiem (w lewo się już nie da, więc wychodzimy na zewnątrz nawiasu i czytamy z prawej

```
(*fw)(int, char *)
```

do funkcji wywoływanej z 2 argumentami: typu `int` i typu `char*`, a zwracającej

```
( * (*fw)(int, char *) )
```

(dalej w prawo się już nie da, bo jest nawias, próbujemy w lewo, a tam jest * czyli czytamy:) **...wskaźnik do...** (wychodzimy na zewnątrz nawiasu i czytamy z prawej)

```
( * (*fw)(int, char *) )[2]
```

...dwuelementowej tablicy (w prawo się już nie da więc w lewo, a tam stoi tylko `int`)

```
int ( * (*fw)(int, char *) )[2]
```

...obiektów typu `int`.

W skrócie brzmi to tak: `fw` jest wskaźnikiem do funkcji wywoływanej z argumentami `(int, char)`, a zwracającej wskaźnik do dwuelementowej tablicy typu `int`.

W tym przykładzie chodziło mi bardziej o to, byś zobaczył **jak** próbować takie zagadki rozwikłać. Nie przejmuj się też jeśli Ci to nie poszło. W mojej codziennej praktyce nigdy nie wymyślałem tak skomplikowanych wskaźników do funkcji.

Zastrzeżenia

Należy podkreślić, że

> typ wskaźnika do funkcji musi się zgadzać z typem funkcji.

Jeśli mamy np. funkcję

```
char fun(float, int, int) ;
```

to nie możemy na nią pokazać wskaźnikiem

```
int (*wsk)()
```

Kompilator się na to nie zgodzi. Do pokazania na taką funkcję nadaje się tylko wskaźnik

```
char (*www)(float, int, int) ;
```

Dlaczego? Dlaczego wskaźniki nie są uniwersalne? Dlaczego nie jest to po prostu wskaźnik do funkcji i już?!
Tak nie jest – znowu dla naszego dobra. Kompilator musi wiedzieć jaki jest typ wskaźnika po to, by wykryć czy nie pomyliliśmy się przy wysłaniu argumentów. Przy wywołaniu funkcji za pomocą wskaźnika

```
www(3.14, 1,5);
```

kompilator sprawdza nas czy ta lista argumentów zgadza się z listą, która stoi przy definicji wskaźnika `www`. A czy ta lista zgadza się z listą oczekiwaną przez wskazywaną funkcję? Musi się zgadzać, bo inaczej kompilator odmówiłby ustawienia tego wskaźnika na tę funkcję. Po prostu nie dałoby się wykonać przypisania

```
wsk = fun ;          // Błąd – niezgodność typu
                     // wskaźnika i funkcji
```

```
www = fun ;          // O.K. - zgadzają się te typy
```

I jeszcze jedno: Na wskaźnikach do funkcji nie wolno robić operacji arytmetycznych. To oczywiście jest intuicyjnie wyczuwalne. Co bowiem miałoby znaczyć odjęcie od siebie dwóch wskaźników do funkcji? Bezsens.

Tyle do tej pory mówiliśmy o definicjach wskaźników, że mogłeś odnieść wrażenie, że to coś trudnego. Wręcz przeciwnie. Przypominają one poznane wcześniej wskaźniki do zwykłych obiektów. W szczególności trzeba przypomnieć, że tak, jak w przypadku każdego wskaźnika, i ten nie pokazuje na nic, dopóki nie wstawimy do niego adresu funkcji, na którą ma pokazywać.

Podsumujmy naszą dotychczasową wiedzę o wskaźnikach do funkcji:

> Podobnie jak w przypadku tablic nazwa funkcji jest równocześnie adresem jej początku (- czyli adresem miejsca w pamięci, gdzie zaczyna się kod odpowiadający instrukcjom tej funkcji).

Tę zasadę także proponuję przykleić sobie nad biurkiem. Przyda się ona jednak trochę rzadziej, bo wskaźnikami do funkcji nie posługujemy się aż tak często.

Jeśli zdefiniowaliśmy sobie wskaźnik

```
int (*wf)() ;
```

to instrukcja

```
wf = muzyka ;
```

sprawia, że od tej pory wskaźnik zaczyna pokazywać na funkcję `muzyka`. A zatem możemy tę funkcję wywołać teraz za pomocą jej prawdziwej nazwy lub za pomocą wskaźnika. Zauważ, że są dwa sposoby z użyciem wskaźnika, drugi jest wyraźnie czytelniejszy:

```
muzyka();          // za pomocą nazwy

(*wf)();           // za pomocą wskaźnika
wf();              // za pomocą wskaźnika
```

Wyrażenie `(*wf)` oznacza: „skocz do miejsca w pamięci, na które pokazuje wskaźnik – zapewniam, że jest tam funkcja" – a stojące dalej dwa nawiasy mówią: „proszę tę funkcję wykonać".

Gdyby jednak naprawdę tak miało wyglądać używanie wskaźników do funkcji, to korzyść nie byłaby duża.

Kiedy zatem tak naprawdę wskaźnik do funkcji może się przydać ?

W sytuacjach podobnych do tych, w których najczęściej używa się zwykłych wskaźników:

- Przy przesyłaniu argumentów do funkcji. Adres innej funkcji można też wysłać jako argument. To tak, jakbyśmy powiedzieli funkcji: tu masz

takie argumenty, a dodatkowo tu jeszcze adres funkcji, którą masz u siebie wykonać.

- Do tworzenia tablic ze wskaźników do funkcji ; w takiej tablicy mamy jakby listę działań i odtąd możemy mówić: – „a teraz wykonajmy funkcję numer 5".

O obu tych sprawach powiemy szerzej w następnych paragrafach.

8.17.2 Wskaźnik do funkcji jako argument innej funkcji

Wyobraź sobie taką sytuację: piszesz bardzo ogólną funkcję, która służy do jakiejś rozmowy z użytkownikiem. Funkcja zadaje pytanie, na które użytkownik odpowiada „tak" lub „nie". W trakcie, gdy użytkownik będzie się zastanawiał nad odpowiedzią, funkcja ma coś robić. To, co ma robić, przyślemy do niej jako argument. Inaczej - nazwę funkcji, która ma być „w międzyczasie" wykonywana – przysyłamy jako argument.

Oto przykład. Korzysta on z dodatkowych funkcji bibliotecznych dostępnych w kompilatorze Borland C++ (wersja 3.1) dla komputera klasy IBM PC. Jeśli posługujesz się innym kompilatorem, to niektóre z tych funkcji mogą mieć inne nazwy – nie są to bowiem funkcje standardowe. Jednak nie o to tu chodzi – mówimy tu tylko o tym, jak do funkcji wysyła się nazwę innej funkcji. Zresztą zobaczmy!

```
/*-----------------------------------------------------------------
Program zostal napisany z wykorzystaniem niektorych
funkcji bibliotecznych charakterystycznych dla
kompilatora
          Borland C++
-----------------------------------------------------------------*/

#include <iostream.h>          // dla cin, cout
#include <ctype.h>             // dla tolower                ❶
#include <conio.h>             // dla kbhit
#include <dos.h>               // dla sound, nosound, delay

int pytanie(char *pyt,void (*wskaznik_funkcji)() ) ;// ❷
void muzyczka() ;                                   // ❸
void wiatraczek() ;
void kurs() ;
/*****************************************************/
main()
{
int i ;
     cout << "Samolot gotowy \n" ;
     while(1)
     {
          i = pytanie("Czy mam juz startowac ?",
                                   muzyczka ) ;   // ❹
          if(i)
          {
               cout << "Uwaga, startujemy !\n" ;
               break ;
          }
```

```
            else
            {
                cout << "nie to czekam...\n " ;
            }
        }
        cout << "Lecimy...\n" ;
        switch(pytanie("Czy dodac gazu ? ",wiatraczek) )//❺
        {
            case 1 :
                cout << "Zrobione !\n" ;
                break ;
            case 0 :
                cout << "Nie zmieniam !\n" ;
                break ;
        }
        pytanie("dobrze sie leci, prawda ? ", kurs);    // ❻
    }
    /*****************************************************/
    int pytanie(char *pyt, void (*wskaznik_funkcji)() )
    {
    char c ;
        cout << pyt << endl;
        while(1)
        {
            (*wskaznik_funkcji)() ;                     // ❼
            cin >> c ;
            switch(tolower(c) )
            {
                case 't' :
                    return 1;
                case 'n' :
                    return 0 ;
                default :
                    cout<< "odpowiedz 't' lub 'n' \n" ;
                    break ;
            }
        }
    }
    /*****************************************************/
    void muzyczka()
    {
    int i ;
        while(!kbhit() )                                // ❽
        {
            for(i=100 ; i < 1200 ; i+=100)
            {
                sound(i) ;
                delay(250);
            }
        }
        nosound();
    }
    /*****************************************************/
    void wiatraczek()                                   // ❾
    {
    char t[] = {  '|', '\\',
```

```
                    '-', '/' };
int i ;

     while(!kbhit() )
     {
          cout << " " << t[(i++) % 4] << "\r";
          delay(200);
     }
}
/*****************************************************/
void kurs()
{
int i ;
     while(!kbhit() )
     {
          cout << "kurs " << (239 + ((i++) % 4))
                              << "...\r";
          delay(200);
     }
}
```

Trudno dokładnie pokazać to, co zobaczymy na ekranie, jednak w przybliżeniu będzie to wyglądało tak:

```
Samolot gotowy
Czy mam juz startowac ? T
Uwaga, startujemy !
Lecimy...
Czy dodac gazu ?
          \                    <- tu ciągle kręci się wiatraczek
T
Zrobione !
dobrze sie leci, prawda ?
kurs 240...                    <- aktualizowane informacje o kursie
N
```

Przyjrzyjmy się teraz ciekawszym miejscom tego programu

❶ Powiedziałem, że program jest napisany z użyciem pewnych funkcji bibliotecznych dostępnych dla kompilatora Borland C++. Jak pamiętamy, aby posłużyć się funkcją biblioteczną należy do programu włączyć (wstawić) plik nagłówkowy zawierający deklarację tej funkcji. W naszym wypadku jest to nawet kilka nagłówków. W komentarzach podałem jakie funkcje wymagają danego pliku.

❷ Deklaracja funkcji. Funkcja `pytanie` jest właśnie tą, która zawiera istotę naszego przykładu. Z deklaracji tej widzimy, że pierwszym argumentem jest string. (Tak prześlemy tekst pytania, które funkcja ma zadać). Natomiast drugi argument służy do przesyłania nazwy funkcji. Jest to wskaźnik do funkcji wywoływanej bez żadnego argumentu i zwracającej rezultat typu `void` (czyli nic nie zwracającej).

❸ Deklaracje trzech funkcji. To właśnie na te funkcje będziemy pokazywali wskaźnikiem.

❹ **Istota tego programu.** Wywołanie funkcji `pytanie`. Jako drugi argument wysyłamy jej nazwę funkcji - `muzyczka`. Zauważ, że nazwa jest bez nawiasów. To dlatego, że jedynie mówimy o funkcji `muzyczka`, a nie chcemy by właśnie teraz startowała.

❼ Oto co się dzieje wewnątrz funkcji `pytanie`. Do funkcji wysłaliśmy nazwę funkcji (np. `muzyczka`). Tymczasem funkcja definiuje sobie wskaźnik do funkcji i inicjalizuje go (przysłanym jako argument) adresem funkcji `muzyczka`. Czyli odpowiada to jakby instrukcji

```
void (*wskaznik_funkcji)() = muzyczka ;
```

aby wywołać teraz funkcję pokazywaną przez ten wskaźnik wystarczy napisać

```
(*wskaznik_funkcji)() ;
```

Wiem co pomyślałeś – że to prawie identyczne jak to powyżej – ależ oczywiście! Według zasady, że składnia instrukcji używającej wskaźnika przypomina składnię w jego definicji.

Dzięki tej wspaniałej zasadzie mniej się musimy uczyć. (A swoją drogą to także i tę zasadę napisz sobie nad biurkiem).

❽ Jest to jedna z tych funkcji „zabijających czas". Sama funkcja `muzyczka` nie ma w sobie nic specjalnego. Zwykła funkcja. Tyle, że jej nazwę ktoś komuś przesyłał. Jedyną interesującą rzeczą w tej funkcji jest nieskończona pętla przerywana w momencie, gdy użytkownik tylko dotknie klawiatury. To robi właśnie funkcja biblioteczna `kbhit`. †) Funkcja ta zwraca 0, gdy nikt nie nacisnął niczego na klawiaturze. Jeśli ktoś coś nacisnął, to zwraca 1, ale wcale nie interesuje ją co zostało naciśnięte. To odczyta dopiero instrukcja

```
cin >> c ;
```

w funkcji `pytanie`.

W funkcji `muzyczka` obecne są wywołania funkcji bibliotecznych:

- `sound` – uruchamiająca generator wytwarzający dźwięk o podanej w hercach wysokości,
- `nosound` – zatrzymująca generator dźwięku,
- `delay` – funkcja powodująca zwłokę czasową o żądanym czasie trwania (zadanym w milisekundach).

W sumie więc nasza funkcja muzyczka wygrywa pożal-się-Boże melodyjkę. Czas trwania jednego dźwięku: 250 milisekund.

❾ Funkcja `wiatraczek` zabawia użytkownika rysując kręcący się wiatraczek. Jest on robiony przez kolejne rysowanie w tym samym miejscu na ekranie znaków

```
|    \    –    /
```

Te znaki są umieszczone w tablicy znaków. (Zauważ jaki chwyt musiał zostać zastosowany, by można było umieścić tam \ (bekslesz)). Kolejne wypisywanie

†) keyboard hit – ang. uderzenie w klawiaturę

na ekran tych czterech znaków jest zrobione za pomocą operatora dzielenia modulo 4. Zmienna `i` cały czas rośnie, a mimo to indeks tablicy jest zawsze z przedziału 0 - 3

❺ Wywołanie funkcji `pytanie` z innym zabawiaczem - `wiatraczek`. To wywołanie jest trochę trudniejsze, bo zapakowane do instrukcji `switch`. Ciągle jednak zasada jest ta sama.

❻ Wywołanie funkcji `pytanie` z zabawiaczem `kurs`. Nie reagujemy tu wcale na odpowiedź.

Program nie jest napisany elegancko. Przykładowo jeśli wciśniesz literę 't', ale jeszcze nie wciśniesz ENTER to `muzyczka`, czy `wiatraczek` zatrzymują się. Oczywiście da się to zrobić lepiej, jednak z tym musimy zaczekać do rozdziału o operacjach wejścia i wyjścia. Tam poznamy wszystkie takie sztuczki.

Z tego paragrafu dowiedzieliśmy się, że wskaźnik do funkcji może być argumentem innej funkcji i jest to zupełnie zwyczajna sytuacja. Do tego stopnia, że argument ten może też mieć wartość domniemaną. Gdyby deklaracja naszej funkcji `pytanie` wyglądała tak:

```
int pytanie(char *pyt, void (*wskaznik_funkcji)() = muzyczka);
```

to możliwe by było takie wywołanie funkcji:

```
pytanie("tekscik");
```

co odpowiada jak wiadomo

```
pytanie("tekscik", muzyczka);
```

Drobna uwaga: deklaracja funkcji `pytanie` powinna być wówczas w programie o jedną linijkę niżej – gdyż kompilator powinien już w tym momencie znać deklarację funkcji `muzyczka`.

8.17.3 Tablica wskaźników do funkcji

Wiemy już, że w tablicach można przechowywać wskaźniki (czyli adresy) do jakichś obiektów. Można też sporządzić tablicę składającą się ze wskaźników do funkcji.

Oto przykład tablicy wskaźników do funkcji:

```
void (*(twf[5]) )() ;
```

Przeczytajmy tę definicję, jak zwykle zaczynając od środka, czyli od nazwy

`twf`	-	`twf`
[5]	-	...jest 5 elementową tablicą
`*`	-	...wskaźników...
`()`	-	...do funkcji wywoływanej bez żadnych argumentów...
`void`	-	...a zwracającą typ `void` (czyli nic).

Jeśli pamiętamy, że operator `[]` jest o wiele mocniejszy od operatora `*`, to tę samą definicję możemy napisać po prostu tak:

```
void (*twf[5]) () ;
```

Zastanówmy się teraz co naprawdę zdefiniowaliśmy i kiedy może się nam to przydać. Otóż mamy tablicę, w której możemy przechować wskaźniki pokazujące na jakieś wybrane funkcje naszego programu. Jest to jakby lista czynności, które można wykonywać. Możemy załadować taką tablicę, a potem komenderować: a teraz proszę wykonać funkcję trzecią, a teraz piątą.

Żeby było jaśniej pokażmy to na przykładzie. W programie tym występują trzy funkcje, na które pokazujemy wskaźnikami z tablicy. Dla ułatwienia stosujemy tu te same funkcje `wiatraczek`, `kurs` i `muzyczka`. Ciał tych funkcji dla skrócenia nie zamieszczam. Są takie same jak w poprzednim paragrafie.

```
#include <iostream.h>          // dla cin, cout
#include <ctype.h>             // dla tolower
#include <conio.h>             // dla kbhit
#include <dos.h>               // dla sound, nosound, delay
#include <stdlib.h>            // dla exit

void muzyczka() ;
void wiatraczek() ;
void kurs() ;
/*****************************************************/
main()
{
void (*twf[3])() = { muzyczka, wiatraczek, kurs } ; //❶
int co ;

     while(1)
     {
          cout << "Menu :"
               << "\t0 - muzyczka\n\t1 - wiatraczek \n\t"
               << "2 - kurs\n\t3 - koniec programu\n\n"
               << "podaj numer zadanej akcji :" ;
          cin >> co ;                                  // ❷
          switch(co)
          {
               case 0 :
               case 1 :
               case 2 :
                    (*twf[co])() ;                     // ❸
                    break ;
               case 3 :
                    exit(1) ;

               default:
                    break ;
          }
     }
}
/*****************************************************/
void muzyczka()
{    /* ... */ }
```

```
/*****************************************************/
void wiatraczek()
{    /* ... */ }
/*****************************************************/
void kurs()
{    /* ... */ }
```

Wygląd ekranu po wykonaniu tego programu zależy oczywiście od wyboru „opcji" naszego menu.

```
Menu :
     0 - muzyczka
     1 - wiatraczek
     2 - kurs
     3 - koniec programu

podaj numer zadanej akcji : 2
kurs 239...
kurs 240...
...........
Menu :
     0 - muzyczka
     1 - wiatraczek
     2 - kurs
     3 - koniec programu

podaj numer zadanej akcji : 1
          \                    <- kręci się wiatraczek
Menu :
     0 - muzyczka
     1 - wiatraczek
     2 - kurs
     3 - koniec programu

podaj numer zadanej akcji : 0
                               <- tutaj gra muzyczka
Menu :
     0 - muzyczka
     1 - wiatraczek
     2 - kurs
     3 - koniec programu

podaj numer zadanej akcji : 3
```

Uwagi :

❶ Definicja 3 elementowej tablicy wskaźników do funkcji (funkcji wywoływanych bez żadnych argumentów i zwracających `void`). Od razu inicjalizujemy tę tablicę tak, że wskaźniki pokazują na nasze funkcje. Zauważ, że w klamrze są tylko nazwy funkcji - bez nawiasów. Pamiętamy, że nazwa funkcji jest adresem jej początku.

❷ Menu wypisywane na ekranie daje możliwość wybrania żądanej akcji. W tablicy czekają już wskaźniki pokazujące na różne funkcje. Tutaj tylko prosimy o podanie numeru funkcji, którą mamy uruchomić.

❸ Skoro wybraliśmy numer funkcji, to pozostaje tylko ją uruchomić. Dzieje się to właśnie tą instrukcją. Znowu zaznaczam, że moment użycia wskaźnika przypomina składnię jego definicji (porównaj z ❶).

Jak widać dzięki tablicy wskaźników możemy wydać polecenia: jeśli tak – to wykonaj funkcję nr... Jest to jeden ze sposobów sporządzania menu.

Oczywiście są także inne sposoby sporządzania menu:

```
switch(co_robic)
{
     case 1 :
          muzyczka(); break ;
     case 2 :
          wiatraczek(); break ;
     case 3 :
          wiatraczek(); break ;
     default :
          break ;
}
```

Jednak dzięki tablicy wskaźników można na przykład w trakcie pracy programu zmienić jeden z nich tak, by pokazywał na inną funkcję. Czasem takie operacje są przydatne. Jeśli na przykład mamy w programie funkcję

```
void symfonia() ;
```

a nastąpi gdzieś w programie instrukcja

```
twf[0] = symfonia ;
```

to od tej pory po wybraniu wariantu `0` zamiast funkcji `muzyczka` będzie się wykonywała funkcja `symfonia`.

8.18 Argumenty z linii wywołania programu

Spotkałeś na pewno programy, które uruchamia się pisząc obok nazwy programu dodatkowo jakieś opcje. Jest to eleganckie rozwiązanie problemu przesłania parametrów do programu. Rozwiązanie takie pozwala też pisać programy, które wywołuje się tak, jakby były komendami systemu operacyjnego.

Wyobraź sobie, że napisałeś ładny program, który - w momencie, gdy się zaczyna - maluje na ekranie piękną kolorową planszę tytułową.

Pracując nad modyfikacjami takiego programu – wielokrotnie musisz go uruchamiać. Oczywiście za każdym razem najpierw plansza tytułowa. Jeśli uruchamiasz program dziesiątki razy, to w pewnym momencie ta plansza Cię zdenerwuje. Chciałbyś by nie pojawiała się wtedy, gdy tego nie chcesz.

Oczywiście jest rozwiązanie: program startuje i pyta „czy mam pokazać planszę?". Na odpowiedź „nie" przeskakuje wywołanie funkcji zajmującej się rysowaniem planszy. Jest to rozwiązanie, które nie jest żadnym rozwiązaniem. Uruchamiając teraz kilkadziesiąt razy program musisz kilkadziesiąt razy odpowiadać na pytanie „Czy mam...?" Święty by nie wytrzymał!

Potrzebna jest możliwość, by już w momencie uruchamiania programu móc przesłać do programu informację o tym, że nie życzymy sobie planszy.

Jak zatem przesłać do programu parametry ?

Sprawa jest bardzo prosta. Przykładowo jeśli nasz program nazywa się „pelikan", to wywołujemy go pisząc jego nazwę, a dalej kolejno parametry

```
pelikan param1 77.2
```

w naszym wypadku są to: string: "param1" i liczba 77.2

Wysłać parametry to jeszcze nie wszystko. Program powinien umieć je odebrać. To też nie jest trudne. Aby odebrać tak wysłane parametry, funkcję `main` musimy zapisać w taki sposób:

```
main(int argc, char *argv[])
{
      // ... normalna treść funkcji main
}
```

Jak widzimy przesłanie parametrów do programu polega na tym, że funkcja `main` dostaje w prezencie od systemu operacyjnego dwa argumenty. Pierwszy typu `int`, a drugi nieco bardziej skomplikowany... To, jakie im nadamy nazwy w naszym programie, zależy wyłącznie od nas. Zwyczajowo nazywają się:

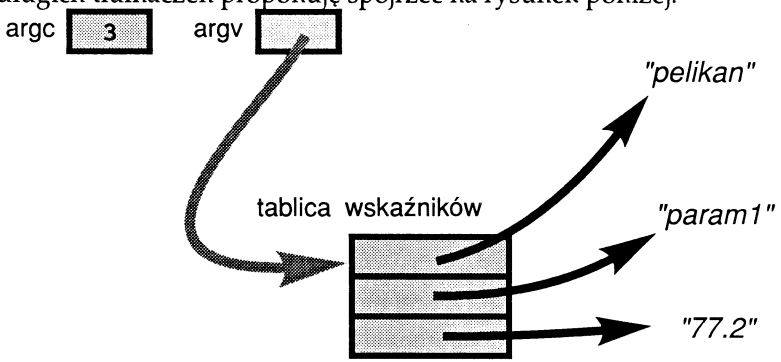
argc – od ang.: argument counter – licznik argumentów. Mówi nam ile parametrów system operacyjny wysłał do programu. Licznik ten ma co najmniej wartość 1. (Czy chcemy czy nie – system wysyła nam jako parametr nazwę programu, który właśnie uruchamiamy).

argv – od ang.: argument vector – tablica argumentów. Jest to wskaźnik do tablicy, w której poszczególnymi elementami są adresy stringów. Te stringi, to właśnie nasze kolejne parametry wywołania programu. Bardziej formalnie – zapis

```
char *argv[]
```

czyta się: `argv` jest tablicą wskaźników do (ciągów) znaków.

Zamiast długich tłumaczeń proponuję spojrzeć na rysunek poniżej.



Łatwo zauważyć, że poszczególne parametry są zapisane jako ciągi znaków pod następującymi adresami

```
*argv[0]          -          "pelikan"
*argv[1]          -          "param1"
*argv[2]          -          "77.2"
```

Poniższy przykład pokazuje jak to zrealizować w programie

```
#include <iostream.h>
#include <stdlib.h>

main(int argc, char * argv[])
{
     cout << "Wydruk parametrow wywolania :\n" ;

     for(int i = 0 ; i < argc ; i++)
     {
          cout << "Parametr nr "<< i
               << " to string: " << argv[i]
               << endl ;
     }

     /*——— zamienimy string na liczbę ———*/
     float x ;
     x = atof(argv[2]);
     x = x + 4;
     cout << "x = " << x << endl ;
}
```

Jeśli program ten wywołamy tak:

pelikan param1 77.2

to na ekranie zobaczymy

```
Wydruk parametrow wywolania :
Parametr nr 0 to string: pelikan
Parametr nr 1 to string: param1
Parametr nr 2 to string: 77.2
x = 81.2
```

Pamiętajmy jednak, że:

wszystkie parametry zostają przysłane jako stringi. Czyli parametr nr 2 przysłany zostaje nie jako liczba, ale jako ciąg takich znaków : cyfra 7, cyfra 7, kropka, cyfra 2, NULL.

Jeśli chcemy zamienić taki string na liczbę – możemy posłużyć się jedną z funkcji bibliotecznych. Nazywa się ona `atof` – od ang: Ascii TO Float. Deklaracja tej funkcji jest w pliku nagłówkowym `stdlib.h`

Widzimy, że w programie zamieniliśmy ten parametr na liczbę i złożyliśmy w zmiennej `x`, na której możemy już przeprowadzać operacje matematyczne.

W rozdziale o operacjach wejścia/wyjścia poznamy o wiele wygodniejsze sposoby odbierania przysłanych argumentów (str. 688).

9 Przeładowanie nazw funkcji

Są sytuacje, gdy nazwa funkcji doskonale określa akcję, którą wykonuje. Jeśli więc Czytelniku zamierzasz swoim funkcjom nadawać nazwy typu `X24a15c()` to możesz w ogóle nie czytać tego rozdziału.

9.1 Co to znaczy: przeładowanie

W języku angielskim przeładowanie (overloading) jakiegoś słowa oznacza, że ma ono więcej niż jedno znaczenie. Powiedzmy obrazowo: słowo jest przeładowane znaczeniami. Zjawisko to występuje także z nazwami funkcji w języku C++.

Na podstawie swoich doświadczeń z innymi językami programowania przywykłeś zapewne do faktu, iż w programie może być tylko jedna funkcja o danej nazwie. Używając tej nazwy mówiliśmy kompilatorowi o jaką funkcję nam w danym momencie chodzi. Gdybyśmy mieli w programie dwie funkcje o tej samej nazwie, to kompilator nie wiedziałby, którą z nich w danym momencie mamy na myśli, i którą z nich ma uruchomić.

Kompilator C++ jest inteligentniejszy. Wyobraź sobie takie dwie funkcje:

```
void wypisz_na_ekran(int);
void wypisz_na_ekran(char, float, char) ;
```

Pytanie: Gdybyś to Ty był kompilatorem C++ i napotkał w programie wywołanie

```
wypisz_na_ekran('A', 3.14, 'E');
```

to czy miałbyś jakieś wątpliwości o wywołanie której z dwóch powyższych funkcji chodzi?

Koń, jaki jest – każdy widzi! – mówi stara encyklopedia. Tę zasadę stosuje się także czasem w programowaniu. Otóż jeśli przyjąć zasadę, że funkcję rozpoznaje się nie tylko po jej nazwie, ale także po typie argumentów, to w pewnych

warunkach może istnieć więcej niż jedna funkcja o tej samej nazwie. Byle tylko te dwie funkcje różniły się argumentami.

To zjawisko nazywamy przeładowaniem nazwy funkcji. Uściślijmy:

> Przeładowanie nazwy funkcji polega na tym, że w danym zakresie ważności jest więcej niż jedna funkcja o takiej samej nazwie. To, która z nich zostaje w danym wypadku uaktywniona zależy od typu argumentów wywołania jej.

Funkcje takie mają tę samą nazwę, ale muszą się różnić liczbą lub typem poszczególnych argumentów. Znaczy to, że może być np. jedna taka funkcja z jednym argumentem typu `int`. Próba zdefiniowania w tym samym zakresie ważności drugiej takiej funkcji o tej samej nazwie i identycznym zestawie argumentów – czyli tutaj jedynym argumencie `int` – uznana zostanie za błąd.

Dla porządku trzeba dodać, że we wcześniejszych wersjach języka obowiązywało specjalne słowo kluczowe `overload` – ostrzegające kompilator, że zamierzamy daną nazwę funkcji przeładowywać. W nowszych wersjach języka to słowo nie jest już konieczne. Ze względów na zgodność jest jednak tolerowane. Zwykle jednak kompilator ostrzega, że jest ono staromodne.

Oto przykład programu z przeładowanymi funkcjami:

```
#include <iostream.h>
void wypisz(int liczba);                                    // ❶
void wypisz(char znak1, float x, char *tekst ) ;
void wypisz(int liczba, char znak) ;                        // ❷
void wypisz(char znak, int liczba) ;                        // ❸
/*****************************************************/
main()
{
        wypisz(12345);                                      // ❹
        wypisz(8, 'X');
        wypisz('D', 89.5, " stopni Celsiusza ");
        wypisz('M', 22);
}
/*****************************************************/
void wypisz(int liczba)
{
    cout << "Liczba typu int : " << liczba << endl ;
}
/*****************************************************/
void wypisz(char znak1, float x, char *tekst )
{
    cout << "Blok " << znak1 << " : "
        << x << tekst << endl ;
}
/*****************************************************/
void wypisz(int liczba, char znak)
{
    cout << znak << ") " << liczba << endl ;
}
/*****************************************************/
void wypisz(char znak, int liczba)
```

```
{
     cout << liczba << " razy wystapil stan "
          << znak << endl ;
}
```

Po wykonaniu tego programu na ekranie ujrzymy

```
Liczba typu int : 12345
X) 8
Blok D : 89.5 stopni Celsiusza
22 razy wystapil stan M
```

Komentarz:

❶ Program ten nie byłby niczym zajmującym, gdyby nie to, że posługujemy się w nim czterema funkcjami o identycznej nazwie. Widzimy tu deklaracje tych funkcji. Nazwa funkcji `wypisz` jest czterokrotnie przeładowania. Poszczególne funkcje różnią się typem i ilością argumentów.

❷ Zastrzeżenie o odmiennym typie argumentów dotyczy także kolejności. Mogą być dwie funkcje, które pracują na tym samym zestawie argumentów – porównaj z deklaracją funkcji ❸. Nie jest to nic dziwnego. W obu wypadkach chodzi o argumenty `int` oraz `char`, jednak ich inna kolejność sprawia, że funkcje są łatwo rozróżniane. Jest tylko jedna taka funkcja o nazwie `wypisz`, której pierwszy argument ma typ `int` a drugi `char`. Podobnie jest tylko jedna funkcja `wypisz`, której pierwszy argument jest typu `char` a drugi `int`.

❹ Wywołania funkcji `wypisz`. Kompilator przygląda się argumentom i stąd dobiera funkcje, do której one pasują. O tym, że przychodzi mu to łatwo, przekonuje nas to, co otrzymujemy na ekranie w rezultacie wykonania programu.

Przedstawiony przykład przekonał Cię chyba jak dziecinnie łatwe jest przeładowywanie nazw funkcji.

Tu chciałbym zrobić zastrzeżenie: co prawda to *nazwa* funkcji jest przeładowana, jednak często będziemy mówić, że to po prostu funkcja jest przeładowana.

Podsumujmy:

> Przeładowanie nazwy funkcji polega na nadaniu jej wielu znaczeń. Istnieje bowiem kilka funkcji o identycznej nazwie. To, która „wersja" funkcji jest uruchamiana zależy od kontekstu, w jakim została użyta – czyli od towarzyszących tej nazwie argumentów wywołania.

To tak, jak w życiu. Mamy funkcję „wywołaj". Nazwa wywołaj jest przeładowana znaczeniami. Powiedzenie: „wywołaj" z argumentem „ducha" rozumiane jest inaczej niż powiedzenie: „wywołaj" z argumentem „szefa z zebrania", a jeszcze co innego znaczy „wywołaj" z argumentem „film kolorowy". Nie ma jednak nieporozumień, bo kontekst jest jasny.

Słowa, słowa, słowa - uwaga do wydania czwartego

Pisząc w Niemczech tę książkę nie wiedziałem, że pewien polski autor, przetłumaczył termin *overloaded* – na polski jako "*przeciążony*". Dziś, wiedząc już o tym - nadal z całą świadomością pozostaję przy "przeładować". W literaturze możesz jednak spotkać także echa tamtego tłumaczenia.

Warto dodać, że na niemiecki przetłumaczono ten termin jako *überladen* (przeładować), a nie *überlasten* (przeciążyć - np. obwód elektryczny). Podobnie na francuski - przetłumaczono jako "*recharger*" (naładować od nowa). Na włoski ten termin przetłumaczono jako *sovraccaricare* - a słowo to nie oznacza wcale czegoś, co jest ciężkie. (Porównaj: *carica* - ładunek, *caricare* ładować [także broń]).

Te fakty potwierdzają słuszność tłumaczenia: "przeładować".

> **Chodzi przecież o to, by z danej nazwy zdjąć jedno znaczenie i naładować nowe.**

Niezależnie jednak od tego wszystkiego drogi Czytelniku - mów jak chcesz, byłeś tylko wiedział o przeładowanie czego i czym - tu chodzi.

Kiedy przeładowywać ?

Przeładowywać nazwę funkcji tylko dlatego, że wolno – byłoby głupotą. Jak to często bywa – i tego narzędzia należy używać z pewną logiką. W naszym poprzednim przykładzie przeładowaliśmy nazwę funkcji `wypisz` dlatego, że w różnych wariantach wykonywała ona analogiczną akcję na różnych zestawach obiektów. Zawsze chodziło na wypisanie czegoś na ekranie. Te cztery funkcje miały pewną cechę wspólną: wszystkie wypisywały coś na ekranie. Ta wspólna cecha jest najczęściej nazwą danego zestawu funkcji.

(Na przykład liczenie średniej z zadanych różnych obiektów, albo wyławianie wartości maksymalnej, albo sortowanie).

Można by też zapytać odwrotnie: kiedy **nie** przeładowywać nazwy funkcji? Odpowiedź jest prosta: Wtedy, gdy nie potrzebujemy tej samej nazwy dla różnych działań. Nie ma sensu nadawanie tej samej nazwy funkcji, która liczy logarytm, co funkcji wygrywającej melodyjkę.

Problem moim zdaniem nie jest poważny i nie ma ryzyka, że ktoś zechce przeładować wszystkie nazwy funkcji. Najczęściej nazwa funkcji określa istotę jej działania. To znaczy można spotkać nazwę funkcji

```
oblicz_srednia(...)
```

a prawie nigdy nazwę funkcji

```
x12_em4(...)
```

Dyktuje to zmysł praktyczności. Gdy więc w innym miejscu programu zechcemy liczyć średnią dla innych obiektów (np. nie dla liczb całkowitych tylko zespolonych), to wówczas przeładowanie nazwy funkcji nasunie się nam samo. Bez powodu takie skojarzenia i pomysły raczej nam nie grożą.

Myślę, że do tej pory nie udało mi się jeszcze namówić Cię na przeładowywanie nazw funkcji. Nie było to moim zamiarem. Tak naprawdę, to prawdziwe zastosowanie przeładowania nazw funkcji poznamy dopiero później, gdy mówić będziemy o definiowaniu swoich własnych typów.

9.2 Bliższe szczegóły przeładowania

Jak już wiemy, przeładowanie oznacza, że są w dwie (lub więcej) funkcje o identycznej nazwie, ale różniące się listą argumentów. Błędem by była próba definicji dwóch funkcji o identycznej nazwie i identycznej liście argumentów.

```
int rysuj(int aaa);
int rysuj(int zmienna);        // błąd - taka funkcja już jest
int rysuj(float x);            // o.k. - takiej jeszcze nie ma
int rysuj(int n, int m);       // o.k. - takiej też jeszcze nie było
```

Zwracam jednak uwagę, że to nie powtórna deklaracja wywoła błąd. Pamiętamy, że nawet bez żadnego przeładowywania – deklaracja (funkcji czy zmiennej) może wystąpić wielokrotnie. To nic nie przeszkadza. To tak, jakbyśmy kompilatorowi przypominali coś wielokrotnie. Coś, o czym on już dawno wie, a na te dalsze powtórzenia nie reaguje, sądząc że mamy sklerozę. O ile możliwe są wielokrotne deklaracje, o tyle definicja może być tylko jedna.

Zatem w wypadku naszych funkcji `rysuj` – kompilator nie zareaguje jeszcze przy powyższych deklaracjach. Zaprotestuje dopiero przy definicjach tych funkcji. Czyli tam, gdzie jest ciało (treść) tych funkcji. Konkretnie wtedy, gdy napotka drugą definicję funkcji o nazwie `rysuj`, a lista argumentów będzie taka, jaką już w innej definicji funkcji `rysuj` kiedyś napotkał.

Przy przeładowywaniu ważna jest tylko odmienność argumentów. Natomiast typ zwracany przez funkcję nie jest brany pod uwagę.

Zatem niepoprawna jest próba takiego przeładowania:

```
int akcja(int) ;
float akcja(int) ;                    // błąd !
```

W trakcie kompilacji takie przeładowanie uznane zostanie za błąd.

Można zdefiniować funkcje o identycznej nazwie i takim samym typie argumentów, pod warunkiem, że kolejność argumentów będzie inna. Poniższe funkcje są zatem poprawnym przeładowaniem

```
int fun(int, float) ;
int fun(float, int) ;
```

A teraz zagadka

Mamy taki zestaw przeładowanych funkcji:

```
void dru(int);
void dru(float);
void dru(int, int) ;
void dru(int, float) ;
```

a w programie występuje takie wywołanie funkcji:

```
dru(5, (int) 62.34);
```

Która z tych funkcji zostanie uruchomiona? Odpowiedź jest prosta: Pierwszy argument ma typ `int`. Drugi argument to wyrażenie, w którym zastosowano rzutowanie. Liczba 62.34 zamieniona jest operatorem rzutowania na liczbę typu `int` (czyli wartością wyrażenia jest 62). Wartością wyrażenia jest typ `int`, a zatem zostanie uruchomiona wersja

```
void dru(int, int);
```

Druga zagadka. Czy poprawne jest takie przeładowanie funkcji?

```
void zz(int) ;
void zz(unsigned int);
```

Tak, poprawne! Albowiem typ `unsigned int` oraz typ `int` to różne typy.

Co zrobić, gdy się nie da przeładować?

Może się tak zdarzyć, że dany zbiór argumentów wystąpił już w definicji funkcji o takiej samej nazwie. Jeśli mimo wszystko potrzebujemy powtórnie właśnie takiego zestawu argumentów, to oczywiście jeśli wystąpią one w identycznej kolejności – kompilator uzna to za błąd. Aby mimo wszystko móc taką nową funkcję zdefiniować należy rozważyć zmianę kolejności argumentów. To zwykle daje pozytywny efekt

```
void przeglad(float, float, char *);
void przeglad(float, char*, float);
```

Jeśli takie rozwiązanie nam nie odpowiada, należy rozważyć dodanie jakiegoś argumentu tak, aby lista stała się unikalna.

Przeładowanie w wypadku argumentów domniemanych

Wyobraźmy sobie taką sytuację. Mamy takie oto wersje przeładowanej funkcji `fun`:

```
void fun(float) ;
void fun(char *) ;
void fun(int, float = 0) ;
```

A oto wywołania funkcji, które kompilator musi dopasować:

```
fun(3.14);              // fun(float);
fun("napis");           // fun(char *);
fun(5);                 // fun(int, float = 0);
fun(5, 6.5);            // fun(int, float);
```

W komentarzu podana jest funkcja, którą wybierze kompilator.

Sprawa wygląda na oczywistą, jednak trzeba uważać. W powyższym zestawie przykładowych wersji funkcji nie może się znaleźć taka deklaracja:

```
void fun(int) ;
```

gdyż to powodowało by dwuznaczność. Wywołanie

```
fun(5);
```

pasuje bowiem jednakowo do obydwu poniższych deklaracji funkcji

```
void fun(int);
void fun(int, float = 0);
```

Nie ma tu żadnej preferencji wynikającej z faktu, iż skoro w wywołaniu jest jeden argument, to zapewne chodzi o funkcję z jednym argumentem. Preferencji takiej nie ma, bo sami z niej zrezygnowaliśmy. Kiedy? Wtedy, gdy zdecydowaliśmy, że drugi argument w deklaracji

```
void fun(int , float = 0);
```

jest domniemany. Kompilator rozumie to w ten sposób, że nadałeś tym samym tej funkcji identyczne prawo jak funkcji z jednym argumentem.

```
void fun(int) ;
```

Sprawę łatwo zapamiętać w momencie, gdy uświadomimy sobie, że deklaracja jednej funkcji z domniemanymi argumentami (w liczbie n) jest jakby równoznaczna n+1 deklaracjom, w których te argumenty w różnej liczbie występują; a więc deklaracja

```
void fun(int, float = 0);
```

(gdzie, jak widać, liczba domniemanych argumentów jest n=1) odpowiada takim n+1 = 2 deklaracjom

```
void fun(int, float)
void fun(int);
```

Tu masz odpowiedź na pytanie dlaczego do naszego zestawu funkcji przeładowanych nie da się dołączyć funkcji

```
void fun(int);
```

Po prostu dlatego, że taka deklaracja już tam jest. Co prawda zakamuflowana w deklaracji z argumentem domniemanych, ale od tej pory ten kamuflaż już Cię chyba nie zwiedzie.

9.3 Czy przeładowanie nazw funkcji jest techniką obiektowo orientowaną?

Jak widać, dzięki możliwości przeładowania nazwy funkcji mamy sytuację, w której sama maszyna decyduje, którą funkcję zastosować dla danego obiektu. Uwalnia to programistę od myślenia o szczegółach leksykalnych. Mówimy: wykonaj działanie na takim-a-takim obiekcie (liście obiektów), i wtedy uruchomiona zostaje funkcja właściwa dla danego obiektu.

Można by teraz od razu krzyknąć „Wreszcie narzędzie prawdziwie obiektowo orientowane! Hosanna! ".

Problem w tym, że różni ludzie różnie rozumieją granicę, gdzie naprawdę zaczyna się programowanie obiektowo orientowane. O tym jednak, przy innej okazji.

Zasłona spada

Czas by wyjawić jak to jest zrobione, że nazwa funkcji może być przeładowana, i że program (kompilator) orientuje się według obiektów będących argumentami funkcji. Będziesz pewnie rozczarowany, że to takie prymitywne, jednak sądzę, że musisz to rozczarowanie przeżyć. Wiedząc jak to jest naprawdę – łatwiej zrozumiesz dalszą część rozdziału.

Otóż: tak naprawdę, to te funkcje mają różne nazwy. Ty, co prawda, dałeś dwu funkcjom tę samą nazwę, jednak kompilator zmienia nazwy wszystkim funkcjom Twojego programu. Nie zapomina Twoich nazw, tylko je uzupełnia. Dopisuje do nich po prostu z jakimi argumentami jest ta funkcja.

Pokażmy zasadę tych zmian. Funkcja

```
void akcja(void);
```

otrzymuje *przykładowo* nazwę

```
akcja__Fv
```

`F` oznacza tu słowo funkcja, litera `v` oznacza `void` - pusta lista argumentów. Sposób oznaczania może być zależny od typu kompilatora.

Z kolei funkcja

`void akcja(int, float);`	*ma nazwę*	`akcja__Fif`
`void akcja(float, int);`	*ma nazwę*	`akcja__Ffi`

czyli jeśli są argumenty, to nazwy ich typów także doczepiane są do naszej nazwy. Zamiana ta zostaje zrobiona bez naszej wiedzy. Dotyczy ona zarówno definicji i deklaracji funkcji, jak też i wywołań funkcji. Zatem wywołanie

```
akcja(3.14, 100) ;
```

zostaje zastąpione wywołaniem

```
akcja__Ffi(3.14, 100);
```

Czyli w rezultacie w programie są teraz funkcje o zupełnie innych nazwach. To tutaj czar pryska. Nie ma już więcej programu „obiektowo orientowanego" — okazuje się, że kompilator zamienił go sobie na zwykły „klasyczny" program.

Rozumiemy teraz dlaczego dwie funkcje o identycznych nazwach muszą mieć inną listę argumentów. To gwarantuje kompilatorowi, że jeśli nawet trzon nazwy będzie ten sam, to przynajmniej doczepiony fragment opisujący argumenty – rozróżni te nazwy.

Dodatkowo rozumiemy dlaczego przeładowane funkcje nie mogą się różnić jedynie typem zwracanym: informacja o typie zwracanym nie jest doczepiana do nazwy. Nie da się więc takich funkcji rozróżnić.

9.4 Linkowanie z modułami z innych języków

Ważny jest fakt, że opisanej zmianie nazw podlegają wszystkie funkcje. Nie tylko te, które są przeładowane, ale naprawdę wszystkie.[†)]

W zasadzie o tym fakcie można by w ogóle nie myśleć – jest to w końcu prywatna sprawa kompilatora jak on sobie radzi ze swoją pracą. Niestety, tutaj jest pewien kłopot. Otóż jeśli masz program, na który składają się dwa moduły: jeden stary, dobrze chodzący, napisany i skompilowany w klasycznym C, a drugi moduł skompilowany kompilatorem C++, to podczas linkowania tych modułów w jeden program wyniknie problem:

> dostaniesz mianowicie komunikat, że linker nie odnajduje niektórych funkcji. Funkcji, o których wiesz na pewno, że tam przecież są !

Podajmy przykład takiej sytuacji. Załóżmy, że w „starym" module programu (tym z klasycznego C) jest funkcja

```
void mapa(int, float);
```

a Ty wywołujesz ją z modułu C++. Aby to było możliwe, musisz ją oczywiście wcześniej w tym module zadeklarować.

```
extern void mapa(int, float);
```

Gdy linkujesz taki program, otrzymujesz komunikat, że funkcja `mapa(int, float)` w ogóle nie istnieje. Dlaczego?

Powód jest bardzo prosty. Otóż w module C++ automatycznie powyższa deklaracja – a także właściwe wywołanie funkcji `mapa` –uległo zmianie nazwy. Deklaracja ta zmieniła się na taką:

```
extern void mapa__Fif(int, float)
```

W trakcie linkowania okazało się, że funkcji o nazwie `mapa__Fif` nie ma zdefiniowanej nigdzie. I słusznie, bo przecież w klasycznym C nie następują żadne zmiany nazwy. Tam funkcja nazywa się po prostu `mapa`.

Tak samo będzie jeśli nasz „stary" moduł jest w napisany asemblerze, Pascalu lub języku innym niż C++. Nie da się takich modułów zlinkować w jeden program.

Impas ?

Nie! Język C++ nic by nie był wart, gdyby nie pozwalał na łączenie z modułami pochodzącymi z C, asemblera czy innych języków programowania.
Oto wyjście: w module C++ należy zadeklarować funkcję w ten sposób

```
extern "C" void mapa(int, float);
```

†) Jest to zmiana w stosunku do wcześniejszych wersji języka C++

Litera C nie mówi, że to musi być koniecznie funkcja z klasycznego C. Mówi tylko, że **nie jest** to według konwencji C++. Czyli według takiej konwencji jak to jest np. w klasycznym C. Innymi słowy postawienie tam symbolu "C" jest jakby powiedzeniem: –Bardzo proszę nie robić mi żadnych kombinacji z nazwą tej funkcji, albowiem jest to funkcja, która została skompilowana bez modyfikacji nazwy !

Jeśli masz zadeklarować więcej takich funkcji, to możesz je umieścić w środku nawiasu klamrowego

```
extern "C" {
     pierwsza(int);
     druga(float, char*, int);
     // ...
}
```

W środku takiego nawiasu (czyli bloku) może się znaleźć nawet dyrektywa `include`.

```
extern "C" {
     #include "moje_deklar.h"
}
```

Wtedy wszystkie umieszczone we włączanym pliku deklaracje funkcji też traktowane są jak deklaracje funkcji w klasycznym C.

9.5 Przeładowanie a zakres ważności deklaracji funkcji

Definiując pojęcie „przeładowanie" powiedzieliśmy, że przeładowanie nazwy funkcji następuje wtedy, gdy w danym zakresie ważności jest więcej niż jedna funkcja o takiej samej nazwie. Nie rozwijaliśmy tego zastrzeżenia o identyczności zakresów ważności.

Pamiętamy, że:

> *najczęściej funkcje mają zakres ważności pliku, w którym je zdefiniowano. Czyli są znane w pliku od linijki ich deklaracji. Jeśli program składa się z kilku plików, to w tych innych plikach funkcja jest nieznana, dopóki nie zostanie tam zadeklarowana. Deklaracja może objąć zakres całego pliku, albo też mieć zakres mniejszy – lokalny.*

Oto przykład. Załóżmy, że mamy następujący plik:

```
#include <iostream.h>
/*****************************************************/
void dzwiek(int a)
{
     cout << a << " nuty \n" ;
}
/*****************************************************/
void dzwiek(float h)
{
     cout << "Dzwiek o czestotliwosci : "
          << h << " hercow \n" ;
}
/*****************************************************/
```

Zawiera on, jak widać, definicje dwóch funkcji o nazwie `dzwiek`. A oto inny plik, w którym korzystamy z tych funkcji:

```
#include <iostream.h>
extern void dzwiek(int);          // deklaracja o zasięgu pliku    ❶
/*****************************************************************/
main()
{
     dzwiek(1);                                                  // ❷
     {                                          // ←zakres lokalny  ❸
          extern void dzwiek(float);            // deklaracja lokalna ❹
          dzwiek(2) ;                                            // ❺
          dzwiek(3.14) ;                                         // ❻
     }                                                           // ❼
     dzwiek(5) ;                                                 // ❽
     dzwiek(6.28);                                               // ❾
}
```

Po zlinkowaniu tych plików i wykonaniu programu na ekranie zobaczymy tekst

```
1 nuty
Dzwiek o czestotliwosci : 2 hercow
Dzwiek o czestotliwosci : 3.14 hercow
5 nuty
6 nuty
```

Bliższe przyjrzenie się programowi upewnia nas, że żadne przeładowanie nie nastąpiło. Spodziewaliśmy się, że będzie wykonywana ta wersja funkcji dźwięk, która jest właściwa argumentom wywołania. Tymczasem tak się nie stało. Dlaczego ? Powód jest jeden. Deklaracje tych funkcji nie mają tego samego zakresu ważności. Zamiast przeładowania nastąpiło zasłonięcie.

Przyjrzyjmy się ciekawszym punktom programu

❶ Jest deklaracją o zasięgu pliku.

❷ Jest wywołaniem funkcji `dzwiek` - jedynej znanej w tym momencie, czyli tej zadeklarowanej powyżej.

❸ Otwierany jest jakiś blok lokalny. Może być to tak sztuczne jak u nas, a może być to po prostu wnętrze jakiejś funkcji.

❹ W tym lokalnym bloku deklarujemy, że istnieje gdzieś funkcja `dzwiek(float)`. Nazwa `dzwiek` zasłania wszystkie inne możliwe nazwy dzwiek z innych zakresów. Stają się niedostępne.

❺ Wywołanie funkcji `dzwiek` z argumentem typu `int`. Jedyną dostępną funkcją o nazwie dźwięk jest teraz funkcja `dzwiek(float)`. Tamta funkcja jest co prawda kompilatorowi znana, ale jej nazwa jest teraz zasłonięta. Zatem kompilator zamienia argument typu `int` na typ `float` i uruchamia tę jedyną możliwą teraz funkcję. Zamiana następuje przy użyciu tzw. konwersji standardowej.

❻ Wywołanie funkcji z argumentem typu `float`. Tu nie ma problemu. Takiego argumentu kompilator właśnie oczekiwał.

❼ Kończy się zakres lokalny. Deklaracja funkcji `dzwiek(float)` zostaje zapomniana i odsłania deklarację `dzwiek(int)`.

❽ Wywołanie teraz z argumentem typu `int` uruchamia po prostu tę jedyną możliwą teraz funkcję `dzwiek`.

❾ Natomiast wywołanie z argumentem typu `float`, to znowu dla kompilatora pewien kłopot. Funkcji o takim argumencie on już nie zna (deklarację lokalną przed chwilą zapomniał). Zamienia więc argument typu `float` na typ `int` i uruchamia jedyną możliwą funkcję o nazwie `dzwiek`.

Sformułujmy wniosek:

> Aby mieć rzeczywiście dwie funkcje o tej samej nazwie dostępne równocześnie (przeładowane) obie muszą mieć identyczny zakres ważności.

W naszym przykładzie tak by było, gdyby obie funkcje były zadeklarowane tak, by miały w zakres ważności pliku, czyli gdyby deklaracja ❹ była tam gdzie ❶

Trzeba jednak zaznaczyć, że klauzula o identyczności zakresu ważności przy przeładowaniu nie jest bynajmniej balastem, z którym trzeba nauczyć się żyć! Wręcz przeciwnie – otwiera nam drogę do lokalnego przeładowywania funkcji. W ramach jednego lokalnego obszaru funkcje mogą się przeładowywać, a nie ma to żadnego skutku wobec świata zewnętrznego, gdzie mogą być inne lokalne obszary, w których dokładnie ta sama nazwa funkcji może być również przeładowana. Jeden lokalny obszar nie wchodzi w kolizję z drugim.

Wiem, brzmi to może trochę zawile. Wszystko jednak stanie się jasne, gdy wkroczymy w krainę lokalnych obszarów jakimi są definicje klas, czyli typów, które możemy definiować sami.

9.6 Rozważania o identyczności lub odmienności typów argumentów

Wróćmy na poziom prostych spraw. Powiedzieliśmy, że przeładowanie funkcji jest możliwe wtedy, gdy funkcje różnią się listą argumentów. Funkcje mogą mieć tę samą nazwę - pod warunkiem, że argumenty będą inne.

9.6.1 Przeładowanie a typedef i enum

Przypominamy sobie zapewne deklarację `typedef` (str. 50). Wprowadza ona synonim dla jakiegoś istniejącego już typu. Nie tworzy ona nowego typu. Zatem poniższa próba przeładowania zostanie w czasie kompilacji uznana za błąd

```
typedef int calkow ;

void funkcja1(int) ;
void funkcja1(calkow);        // !
```

Ponieważ `calkow` jest tylko innym określeniem tego samego typu `int` dlatego mamy tu w gruncie rzeczy do czynienia z funkcjami

```
void funkcja1(int) ;
void funkcja1(int);           // !
```

Czyli są to dwie funkcje o tej samej nazwie i identycznej liście argumentów, a to jest błędem.

Zupełnie inna sprawa jest z typami wyliczeniowymi definiowanymi instrukcją `enum` (str. 52). Typ wyliczeniowy jest naprawdę odrębnym typem. Co prawda typ ten także bazuje na liczbach całkowitych, ale to nie ma znaczenia. (Przypominam, że typ `int` oraz `unsigned int`, to także odmienne typy).

Zatem instrukcją `enum` definiujemy nowy typ (inny od typu `int`) i nadajemy mu nazwę. Ta nazwa może być używana przy przeładowaniach funkcji.

```
enum operacja { pisz = 1, czytaj, skocz, przewin } ;

void tasma(int);
void tasma(operacja);
```

Jest to poprawne przeładowanie. Listy argumentów obu funkcji różnią się.

9.6.2 Tablica a wskaźnik

Zwróć uwagę na następujące dwie deklaracje funkcji

```
void fff(int tab[]) ;
void fff(int * wsk) ;
```

Obie funkcje pod kątem przeładowania uznawane są za identyczne i dlatego w danym zakresie ważności nie mogą mieć tej samej nazwy. Kompilator uzna to za błąd.

Łatwo to intuicyjnie wyczuć. Wyobraź sobie taki fragment programu:

```
int ta[10];                    // definicja tablicy
fff(ta);                       // wywołanie funkcji
```

Jako argument aktualny w wywołaniu funkcji znajduje się nazwa tablicy, czyli adres jej początku. Gdybyś był kompilatorem, to którą z wyżej zadeklarowanych funkcji powinieneś przy takim wywołaniu uruchomić?

Z rozdziału o wskaźnikach wiemy, że tablice i wskaźniki mogą być w zasadzie traktowane wymiennie. Zatem obie deklaracje funkcji jednakowo pasują do wywołania. A to jest błąd. Nie może być żadnej dwuznaczności.

Mówiąc bardziej formalnie: zarówno

```
int tab[]
```

jak i

```
int *wsk
```

mogą mieć te same inicjalizatory (tutaj – argumenty aktualne w wywołaniu funkcji). Zatem na podstawie wyglądu inicjalizatora nie można zdecydować do której wersji on się nadaje. Nadaje się bowiem do obu. Dwuznaczności być nie może i to właśnie powie Ci kompilator w informacji o błędzie.

Podsumujmy:

Typy argumentów różniące się tylko co do oznaczenia:
wskaźnik * ,
albo: tablica [ ]
- są uznawane przy przeładowaniu za identyczne.

Następne paragrafy mogą Cię trochę nudzić i może wydadzą się sformalizowane. Nie zniechęcaj się jednak. Jeśli czytasz tę książkę po raz pierwszy, to w pewnym momencie zdecydowanie odradzę Ci czytanie dalszej części tego rozdziału. Tymczasem postaraj się jednak mimo wszystko czytać najbliższych siedem paragrafów (są wyjątkowo krótkie), nawet jeśli nie wszystko wyda Ci się interesujące.

Będziemy tu nadal mówili o tym, kiedy argumenty deklarowanych funkcji pozwalają na przeładowanie, a kiedy nie.

Najmłodszym czytelnikom proponuję takie skojarzenia:

Zbiór wszystkich funkcji o jednej (przeładowanej) nazwie, to jakby menażeria zwierząt – po jednym okazie różnych gatunków. Argument wywołania funkcji to jakby pokarm dla zwierząt. Z kolei funkcje to same zwierzęta - niektóre są roślinożerne, niektóre mięsożerne.
Otóż jeśli chcemy by zwierzęta się nie pogryzły powinniśmy dawać zawsze taki pokarm, który może zjeść tylko jedno z nich. To jest oczywiste.
Niestety są zwierzęta, które mogą zjeść to samo co inne. W następnych paragrafach porozmawiamy właśnie o tym, jak rozpoznawać takie zwierzęta i unikać konfliktów.

Cała trudność w tych paragrafach polega na tym, że wielokrotnie pojawia się w nich słowo „inicjalizator". Umówmy się, że ile razy ja napiszę „inicjalizator", czy „argument wywołania funkcji" to Ty sobie myślisz: „pokarm podany zwierzętom".

9.6.3 Pewne szczegóły o tablicach wielowymiarowych

Mówiąc o tablicach wspomnieliśmy o sposobie przesyłania ich do funkcji. Przypominam – tablicy nie przesyła się przez wartość (bo kto by przesyłał do funkcji np. 8192 elementów tej tablicy), ale przesyła się ją przez adres. Nazwa tablicy, jak wiadomo według egipskich papirusów, jest adresem jej początku. Umieszczając w wywołaniu nazwę tablicy wysyłamy więc do funkcji jej adres.

- W obrębie funkcji tak otrzymany adres może posłużyć do inicjalizacji wskaźnika, którym będziemy się swobodnie poruszali po elementach tej tablicy.
- Innym sposobem odebrania przysłanego do funkcji adresu tablicy jest odebranie go jako tablicy.

To wszystko oczywiście przypomnienie, bo mówiliśmy o tym w poprzednim rozdziale, tutaj więc tylko 2 przykłady takich funkcji

```
int tablica[4][2] ;
//...
void funwsk(int *wsk);
void funtab(int t[4][2] );
```

Funkcje te mają inne nazwy, bo jak wiemy z poprzedniego paragrafu - typy **T*** i **T[ ]** są dla przeładowania nierozróżnialne. To dlatego, że oba mają identyczny inicjalizator (czyli tutaj: identyczny możliwy argument wywołania)

```
funwsk(tablica);
funtab(tablica);
```

Z kolei następujące dwie funkcje

```
void chap(int m[2][7]) ;
void chap(int m[3][2]) ;
```

mogą mieć tę samą nazwę funkcji – czyli mogą być przeładowane. To dlatego, że w wypadku wywołania obu funkcji jako argument aktualny musi wystąpić nazwa innej – odpowiedniej dla danego wywołania tablicy. Przykładowo:

```
int kil[2][7] ;
int zagiel[3][2] ;

chap(kil);                  // wywołanie chap(int m[2][7])
chap(zagiel) ;              // wywołanie chap(int m[3][2])
```

Ponieważ obie funkcje mają jako argument inny typ obiektu, dlatego mogą mieć inną nazwę.

I znowu zapytam: -Gdybyś był kompilatorem, to czy miałbyś tu wątpliwości, którą funkcję przy danym wywołaniu uruchomić? Da się tu bez dwuznaczności określić, która funkcja pasuje do danego argumentu.

Idźmy dalej. Pamiętasz z paragrafu o przesyłaniu do funkcji tablic wielowymiarowych (str. 147), że dla funkcji istotne są rozmiary tablicy, ale rozmiar pierwszy od lewej nie. Dopiero drugi i następne.

```
int tablica[4][5][1];      // pierwszy od lewej, to ten gdzie jest 4
```

To dlatego, że ten pierwszy od lewej rozmiar nie bierze udziału w obliczaniu pozycji elementu danej tablicy w pamięci komputera. (Zwykła arytmetyka). Dlatego więc jeśli w dwóch funkcjach (o tym samym zakresie) argument formalny będący tablicą różniłby się tylko tym wymiarem „pierwszym od lewej", to takie funkcje nie mogą mieć tej samej nazwy. Na przykład:

```
void abra(int x[5][10][2] ) ;
void abra(int y[9][10][2] ) ;  // błąd, bo identyczna jak powyższa
```

Także funkcje

```
void kadabra(int s[3]) ;
void kadabra(int m[8]) ;
```

Nie mogą mieć tej samej nazwy. Tablice będące ich argumentami formalnymi różnią się jedynie wymiarem pierwszym z lewej. To, że jest to jedyny wymiar — nie ma znaczenia. Przy obliczaniu pozycji elementu w pamięci nie bierze on udziału.

Podsumujmy:

Nie możemy mieć przeładowania funkcji w sytuacji, gdy dwie funkcje różnią się jedynie argumentem tablicowym tak, że tylko ten rozmiar najbardziej z lewej jest inny. Dla kompilatora jest to nierozróżnialne - dlatego przy definicji takich funkcji zaprotestuje.

9.6.4 Przeładowanie a referencja

Opiszemy tu kolejną sytuację, gdy przy przeładowaniu argumenty traktowane są jako identyczne. Oto ilustracja:

```
void ggg(int &k);
void ggg(int m);                    // za mało się różnią
// ...
int m ;
     ggg(m);
```

Jak widać, obie funkcje różnią się tylko tym, że jedna przyjmuje argument przez wartość, a druga przez referencję. Jest to błąd. Takie argumenty są pod kątem przeładowania identyczne.

Tu znowu bardzo łatwo wyczuć dlaczego. Zapytam znowu: –Gdybyś to Ty był kompilatorem i zobaczył takie wywołanie funkcji:

```
ggg(m);
```

to którą funkcję należałoby uruchomić? No właśnie – nie wiadomo którą. Wywołanie to pasuje jednakowo do jednej, jak i do drugiej funkcji. Następuje niedopuszczalna dwuznaczność.

Inaczej mówiąc – oba typy argumentu formalnego mają taki sam typ inicjalizatora. Sam zobacz inicjalizację referencji i obiektu:

```
int m ;                             // obiekt ;
int &ref = m ;                      // inicjalizacja referencji
int a = m ;                         // inicjalizacja innego obiektu
```

Z prawej strony znaku '=' stoi w obu wypadkach to samo. Zatem po inicjalizatorze (argumencie wywołania) rozstrzygnąć problemu nie można.

Podsumowanie:

Ponieważ dla dowolnego typu `T` – następujące deklaracje argumentów formalnych

```
T
T &
```

mają te same inicjalizatory, dlatego pod względem przeładowania nie może być funkcji o tej samej nazwie, a różniącej się jedynie tym, że jeden argument jest w jednej odbierany przez wartość, a w drugiej przez referencję.

Dygresja dla najmłodszych:

Okazuje się, że funkcje mające argumenty formalne `T` i `T &` to dwa gatunki zwierząt mogących zjeść to samo. Wypróbowaliśmy to właśnie – oba zjadają

„inicjalizator" (pokarm) będący obiektem typu `T`. (To nic, że potem trawią go inaczej). Aby uniknąć awantury, nigdy nie należy trzymać w menażerii dwóch takich zwierząt.

9.6.5 Identyczność typów: T, const T, volatile T

Inną sytuacją, gdy nastąpiłaby dwuznaczność jest

```
void f(int);
void f(const int m);          // dla przeładowania
void f(volatile int m) ;      // nierozróżnialne - błąd !
```

Każda z tych funkcji akceptuje taki sam argument aktualny (inicjalizator). W konsekwencji przy poniższym fragmencie programu:

```
int x = 6 ;
f(x) ;
```

Nie dało by się określić, o którą wersję funkcji chodzi. Wszystkie - jednakowo dobrze pasują do tego wywołania.

Wszystkie otrzymują tak samo argument przez wartość. Różnica między nimi polega tylko na tym, że:

- pierwsza funkcja jest zwykłą funkcją,
- druga –obiecuje, że swojej lokalnej kopii przesłanego argumentu nie będzie zmieniała,
- trzecia –obiecuje nie robić żadnych optymalizacji z lokalną kopią.

Te obiecanki są jednak prywatną sprawą funkcji. Każda z nich może zainicjalizować swój argument formalny tym samym inicjalizatorem.

Nasze wywołanie `f(x)` jest jednakowo dobre do wywołania każdej z nich. Następuje wieloznaczność, więc już w trakcie kompilacji wykryty zostanie błąd.

Podsumujmy:

Ponieważ dla dowolnego typu `T` –

zarówno	`T`
jak i	`const T`
jak i	`volatile T`

inicjalizuje się **identycznym inicjalizatorem**, dlatego tak określone argumenty formalne uznawane są za identyczne.

Dygresja dla najmłodszych:

Okazuje się, że trzy funkcje mające argumenty formalne typu `T`, `const T` i `volatile T` - to trzy gatunki zwierząt mogących zjeść to samo. Słowa `const` i `volatile` określają tu nie pokarm, ale to, co z niego jest już lokalnie w przewodzie pokarmowym bestii. To jednak nie ma żadnego znaczenia – ważne, że przed spożyciem był to pokarm typu `T`, o który wszystkie trzy zwierzaki pogryzły się nawzajem. Aby uniknąć awantury nie należy nigdy trzymać w menażerii choćby dwóch takich zwierząt.

9.6.6 Przeładowanie a typy: T*, volatile T*, const T*

Zagadka. Czy możliwe jest takie przeładowanie funkcji?

```
void radio(float *k);
void radio(const float *k);
void radio(volatile float *k);
```

Jeśli na podstawie poprzedniego paragrafu mówisz że nie, to przegrałeś. Jeśli nie przychodzi Ci do głowy żadna odpowiedź, to spróbuj pomyśleć, jak każdą z tych funkcji się wywołuje. Jeśli wywołanie pasować może do więcej niż jednej funkcji – to takie przeładowanie jest niemożliwe.

Co widzimy w pierwszej funkcji?

- Jest to deklaracja funkcji `radio`, którą to funkcję wywołuje się podając jako argument – adres obiektu typu `float`. Zatem na przykład tak:

```
float obj = 15.6 ;
radio(&obj);
```

Co widzimy w drugiej funkcji?

- Jest to deklaracja funkcji `radio` wywoływanej z argumentem będącym adresem obiektu, który jest stały `(const)`

```
const float pi = 3.14 ;
radio(&pi);
```

Zaś deklaracja trzeciej funkcji mówi:

- Jest to funkcja `radio` wywoływana z argumentem będącym adresem obiektu typu `float` – i to nie zwykłego obiektu, ale takiego, który ma cechę `volatile`.

```
volatile float wulkan ;
radio(&wulkan);
```

We wszystkich trzech wypadkach chodzi o różne typy obiektów. Kompilator napotykając na wywołanie funkcji z udziałem jednego z tych obiektów (jako argumentu wywołania funkcji) – wystarczy, że spojrzy na ten argument i już wie, do której z wersji funkcji `radio` on pasuje - a pasuje tu zawsze tylko do jednej.

Możesz jednak zapytać: – Jak to, przecież w poprzednim paragrafie mieliśmy podobną sytuację, a tam nie wolno było przeładowywać. Jak jest różnica?

Taka, że w poprzednim rozdziale słowa `const` i `volatile` określały lokalny obiekt tworzony w obrębie funkcji. Czyli kopię. Ten obiekt (kopia) może sobie być jaki chce – i tak do jego inicjalizacji nadawał się dowolny typ obiektu T. To powodowało wieloznaczność.

Tutaj jest odwrotnie:

> słowa `const` oraz `volatile` określają tu typ obiektu, który służy do inicjalizacji – czyli określają argument wywołania funkcji. (A nie jak poprzednio: lokalny obiekt inicjalizowany wewnątrz funkcji). Skoro aż tak precyzyjnie określiliśmy typ argumentu wywołania,

to znaczy, że tylko on może wywołać daną wersję funkcji. Przy tak precyzyjnym określeniu nie ma mowy o wieloznaczności.

Podsumujmy:

Ponieważ dla dowolnego typu `T` następujące deklaracje argumentów formalnych:

```
                T*
          const T*
oraz   volatile T*
```

wymagają każda odmiennego inicjalizatora (odmiennego argumentu wywołania) – dlatego funkcje różniące się tylko obecnością jednej z wersji wspomnianych deklaracji – mogą być przeładowane.

Zatem funkcje zadeklarowane na początku tego paragrafu są poprawnym przeładowaniem.

Dygresja dla najmłodszych:

Tu sprawa była inna. Słowa `const` i `volatile` stoją tak, że określają nie to, co lokalnie jest w przewodzie pokarmowym zwierzaka, ale są określeniem pokarmu – jaki ma być *przed* zjedzeniem. Okazuje się, że takie trzy zwierzaki mają ściśle sprecyzowane jaki pokarm zjadają. Jeden zwierzak je tylko befsztyki krwiste, drugi średnio przysmażone, a trzeci bardzo przysmażone. Jeśli znasz osobiście jakieś bestie żywiące się befsztykami, to wiesz, że jeśli taki jada średnio przysmażone, to brzydzi się ociekającymi krwią lub spalonymi na węgiel. Zatem nie ma konfliktu – po wyglądzie befsztyka kompilator łatwo zdecyduje komu go rzucić na pożarcie.

9.6.7 Przeładowanie a typy: T&, volatile T&, const T&

Dokładnie ta sama argumentacja, którą omówiliśmy w poprzednim paragrafie dotyczy różnych wersji referencji. Oto ilustracja. Funkcje:

```
void lad(int &m);
void lad(const int &m);
void lad(volatile int &m);
```

są wywoływane – każda z innym rodzajem obiektu

```
const int stala = 12 ;
volatile int elektron = 4 ;
int liczba = 1 ;

lad(stala) ;            // wywoła lad(const int &)
lad(elektron);          // wywoła lad(volatile int &)
lad(liczba) ;           // wywoła lad(int &)
```

albowiem słowa `const` i `volatile` określają typ inicjalizatora (czyli typ argumentu wywołania). Tak precyzyjne określenie inicjalizatora nie prowadzi do wieloznaczności zabójczej dla przeładowania.

Podsumujmy:

Ponieważ dla dowolnego typu T następujące deklaracje argumentów formalnych

```
          T &
    const T &
 volatile T &
```

wymagają każda odmiennego inicjalizatora (odmiennego argumentu wywołania) – dlatego funkcje różniące się tylko obecnością jednej z wersji wspomnianej deklaracji – mogą być przeładowane.

Dygresja dla najmłodszych:

Tłumaczenie jest takie jak poprzednio z tym, że teraz na sam befsztyk zwierzak mówi przezwiskiem. Nadal jednak 'to' ma być albo krwiste albo średnio przysmażone albo... Są to upodobania tak wykluczające się, że nigdy nie dojdzie do awantury. Jedzące to trzy zwierzaki mogą być trzymane w tej samej menażerii.

9.7 Adres funkcji przeładowanej

Jeśli w zwykłym przypadku chcemy się posłużyć wskaźnikiem do funkcji, to musimy go oczywiście w pewnym momencie ustawić tak, by na żądaną funkcję pokazywał. Oto ilustracja:

```
int sposob(float) ;
int sposob(char) ;          // <——— funkcja
//...
int (*wskfun)(char) ;       // deklaracja wskaźnika mogącego
                            // pokazywać na powyżej
                            // oznaczoną funkcję
     wsk = sposob ;
```

Jak wiemy nazwa funkcji jest też adresem początku tej funkcji, stąd też przy ostatniej instrukcji nie potrzeba operatora & (adres).

Co zrobić jeśli funkcja jest (wielokrotnie!) przeładowana? Ta sama nazwa oznacza wówczas różne wersje funkcji, a więc różne adresy.

Skąd kompilator będzie wiedział, o adres której wersji nam chodzi?

Kompilator patrzy wówczas na lewą stronę przypisania: w naszym wypadku stoi tam wskaźnik do funkcji i to nie byle jakiej funkcji, ale takiej, która jest wywoływana z argumentem typu `char`. Wszystko jasne! Kompilator, wśród przeładowanych wersji funkcji `sposob`, odszuka tę wersję, która ma argument typu `char` i adres tejże funkcji podstawi do wskaźnika.

Jeśli kiedyś zdefiniujemy wskaźnik

```
int (*wsk2)(float) ;
```

to instrukcja przypisania

```
wsk2 = sposob ;
```

(metodą identycznej dedukcji) sprawi, że do wskaźnika `wsk2` wstawiony zostanie adres wersji `int sposob(float)`. Sprytne, prawda ?

Zasada jest ciągle ta sama:

> w wypadku brania adresu funkcji przeładowanej – spośród wszystkich funkcji o tej samej nazwie (i ma się rozumieć – tym samym zakresie ważności) - do operacji wybierana jest ta wersja funkcji, która **dokładnie** pasuje do celu przypisania. Celem w wypadku przypisania jest wyrażenie stojące po lewej stronie znaku równości. To ono zdecydowało, który adres funkcji będzie użyty.

Wysyłanie do funkcji adresu innej, przeładowanej funkcji

Mogą jednak być inne sytuacje. Na przykład jeśli chcemy do jakiejś funkcji wysłać adres funkcji, która jest przeładowana. Oto przykład programu:

```
#include <iostream.h>
// ---deklaracje funkcji nazwy funkcji o przeładowanej nazwie- przelad
void przelad (int k) ;
void przelad (float x) ;

// -----deklaracje zwykłych funkcji
void pierwsza( void (*adrfun)(int) );
void druga( void (*adrfun)(float) ) ;
/*****************************************************/
main()
{
     pierwsza(przelad);                                  // ❶
     cout << "--------------------\n" ;
     druga(przelad);                                     // ❷
}
/*****************************************************/
void pierwsza( void (*adrfun)(int) )                     // ❸
{
     cout << "Jestem wewnatrz funkcji PIERWSZA\n"
            "teraz wywolam funkcje ktorej adres przyslano"
            " jako argument\n";
     adrfun(5);
     cout << "PO wywolaniu funkcji\n" ;
}
/*****************************************************/
void druga( void (*adrfun)(float) )
{
     cout << "Jestem wewnatrz funkcji DRUGA\n"
     "teraz wywolam funkcje ktorej adres przyslano"
     " jako argument\n";
     adrfun(3.14);
     cout << "PO wywolaniu funkcji\n" ;
}
/*****************************************************/
void przelad (int k)
{
     cout <<
          "*** Funkcja przelad - wersja: przelad(int) \n"
          " argument k = " << k << endl ;

}
/*****************************************************/
```

```
void przelad (float x)
{
     cout <<
      "*** Funkcja przelad - wersja: przelad(float) \n"
      " argument x = " << x << endl ;

}
```

Po wykonaniu tego programu na ekranie pojawi się

```
Jestem wewnatrz funkcji PIERWSZA
teraz wywolam funkcje ktorej adres przyslano jako argument
*** Funkcja przelad - wersja: przelad(int)
 argument k = 5
PO wywolaniu funkcji
---------------------
Jestem wewnatrz funkcji DRUGA
teraz wywolam funkcje ktorej adres przyslano jako argument
*** Funkcja przelad - wersja: przelad(float)
 argument x = 3.14
PO wywolaniu funkcji
```

Komentarz :

W `main` widzimy wywołania dwóch różnych funkcji – `pierwsza` i `druga`. Mają one jednak ten sam argument będący wskaźnikiem do funkcji. Skąd kompilator wie, o którą wersję przeładowanej funkcji `przelad` w konkretnym wypadku chodzi? (Czyli adres której funkcji `przelad` ma wysłać jako argument?)

❶ Przy wywołaniu funkcji `pierwsza` kompilator sprawdza jakiego wskaźnika do funkcji się `pierwsza` spodziewa. Z deklaracji, a także definicji ❸ orientuje się, że skoro celem przypisania jest wskaźnik do funkcji z argumentem typu `int` – to znaczy, że ma wysłać adres funkcji:

```
void przelad (int k) ;
```

bo to przecież tak, jakby robić przypisanie

```
void (*adrfun)(int) = przelad ;
```

❷ Przy wywołaniu funkcji `druga` sprawdza jaki jest cel przypisania wysłanego adresu funkcji. Celem jest wskaźnik do funkcji z argumentem typu `float`. Acha, to znaczy, że należy wysłać adres tej wersji:

```
void przelad (float x) ;
```

9.7.1 Zwrot rezultatu będącego adresem funkcji przeładowanej

Celem przypisania może być nie tylko argument formalny funkcji. Może być też typ wartości zwracanej przez funkcję. To znaczy w instrukcji `return` stawiamy przeładowaną nazwę funkcji, a kompilator decyduje, którą z wersji wybrać.

Decyduje na podstawie tego, jaki typ zadeklarowany jest jako typ zwracany przez bieżącą funkcję.

Jeśli funkcja ma na przykład zwrócić wskaźnik do takiej funkcji, która jest wywoływana z argumentem typu `int` – to wybrana zostanie wersja

```
void przelad(int);
```

Chciałem od razu napisać przykładową funkcję, która to właśnie robi, ale boję się, że się przerazisz. Zróbmy to więc etapami.

Funkcja będzie się nazywać `zwrot`. Po pierwsze ustalmy jaka to ma być funkcja. To ustalenie pomoże nam w późniejszym pisaniu deklaracji funkcji.

A zatem zwrot ma być funkcją wywoływaną bez żadnych argumentów, a zwracającą wskaźnik do takiej funkcji, która:

- - wywoływana jest z argumentem typu `int`
- - zwraca typ `void` (czyli nic)

Czyli ma to być wskaźnik mogący pokazać na choćby taką funkcję:

```
void f(int)
```

Korzystając z tych ustaleń zacznijmy budowanie deklaracji. Zatem:

`zwrot` jest funkcją wywoływaną bez żadnych argumentów...

```
zwrot(void)
```

...a która zwraca wskaźnik...

```
*zwrot(void)
```

...do funkcji...

```
(*zwrot(void)) (...)
```

... wywoływanej z argumentem typu `int`...

```
(*zwrot(void)) (int)
```

... i zwracającej typ `void`.

```
void (*zwrot(void)) (int) ;
```

Uff! Zrobione. Na pociechę powiem, że takie funkcje będziesz pisał bardzo rzadko. A na pewno nie na początku.

Skoro już mamy deklarację, to łatwo zapiszemy definicję funkcji – czyli zdefiniujemy ciało tej funkcji

```
void (*zwrot(void)) (int)
{
     cout << "zwracam wskaznik do funkcji ! \n" ;
     return przelad ;
}
```

Istota tego przykładu polega na tym, że obok słowa `return` stoi nazwa funkcji – nazwa, która jak nam wiadomo jest przeładowana. Jest więc kilka funkcji o nazwie `przelad`. O tym, którą z wersji wybierze kompilator, decyduje dek-

laracja funkcji. W deklaracji bowiem jest jasno powiedziane, że życzymy sobie, by został zwrócony adres do funkcji, która jest wywoływana z jedynym argumentem typu `int`. To, co sobie zażyczyliśmy otrzymać, jest jakby celem przypisania. Do tego celu kompilator dobierze pasującą wersję funkcji `przelad`.

Wniosek z tego paragrafu jest taki:

> Przy operacjach zwracania przez funkcję F adresu funkcji przeładowanej P – zwracany zostaje adres tej wersji funkcji przeładowanej, która pasuje do celu przypisania. Celem jest w tym przypadku zadeklarowany typ zwracany przez funkcję F

Opisane przypadki nie wyczerpują wszystkich możliwych celów. Dalszymi takimi sytuacjami zajmiemy się później. (Dla wtajemniczonych: Innym możliwym celem może być także inicjalizowany obiekt, a także argument formalny operatora).

9.8 Kulisy dopasowywania argumentów do funkcji przeładowanych

Powtórzę znowu: wiemy już, że nazwa funkcji może być przeładowana, czyli że może być kilka funkcji o tej nazwie, byle tylko różne wersje tej funkcji różniły się listą argumentów. Kiedy wywołujemy taką funkcję, kompilator patrzy na argumenty wywołania funkcji i - zależnie od ich typu - wybiera tę jedyną funkcję, do której dokładnie pasują.

Tak jest w pierwszym przybliżeniu. Zastanówmy się jednak co się zdarzy, gdy argumenty nie będą takie, że dokładnie pasują do jednej z wersji. Dajmy na to, że pasują prawie zupełnie, gdyby nie jakiś drobny szczegół.

```
void f(float);          // mamy taki zestaw
void f(char) ;
// a wywołujemy-----------------------------------
f(4) ;                  // mimo, że przecież   void f(int)
                        // nie istnieje
```

Co w takiej sytuacji ma zrobić kompilator? Oto dwa warianty tego co kompilator sobie pomyśli:

- wariant 1) :

  -Czy on oszalał? Wywołuje mi funkcję w sposób, który nie pasuje do żadnej z funkcji o tej nazwie. Trudno! Sygnalizuję błąd kompilacji, a on niech się wreszcie nauczy programować!

- wariant 2) :

  -No tak, co prawda wywołanie nie pasuje do żadnej z zadeklarowanych funkcji o tej nazwie, ale z tego co widzę, to najbliższe to jest takiej-a-takiej wersji. Wszystkie inne wersje różnią się o wiele bardziej. Programista zrobił to dlatego, że nie umie jeszcze dobrze programować w C++. Może jednak dlatego, że sądzi iż ja, kompilator, jestem na tyle inteligentny, że jed-

noznacznie domyślę się, o którą wersję może w tym lekko niepasującym wypadku chodzić.

Ponieważ kompilator C++ rzeczywiście ma ambicję, dlatego rozumuje według wariantu 2. W naszym przypadku nie znajdując funkcji

```
void f(int);
```

zamieni 4 na 4.0 i uruchomi funkcję

```
void f(float);
```

Dalsza część tego rozdziału poświęcona jest szczegółom rozumowania, na podstawie którego kompilator ustala, która z wersji przeładowanej funkcji pasuje najbardziej do argumentów wywołania.

Jeśli nie potrafi tego ustalić jednoznacznie, bo na przykład są dwa warianty, które mogą być jednakowo prawdopodobne, to dopiero wtedy sygnalizuje błąd. Jeśli zauważy, iż jeden z wariantów pasuje wyraźnie lepiej niż inne, wówczas ten właśnie zostanie wybrany.

Przy pierwszym czytaniu tej książki radzę Ci w tym miejscu przerwać czytanie tego rozdziału i przeskoczyć do następnego. Wydaje mi się bowiem, że lepiej jeśli najpierw będziesz miał ogólny pogląd na język C++, a dopiero potem należy zacząć studiować szczegóły.

Nie pytaj mnie też dlaczego wobec tego nie umieściłem tego rozdziału na samym końcu książki. To dlatego, że chciałem, aby szczegóły dotyczące przeładowania nazw funkcji były w jednym miejscu. Po to, że gdybyś chciał do tego wrócić, to znajdziesz wszystko w jednym miejscu.

Tymczasem skok do rozdziału następnego, mówiącego o klasach.

9.9 Etapy dopasowania

Kiedy kompilator napotyka wywołanie przeładowanej funkcji – pracuje nad nim w kilku etapach. Jeśli w rezultacie znajdzie dokładnie jedną z wersji funkcji, która pasuje do wywołania lepiej niż inne, wówczas można powiedzieć, że dopasowanie się udało.

Jeśli znajdzie dwie lub więcej funkcji, które jednakowo zbliżone są do tego, co umieściliśmy w wywołaniu funkcji, to kompilator uzna to za błąd – nie może być żadnej dwuznaczności.

A oto etapy poszukiwania właściwej funkcji. Kompilator, mając przed sobą wywołanie funkcji o przeładowanej nazwie, patrzy na możliwe realizacje tej funkcji i rozważa kolejno:

1) dopasowanie dosłowne,

2) dopasowanie dosłowne z trywialną konwersją,

3) dopasowanie na zasadzie awansowania (z awansem, z promocją),

4) dopasowanie z użyciem konwersji standardowych,

5) dopasowanie z użyciem konwersji wymyślonych przez programistę,

6) dopasowanie do funkcji z wielokropkiem.

Wyjaśnijmy teraz poszczególne etapy.

9.9.1 Etap 1. Dopasowanie dosłownie

Inaczej mówiąc kompilator sprawdza czy argumenty wywołania pasują dokładnie do argumentów formalnych jednej z wersji przeładowanej funkcji.

Na przykład: w wywołaniu funkcji mamy argument będący tablicą typu `int`

```
int tablica[10] ;
     fun(tablica);
```

Pasuje to dokładnie do takiej wersji funkcji `fun`

```
void fun(int ttt[] );
```

Jeśli rzeczywiście taką funkcję mamy, to dopasowanie jest dosłowne.

9.9.2 Etap 2. Dopasowanie dosłowne, ale z tzw. trywialną konwersją

Przykładowo: do wywołania

```
int tablica[10] ;
fun(tablica);
```

nie znaleziono w etapie 1 funkcji

```
void fun(int ttt[] );
```

natomiast jest funkcja

```
void fun(int *wsktab) ;
```

Wiemy przecież, że tablicę wysłaną do funkcji można tam odebrać albo jako tablicę, albo jako wskaźnik do niej. To właśnie jest ta sytuacja. Odebranie tablicy jako wskaźnika jest tzw. trywialną konwersją.

Oto zestawienie innych konwersji uznawanych za trywialne. T jest symbolem jakiegoś typu.

Konwersja		Przykład	
z	do	wywołanie funkcji	jej deklaracja
T	T&	fun(7) ;	fun(int& a) ;
T&	T	fun(przezwisko) ;	fun(int b) ;
T[ ]	T*	fun(tablica) ;	fun(int * wskaźnik) ;
T(argum)	(*T)(argum)	fun(funkcyjka) ;	fun((*wskfun)(int)) ;
T	const T	fun(5) ;	fun(const int n) ;
T	volatile T	fun(21) ;	fun(volatile int m) ;
T*	const T*	fun(wsk) ;	fun(const int * www) ;
T*	volatile T*	fun(wsk) ;	fun(volatile int * www);

Ważny jest tu fakt, że obie sytuacje opisane jako etap 1) i etap 2) są dopasowaniem dosłownym. Dosłownym dlatego, że argument wywołania funkcji może dosłownie (czyli bez żadnych przeróbek) zainicjalizować określony argument formalny.

Oczywiście dopasowanie bez konwersji trywialnej jest dokładniejsze niż takie, w którym ta konwersja musi nastąpić. Czyli, że dopasowanie

`T[]` ⟶ `T[ ]`

jest lepsze niż

`T[]` ⟶ `T*`

Następne etapy to już nie dopasowanie dosłowne. Jeśli więc, mimo dotychczasowych prób dopasowania, do wywołania nie udało się dopasować żadnej z funkcji - trzeba zacząć lekko zmieniać typ argumentów wywołania. Lepiej tak, niż nie dopasować wcale.

9.9.3 Etap 3. Dopasowanie z awansem

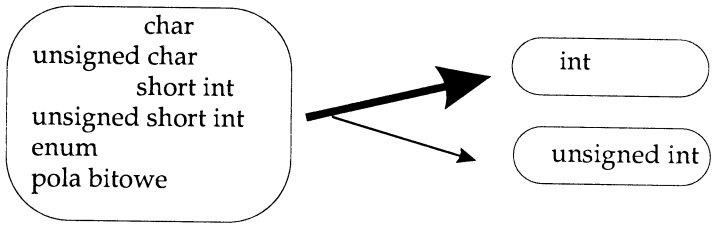
Tak! Argumenty wywołania awansują. Droga awansu dla argumentów zmiennoprzecinkowych jest taka: `float` awansuje na typ `double`.

`float` ⟶ `double`

Jeśli po takim awansie uda się dopasować jednoznacznie jakąś funkcję - to dopasowanie się udało. Jak widać jest to awans do "większej dokładności".

Dla argumentów całkowitych droga awansu jest taka:

Wersje `signed` i `unsigned` następujących typów: `char`, `short int`, typy wyliczeniowe (`enum`), pola bitowe (patrz. str 323) awansują do typu `int` – jeśli taki awans nie powoduje utraty części informacji. W przeciwnym razie jest awans do typu `unsigned int`



Dygresja

Pamiętasz na pewno, że liczby całkowite w różnych typach komputerów mogą być reprezentowane w inny sposób. Nic więc dziwnego, że również awans może odbyć się różnie na różnych komputerach. (Czasem na typ `int`*, a czasem na typ* `unsigned int`*).*
Jeżeli jednak nie zamierzasz uruchamiać swojego programu na różnych typach komputerów to na razie nie musisz się tym martwić.

Przykład dopasowania z awansem.

Mamy takie dwie funkcje `fff`:

```
void fff(int);
void fff(float *) ;
```

Wówczas wywołanie

```
char znak = 'A' ;
fff(znak)
```

rozważane jest etapami tak:

Etap 1 - dopasowanie dosłowne: nierealne.
Etap 2 - dosłowne z trywialna konwersją: też nie wychodzi
Etap 3 - awans. Argument `znak` awansuje do typu `int`. Po tym awansie wywołanie pasuje do funkcji

```
void fff(int) ;
```

Problem więc rozwiązany.

9.9.4 Etap 4. Próba dopasowania za pomocą konwersji standardowych

Konwersjom standardowym poddawany jest oczywiście argument wywołania funkcji.

Do konwersji standardowych oprócz wspomnianych wyżej awansów należą

- Konwersja typu całkowitego

```
int          ------>  unsigned int
unsigned int ------>  int
```

- Konwersja typów zmiennoprzecinkowych

`float` ⟶ `double` *(było w etapie 3)*
`double` ⟶ `float`

❖ Konwersja między typami całkowitym a zmiennoprzecinkowym

typ zmiennoprzecinkowy ⟶ typ całkowity

typ całkowity ⟶ typ zmiennoprzecinkowy

❖ Konwersje arytmetyczne

- czyli takie konwersje jak te, które zwykle robione są automatycznie przy wykonywaniu wyrażeń arytmetycznych.
  Przykładowo: jeśli trzeba pomnożyć dwie liczby: jedną `long` a drugą `short`, to liczba `short` najpierw zamieniana jest na liczbę `long`, po czym dopiero na dwóch liczbach `long` wykonywane jest działanie.
  Takie konwersje nigdy nie powodują utraty jakiejkolwiek części informacji.

❖ Konwersje wskaźników

- 0 (zero) może być zamienione na wskaźnik do NULL
- wskaźnik dowolnego nie-`const` i nie-`volatile` typu może być zamieniony na wskaźnik typu `void*`

(dalsze uwagi dotyczą klas – zakładam, że czytasz książkę po raz drugi)

- wskaźnik do klasy pochodnej może być zamieniony na wskaźnik do klasy podstawowej, od której ta klasa pochodna się wywodzi.

❖ Konwersja referencji

- referencja do klasy pochodnej może być zamieniona na referencję do klasy podstawowej.

W czwartym etapie dopasowania argument wywołania funkcji poddawany będzie powyższym konwersjom standardowym. Możesz więc wysłać do przeładowanej funkcji argument, któremu (dzięki tym konwersjom) kompilator znajdzie jakąś pasującą do niego funkcję.

Jednak rada jest taka:

Nie licz zbytnio na standardowe konwersje, bo może się zdarzyć, że przy konwersji część informacji zostanie stracona. Przykładowo:

```
void fff(int) ;
void fff(char *);

float pi = 3.14 ;

fff(pi) ;
```

Po dopasowaniu do funkcji `void fff(int)` wartość 3.14 zostanie zamieniona na 3 – Tego chciałeś?

9.9.5 Etap 5. Próba dopasowania z użyciem konwersji zdefiniowanych przez użytkownika.

Jak pewnie już wiesz – z pierwszego czytania tej książki - w zdefiniowanej przez siebie klasie możesz umieścić funkcję zamieniającą obiekt jakiegoś typu na obiekt typu tej klasy. (Patrz str. 399).

Jeśli argument wywołania da się według takiej konwersji zamienić na typ odpowiadający argumentowi formalnemu, to dopasowanie jest pomyślne.

> Ale uważaj: dla argumentu robiona jest tylko jedna taka konwersja.
>
> Nawet jeśli zdefiniowałeś jak z typu A zrobić typ B, oraz jak z typu B zrobić typ C, a także jak z typu C zrobić typ D. Nic z tego. Kompilator zastosuje najwyżej tylko jedną z nich. Nie zrobi całej „kaskady".

(Bardziej szczegółowo o tym na stronie 416).

9.9.6 Etap 6. Próba dopasowania do funkcji z wielokropkiem

Jest to już rozpaczliwa próba. Jeśli do tej pory nie powiodło się żadne dopasowanie, to szuka się funkcji mającej w liście argumentów wielokropek. Wielokropek oznacza: dowolna liczba argumentów dowolnego typu.

Np.

```
void fun(int) ;
void fun (float) ;
void fun(...) ;
// ---------------a wywołanie wygląda w ten sposób
fun("napis") ;
```

Żadna z zadeklarowanych nie daje się dopasować, bo argument wywołania to przecież wskaźnik. Jest jednak funkcja, która ma na liście argumentów formalnych wielokropek. Dopasowanie następuje do tejże funkcji.

Jeśli na liście jest kilka argumentów formalnych, a dopiero potem wielokropek, to pamiętajmy, że:

> Wielokropek na liście argumentów formalnych funkcji obejmuje argumenty od tej pozycji, na której stoi, oraz ewentualne dalsze.

9.10 Dopasowywanie wywołań z kilkoma argumentami

Jeśli w wywołaniu funkcji jest kilka argumentów, to wówczas procedura dopasowania argumentów odbywa się na każdym z nich.

Ze wszystkich wersji funkcji wybrana zostaje ta wersja, do której parametry pasują tak samo lub lepiej niż do innych wersji. Co to oznacza ? Wyobraźmy sobie, że mamy wywołanie dwuargumentowe

```
fun(2, 5) ;
```

a funkcje, spośród których będzie kompilator wybierał to:

```
fun(float, unsigned int);
fun(float, float);
fun(int *, float) ;
```

Ostatnia wersja – ta ze wskaźnikiem od razu odpada. Nie da się przecież konwersjami standardowymi zamienić liczby 2 na wskaźnik do `int`.

Zostają zatem tylko dwie funkcje. Pierwszy argument wywołania czyli 2 pasuje tak samo źle do obu wersji, no ale w końcu po standardowej konwersji można wytrzymać.

Natomiast drugi argument wyraźnie lepiej pasuje do wersji

```
fun(float, unsigned int);
```

Wyraźnie lepiej – dlatego, że wystarczy jego awans do typu `unsigned int`. Awans, jak wiemy, jest lepszy niż ewentualna konwersja standardowa podobna do tej, jaką zrobiliśmy z argumentem pierwszym. Zatem – skoro pierwszy argument pasował tak samo (tak samo źle) od obu wersji funkcji, natomiast drugi argument pasował lepiej do wersji

```
fun(float, unsigned int);
```

to ta wersja zostanie przez kompilator wybrana.

Koniec Tomu Pierwszego

10 Klasy

Tak naprawdę, to wszystko co pisałem do tej pory – pisałem z nadzieją, że wreszcie dotrę do tego rozdziału. Tu bowiem zaczniemy mówić o chyba najwspanialszej rzeczy w C++ – czyli o definiowaniu własnych typów danych.

10.1 Typy definiowane przez użytkownika

Czy nigdy nie irytowało Cię, że gdy masz napisać program, to musisz swój problem, który dotyczy jakichś realnych obiektów (silników krokowych, pokoi hotelowych itd.) zamienić na luźne liczby i luźne podprogramy (funkcje) ?

Rozwiązywanie Twojego problemu polega wtedy na żonglowaniu liczbami, które w programie wcale nie są powiązane ze sobą, ani też nie są powiązane z funkcjami. To tylko my wiemy, że dotyczą one tego samego silnika krokowego, tej samej pralki automatycznej, czy tego samego rachunku bankowego. Liczba typu `float` reprezentująca temperaturę powierzchni promu kosmicznego może być przez roztargnienie programisty wysłana do funkcji oczekującej liczby typu `float` reprezentującej wydajność w kwintalach na hektar. Kompilator nie zauważy takiego błędu: miała być liczba `float` i jest `float` – a więc o co chodzi? Tyle, że na skutek tego wynik będzie bzdurny.

W C++ dane mogą zostać powiązane z funkcjami – znaczy to, że kompilator nie dopuści do tego, by do funkcji oczekującej argumentu *typu* „temperatura" wysłać argument *typu* „stan_oszczędności". Żeby tak było musimy najpierw zdefiniować sobie taki typ.

> C++ pozwala nam na zdefiniowanie własnego typu danej. Słowem oprócz typów `float`, `int`, `char` itd., mamy jeszcze nasz własny typ – wymyślony na użytek danego programu. Ten typ, to nie tylko jedna lub kilka zebranych razem liczb – to także sposób ich zachowania jako całości.

Jaka z tego korzyść ?

Tak zdefiniowany typ może być „modelem" jakiegoś rzeczywistego obiektu. Rzeczywisty obiekt – np. pralkę automatyczną – można w komputerze opisać zespołem liczb i zachowań.

Dla pralki automatycznej te *liczby* to na przykład jej cena, rok produkcji, wymiary, kolor, czy wydajność wyrażona w kilogramach bielizny, którą pralka może prać za jednym razem. To także liczby reprezentujące jej stan wewnętrzny opisany przez program prania, na który ją właśnie nastawiliśmy, czy też liczba opisująca bieżący etap prania. Słowem składnikiem takiego obiektu jest zbiór różnych liczb.

Z kolei *zachowania* pralki automatycznej to zbiór funkcji, które może dla nas wykonać. Może to być pranie, płukanie, wirowanie itd. Te funkcje są także składnikiem obiektu typu pralka automatyczna.

Luźne liczby i luźne funkcje – o których wiemy, że opisują w programie pralkę automatyczną – zbieramy razem i budujemy z nich pewną całość. Powstaje w komputerze nowy typ danej: pralka automatyczna.

Mówimy **typ** dlatego, że kreujemy nie jeden konkretny obiekt, ale raczej **klasę** obiektów. Czyli na razie raczej wymyśliliśmy pralkę automatyczną – jeszcze żaden konkretny egzemplarz takiej pralki (obiekt) nie istnieje.

> Klasa to inaczej mówiąc typ.

Tak jak typem jest `int`, `float`, ...

Przyjrzyjmy się teraz jak się klasę definiuje

Nie jest to trudne. Definicja klasy składa się ze słowa kluczowego `class`, po którym stawiamy wybraną przez nas nazwę.

```
class nasz_typ {
                    //.............ciało klasy
                    //.................................
} ;
```

Potem następuje klamra, a w niej ciało klasy - czyli określenie z czego się składa. Zauważ, że po klamrze stawia się średnik. Początkowo będziesz o nim często zapomniał.

Jeśli w przyszłości zechcemy stworzyć egzemplarz obiektu takiej klasy, to wystarczy wówczas podać nazwę typu i nazwę tego konkretnego nowego obiektu. Identycznie jak w wypadku typu wbudowanego `int`. Aby mieć egzemplarz obiektu typu `int` o nazwie `suma` piszemy:

```
int suma ;
```

Przy typie definiowanym przez użytkownika – sprawa wygląda analogicznie. Oto utworzenie obiektu `abc`, który jest klasy `nasz_typ`:

```
nasz_typ abc ;
```

Dzięki temu zapisowi w pamięci maszyny utworzony zostaje jeden obiekt klasy `nasz_typ` i temu egzemplarzowi nadana jest nazwa `abc`. Zapis

```
nasz_typ m ;
```

spowoduje utworzenie w pamięci drugiego egzemplarza obiektu klasy `nasz_typ`. Ma on nazwę `m`. To tak, jakbyśmy na podstawie tych samych planów konstrukcyjnych zbudowali drugą pralkę automatyczną. Te plany konstrukcyjne to oczywiście definicja klasy.

Skoro mamy typ, to można utworzyć od niego typ pochodny – czyli na przykład wskaźnik do obiektów tego typu:

```
nasz_typ * wsk ;
```

czy też na przykład referencję (przezwisko) obiektu takiego typu

```
nasz_typ obiekcik ;
nasz_typ & przezw = obiekcik ;
```

10.2 Składniki klasy

W definicji naszej klasy widzimy na razie puste miejsce zwane ciałem klasy. W tym właśnie miejscu deklaruje się **składniki klasy**.

Składnikami mogą być różnego typu dane (np. `int`, `float`, stringi itd). Nazywamy je danymi składowymi tej klasy. Będę też na nie czasem mówił: składniki-dane. Oto definicja klasy, w której jest kilka danych składowych:

```
class pralka {
public:
     int       nr_programu ;
     float     temperatura_prania ;
     char      nazwa[80] ;
} ;
```

(Bardzo proszę nie pytaj mnie jeszcze co znaczy to słowo `public`. Powiemy o tym dopiero za chwilę).

Aby odnieść się do składników obiektu możemy się posługiwać jedną z poniższych notacji:

- obiekt.składnik
- wskaźnik -> składnik
- referencja.składnik

Jak widać występuje tu operator '.' (kropka) – operator odniesienia się do składnika obiektu znanego z nazwy lub referencji. Jeśli mamy obiekt pokazywany wskaźnikiem, to do odniesienia się do składnika takiego obiektu służy nam operator ->

Jeśli mamy obiekt

```
pralka czerwona ;              // definicja egzemplarza obiektu
pralka * wskaz ;               // definicja wskaźnika
pralka & ruda = czerwona;      // definicja referencji
```

to do składnika `temperatura_prania` w obiekcie `czerwona` możemy odnieść się tak:

```
czerwona.temperatura_prania = 60 ;      // nazwą obiektu
```

```
wskaz = & czerwona ;
wskaz -> temperatura_prania = 60 ;      // wskaźnikiem

ruda.temperatura_prania = 60 ;          // referencją
```

Składnikami klasy mogą być też — uwaga-uwaga! — funkcje. Funkcje te nazywać będziemy **funkcjami składowymi**. Za ich pomocą pracujemy zwykle na danych składowych. Oto klasa `pralka` wyposażona w funkcje:

```
class pralka {
public :
                                  //——— funkcje składowe ———
     void      pierz(int program);
     void      wiruj(int minuty);

                                  //——— dane składowe ———
     int       nr_programu ;
     float     temperatura_prania ;
     char      nazwa[80] ;

                                  //——— znowu jakaś funkcja składowa
     int       krochmalenie(void) ;

} ;
```

W definicji tej widzisz deklaracje funkcji pomieszane z deklaracjami danych. Niezależnie od miejsca zdefiniowania składnika wewnątrz klasy – składnik znany jest w całej definicji klasy.

Mówimy, że nazwy deklarowane w klasie mają zakres ważności – równy obszarowi całej klasy.

Inaczej niż to było w zwykłych funkcjach. Przywykliśmy, że jeśli w połowie funkcji zdefiniowaliśmy daną, to była ona znana od miejsca definicji aż do końca funkcji. W linijkach powyżej znana nie była. Przykład:

```
void funkcja()
{
int a,b,c ;

     c = 15 ;
     a = 4 + c ;
                         // tu jeszcze nazwa nnn nie jest znana—
     int nnn ;           // <——— moment definicji obiektu nnn
                         // tu już nnn jest znane ————————
     nnn = a + 6 ;
     // ...
}
```

Dana składowa może być zdefiniowana nawet w ostatniej linijce ciała klasy, a i tak jest znana w całości klasy. Po prostu klasa w przeciwieństwie do funkcji nie ma początku i końca. To jakby pudło na składniki.

Potem zobaczymy, że składnikami klasy może być jeszcze wiele innych ciekawych rzeczy.

10.3 Składnik będący obiektem

Składnikiem klasy może być dana typu wbudowanego np. obiekt typu `int`, `char[20]`, `float*`, a może być też obiekt typu zdefiniowanego przez użytkownika. Innymi słowy obiekt jakiejś innej klasy.

Początkowo może się to wydać zawiłe więc posłużmy się analogią: obiekt klasy **lampa**, który stoi przede mną na biurku ma składniki – liczby – takie jak wysokość, ciężar, ale też jego składnikiem jest inny obiekt: żarówka, czy też obiekt klasy abażur. Każdy z nich sam jest obiektem jakiejś klasy. Oczywiście mógłbym udać, że o tym nie wiem i wpisywać wszystkie składniki obiektu klasy żarówka (moc, wielkość bańki) w obrębie definicji klasy lampa. Tylko co bym przez to zyskał? Nic. A co bym stracił? Straciłbym klasę żarówka, która może mi się przydać w jeszcze innych miejscach programu. Właściwie na tym polega jedna z tajemnic programowania obiektowego: aby umiejętnie używać klas już kiedyś zdefiniowanych.

Wniosek jest taki:

> opłaca się po prostu budować klasy składające się z innych obiektów.

Budując dom opłaca się skorzystać z gotowego obiektu cegła, a nie budować dom wypalając jednocześnie nieopodal cegłę.

10.4 Enkapsulacja

Jak widzisz w ciele klasy są dane i funkcje. Jest to bardzo ważny fakt, bowiem w tej definicji, jak w kapsule, zamknęliśmy dane oraz funkcje do posługiwania się nimi. Po angielsku takie zamknięcie w kapsułę nazywa się: encapsulation. Jeśli chodzi o zgrabne i poprawne przetłumaczenie tego terminu na polski – poddaję się i wybieram wariant najprostszy: enkapsulacja.

Dlaczego enkapsulacja jest taką ważną cechą języka C++? Dlatego, że odzwierciedla nasze codzienne myślenie o obiektach.

Na przykład obiekt `zegarek` to nie tylko trybiki, kółka i wskazówki. To także sposób postępowania z nim i zachowania, które tym składnikom towarzyszą. Ten sposób postępowania jest charakterystyczny dla zegarka. Powiedzenie „nastaw!" w stosunku do zegarka to jest przecież jakaś akcja – charakterystyczna dla zegarka! Powiedzenie „nastaw!" w stosunku do obiektu klasy czajnik spowoduje, że uruchomimy akcję, która w naszym mózgu jest składnikiem klasy czajnik. Zauważ: zupełnie innej akcji.

Zaraz, zaraz – to chyba już znamy! Czyżby przeładowanie nazwy funkcji?

Nie. Nie ma tu mowy o żadnym przeładowaniu nazw funkcji. Jeśli mówimy: wykonaj na obiekcie klasy zegarek funkcję `nastaw`, to kompilator nie próbuje niczego dopasowywać. Sięga na ślepo do klasy zegarek, a tam jest właściwa funkcja `nastaw`.

To było zdroworozsądkowe tłumaczenie. Można to jednak bardzo krótko wytłumaczyć na podstawie definicji przeładowania: Przeładowanie następuje wtedy, gdy funkcje o tej samej nazwie mają identyczny zakres ważności. Jeżeli zaś

mają inny zakres ważności, to nie następuje przeładowanie tylko zasłonięcie. Funkcje składowe klasy mają zakres klasy – przecież ich deklaracje tkwią wewnątrz nawiasu klamrowego, który wyznacza lokalny zakres ważności. Mówimy: zakres ważności klasy.

Zakresem ważności jednej funkcji `nastaw` jest klasa czajnik. Zakresem innej funkcji `nastaw` jest klasa zegarek. Te funkcje mogą się co najwyżej nawzajem zasłaniać. Czasem widać jedną, czasem drugą.[†)]

Wróćmy jednak do istoty rzeczy. Definicja klasy sprawiła, że dane i funkcje, które dawniej mielibyśmy luźnie rozrzucone w programie - teraz zamknięte są w kapsule. Dzięki temu, w momencie definicji pojedynczego egzemplarza obiektu takiej klasy — dostajemy realizację takiej kapsuły.

```
pralka biala ;                    // definicja egzemplarza
                                  // obiektu klasy pralka
```

To tak, jak kupujemy pralkę w obudowie, a nie luźne części. Jeśli chcemy mieć drugi obiekt takiej pralki to definiujemy

```
pralka czerwona ;                 // inny obiekt klasy pralka
```

To ogromna wygoda. W tradycyjnym programowaniu jeśli nawet zdefiniowalibyśmy te wszystkie luźne składniki klasy pralka, to przy drugim egzemplarzu musielibyśmy zrobić to samo jeszcze raz.[††)]

> Kapsułą się łatwiej posługiwać niż rozsypanymi elementami. Jeśli Cię to jeszcze nie przekonuje, to pomyśl dlaczego transport przestawił się na przewożenie obiektów klasy kontener.

10.5 Ukrywanie informacji

Skoro, jak powiedzieliśmy, składniki klasy zamknięte są w kapsule - to ta kapsuła może być przeźroczysta lub nie. Coś może być dostępne spoza klasy lub nie.

Oto przykład klasy:

```
class nasz_typ {
                    // składniki prywatne ***********************
private :
     int        liczba ;  // <— prywatne dane składowe
     float      temperatura ;
     char       komunikat[80] ;

     int czy_gotowe() ;  // <— prywatna funkcja składowa
```

†) Natomiast nic nie przeszkadza, by w obrębie danej klasy dana funkcja była przeładowana. Zakres tych wersji funkcji jest wtedy ten sam. Może być przecież 15 sposobów nastawiania zegarka.

††) Przesadzam. W klasycznym C można się posłużyć tzw. strukturami. Za to w językach FORTRAN, BASIC nie ma już takich narzędzi. Tu byłaby męka.

```
                    // składniki publiczne **********************
public :
     float     predkosc ;      // <— publiczna dana składowa
     int  zrob_pomiar() ;      // <— publiczna funkcja składowa
} ;
```

W ciele tej klasy widzimy wyraźnie dwie grupy – występujące po etykietach `private` i `public`.

Etykieta `private`

oznacza, że deklarowane za nią składniki (funkcje i dane) są **dostępne** tylko z wnętrza klasy. W wypadku danych składowych oznacza to, że tylko funkcje będące składnikami klasy mogą te prywatne dane odczytywać lub do nich zapisywać.

W wypadku funkcji oznacza to, że mogą one zostać wywołane tylko przez inne funkcje składowe tej klasy†).

Etykieta `public`

Dalej w definicji klasy `nasz_typ` widzimy grupę składników pod wspólną etykietą `public`. Publiczne składniki-dane mogą być używane z wnętrza klasy, a także spoza zakresu klasy. Analogicznie publiczne funkcje składowe mogą być wywoływane dodatkowo także spoza klasy.

Publiczną funkcją składową obiektów klasy pralka jest na przykład funkcja `pranie_koszul`, to dlatego, że ja, nie będąc składnikiem klasy pralka, mogę tę funkcję wywołać. Prywatną funkcją jest na przykład jakaś funkcja `obroc_beben_pralki_w_lewo`. Tego nie mogę bezpośrednio wywołać.

> Pomyślisz pewnie tak: skoro składniki prywatne są upośledzone, bo nie ma do nich dostępu z zewnątrz, to dlaczego nie opatrzyć wszystkiego etykietą `public`. A może nie można?

Można! W C++ można prawie wszystko. Tylko zobacz jakie będą tego konsekwencje. Wyobraź sobie, że wytwarzasz klasę obiektów pod tytułem „telewizor". Chodzą one świetnie. Mają w sobie ok. 50 elementów, które można śrubokrętem stroić. Dajesz teraz obiekt takiej klasy użytkownikowi. Co dostaje? Telewizor bez obudowy. Wszystkie 50 miejsc zagrożone jest jego ochoczą akcją śrubokrętem. Nawet jeśli dostarczane przez Ciebie telewizory są świetne, to gdy ktoś weźmie do ręki śrubokręt i zacznie kręcić – rozstroi to tak, że telewizor będzie działał źle. To zepsuje Ci opinię.

Z kolei inny użytkownik, gdy będzie chciał zwiększyć w telewizorze jasność, a zobaczy 50 pokręteł, to po prostu powie: to za trudne, lepiej już pójdę do kina!

Jaka jest sytuacja optymalna? Zaopatrzyć obiekt w obudowę broniącą dostępu do tych 50 miejsc, a na zewnątrz obudowy wystawić do publicznego użytku tylko pokrętła spełniające funkcje: zwiększ jasność, przełącz kanał.

†) Nie wspominam tu na razie o tzw. funkcjach zaprzyjaźnionych

Tak należy nauczyć się rozumować w wypadku wymyślania klasy. My wymyślamy klasę, a ktoś inny będzie musiał za pomocą obiektów tej klasy programować. Taka jest sytuacja przy programowaniu w zespołach. Jeśli nawet jesteś samodzielnym programistą amatorem i to Ty sam będziesz klasę definiował, oraz tylko Ty sam będziesz z niej korzystał, to wszystko to, co powiedziałem, nadal pozostaje ważne. Ktoś, kto amatorsko zbudował sobie na stole zasilacz, wkłada na końcu te wszystkie druty i tranzystory do pudełka po butach. To po to, by – gdy za chwilę tego zasilacza będzie używał przy zasilaniu kolejki elektrycznej – nie zrobić przypadkowego zwarcia między nóżkami tranzystorów. Ani, by nie przestawić omyłkowo napięcia zamiast zwrotnicy.

A zatem dowiedzieliśmy się, że:

Istnieją etykiety za pomocą, których można określać dostęp do składników klasy.

Poznaliśmy już dwie, ale jest ich trzy

```
private:
protected:
public:
```

Określają one dostęp do poszczególnych składników klasy.

Są trzy rodzaje dostępu do składnika klasy

Składnik `private`

- jest dostępny tylko dla funkcji składowych danej klasy. (Także dla funkcji zaprzyjaźnionych z tą klasą – por. str 307). Jeżeli zależy nam na ukryciu informacji, to wówczas składnik powinien być deklarowany właśnie jako prywatny.

Składnik `protected`

- jest dostępny tak, jak składnik `private`, ale dodatkowo jest jeszcze dostępny dla klas wywodzących się od tej klasy.
  (O tym, że klasa może mieć potomków będziemy mówić w rozdziale o dziedziczeniu. Tu tylko zapamiętajmy, że składniki `protected` są to składniki zastrzeżone dla siebie i rodzinki).

Składnik `public`

- jest dostępny bez ograniczeń. Zwykle składnikami takimi są jakieś wybrane funkcje składowe. To za ich pomocą dokonuje się z zewnątrz operacji na danych prywatnych.

Etykiety te można umieszczać w dowolnej kolejności, mogą też się powtarzać. Zawsze oznaczają, że te składniki klasy, które następują bezpośrednio po etykiecie – mają tak określony dostęp.

Domniemanie

Zakłada się, że – dopóki w definicji klasy nie wystąpi żadna z tych etykiet – składniki przez domniemanie mają dostęp `private`.

```
class dzika {
     int       a ;
```

```
        float      b ;
        void       fun1(int) ;
protected :
        char       m ;
        void       fun2(void) ;
public :
        int        x ;
        void       fun3(char*);
private:
        int        d ;
public :
        void       fun4(void) ;
} ;
```

W powyższym przykładzie następujące składniki klasy są prywatne (zastrzeżone dla siebie)

```
a, b, fun1, d
```

protected – (czyli zastrzeżone dla siebie i potomków)

```
m, fun2
```

publiczne (dostępne dla wszystkich)

```
x, fun3, fun4
```

Na początku klasy nie ma żadnej etykiety, więc zakłada się, że składniki `a`, `b`, `fun1` mają być prywatne. Potem występują już etykiety, którymi regulujemy dostęp do wybranych składników.

Pokazany sposób określania dostępu jest co prawda poprawny, jednak nie polecałbym go. Lepiej wszystkie składniki o danym dostępie zgrupować razem. Wówczas wystarcza jeden rzut oka na definicję klasy i już wiemy co jest dostępne z zewnątrz.

> Podkreślić należy wyraźnie, że sterowanie dostępem jest pewnego rodzaju dobrodziejstwem, które chroni nas byśmy sobie czegoś w danym obiekcie przez nieuwagę nie zepsuli.

Sterowanie dostępem nie zabezpiecza jednak przed świadomym działaniem nastawionym na zepsucie. W C++ prawie wszystko jest możliwe, więc jeśli zechcesz to i tak możesz się do takich prywatnych danych dostać. No, ale to już będzie świadomym oszustwem.

10.6 Klasa a obiekt

Wiemy już co to jest klasa. Definicja klasy to jakby projekt techniczny nowego typu zmiennej. Mając już zdefiniowaną klasę możemy stworzyć kilka egzemplarzy obiektów danej klasy. Tak jak w wypadku typu (klasy) `int` możemy stworzyć kilka obiektów typu `int`:

```
int a, m, licznik, cena, kolor ;
```

Ta definicja powoduje utworzenie pięciu różnych obiektów typu (klasy) `int`. Podobnie w wypadku typu zdefiniowanego przez nas samych po definicji klasy możemy przystąpić do definiowania konkretnych egzemplarzy obiektów danej klasy.

Wymyślmy sobie taką prostą klasę:

```
class osoba {
     char nazwisko[80] ;
     int wiek ;
public:
     void zapamietaj(char *, int );
     void wypisz();
} ;
```

W klasie tej widzimy dwa prywatne składniki – są to dane zawierające informacje o nazwisku i wieku. Publicznymi składnikami są dwie funkcje

- `zapamietaj` – wpisująca informacje o nazwisku i wieku do odpowiednich składników
- `wypisz` – do wypisania na ekranie informacji zapisanej wcześniej w obiekcie.

Definicja czterech egzemplarzy obiektów tej klasy to po prostu instrukcja

```
osoba student1, student2, profesor, pilot ;
```

Dygresja:

> Zamiast mówić:
>
> **egzemplarz obiektu danej klasy**
>
> mówić też będziemy po prostu:
>
> **obiekt danej klasy.**
>
> Ponieważ jednak to pierwsze, dłuższe sformułowanie wyraźniej podkreśla różnicę między klasą a obiektem, dlatego przez pewien czas częściej będę używał pierwszej formy.

Zobaczyliśmy więc jak powstają cztery egzemplarze obiektów naszej klasy `osoba`. Jak widać każdy z nich ma swoją nazwę.

> Należy sobie wyraźnie uświadomić, że sama definicja klasy nie definiuje żadnych obiektów.

Konkretniej – w naszym wypadku po definicji klasy nie ma jeszcze w pamięci żadnej tablicy `nazwisko`, ani żadnej zmiennej `wiek`. To dopiero definicja czterech **egzemplarzy obiektów** tej klasy sprawiła, że w pamięci powstały cztery zespoły danych – cztery tablice do przechowywania nazwiska i cztery zmienne typu `int` do przechowywania informacji o wieku osoby. Każdy z tych zespołów danych może przechować dane o jednej osobie.

> Sama definicja klasy to jakby tylko pieczątka. Dopóki jej nie odbijemy czterokrotnie na papierze (pamięć komputera) dotąd na tym papierze nic nie ma. Pieczątka leży sobie na boku.

Łatwo sobie to uzmysłowić patrząc na typ wbudowany jakim jest typ `int`. Twórcy języka zdefiniowali ten typ `int`, aby służył programistom do przechowywania liczb całkowitych. Ta definicja typu (klasy) `int` już gdzieś w języku C++ tkwi. Dopóki jednak nie napiszemy definicji egzemplarza obiektu tego typu, dotąd w pamięci nie jest zarezerwowana żadna komórka na ten cel. To dopiero definicja

```
int x ;
```

sprawia, że w pamięci jest jeden obiekt tego typu.

Zapamiętaj:

Klasa to typ obiektu, a nie sam obiekt.

W definicji klasy składniki dane nie mogą mieć inicjalizatora. Definicja klasy jest przecież tylko jakby formularzem do wypełnienia. Dokument klasy paszport, której matryca (definicja) jest w drukarni państwowej, nie zawiera wydrukowanego nazwiska ani daty urodzenia. Jest tam tylko miejsce na nazwisko. Te dane (inicjalizujące go) wpisuje się dopiero się do konkretnego egzemplarza obiektu klasy paszport.

```
class paszport {
     char nazwisko[40] ;
     char imie [40] ;
     int numer ;
     int  wzrost = 176 ;                    // błąd !!!
} ;
```

Gdyby tak wyglądała definicja klasy paszport, to wszystkie paszporty miały by na zawsze wydrukowany wzrost właściciela równy 176 cm. Na taki bezsens kompilator C++ nie pozwoli.

Dane do obiektu danej klasy wpisuje się dopiero, gdy konkretny obiekt definiujemy (wyrabiamy sobie paszport, kupujemy pralkę) albo później, gdy jakichś zmian potrzebujemy (załatwiamy do paszportu jakąś wizę, lub wsypujemy do pralki proszek). O tym jeszcze porozmawiamy w osobnym rozdziale.

Warto uzmysłowić sobie jeszcze jedną rzecz: Otóż jeśli w naszym wypadku zdefiniowaliśmy cztery obiekty klasy `osoba`, to w pamięci utworzone zostały cztery różne komplety składników danych tej klasy. To w końcu zrozumiałe, bo muszą być np. cztery tablice do przechowywania czterech różnych nazwisk. Co jednak ciekawe:

> funkcje składowe są w pamięci tylko jednokrotnie.

Cztery oddzielne komplety składników danych są zrozumiałe, bo przecież każdy ma przechowywać inną informację. Natomiast funkcje składowe dla każdego egzemplarza obiektu danej klasy działają przecież identycznie. Są więc w pamięci komputera jednokrotnie – po to, by oszczędzać pamięć.

W zasadzie o tej sprawie nie musiałbyś wcale wiedzieć. Powiedziałem to jednak po to, byś nie sądził, że poszczególny obiekt w pamięci jest bardzo duży. Taka obawa prowadziłaby do niechętnego definiowania nowych egzemplarzy obiektów.

Znowu analogia. Składnikiem typu wbudowanego `int` *jest na pewno funkcja (składowa) obsługująca mnożenie liczb typu* `int`*. Nie sądzisz chyba, że za każdym razem, gdy w programie definiujesz zmienną typu* `int` *przydzielana jest mu w pamięci nowa funkcja obsługująca to mnożenie. Obiektów tego typu może być sto, a obsługuje je ta sama funkcja.*

O tym, jak wielki jest obiekt, możesz się łatwo przekonać stosując operator `sizeof`.

10.7 Funkcje składowe

Funkcje zdeklarowane wewnątrz definicji klasy są składnikami tej klasy. Nazywamy je funkcjami składowymi. Funkcje te mają zakres klasy, w której je zadeklarowaliśmy. (Zwykłe funkcje – jak pamiętamy – mają zakres pliku, w którym je zadeklarowano).

Funkcje składowe mają pełny dostęp do wszystkich składników swojej klasy — to znaczy: i do danych (mogą z nich korzystać) i do innych funkcji (mogą je wywoływać). Do składnika swojej klasy odwołują się po prostu podając jego nazwę.

10.7.1 Posługiwanie się funkcjami składowymi

Funkcja składowa jest jakby narzędziem, za pomocą którego dokonujemy operacji na danych składowych klasy. Szczególnie na tych składnikach, które są `private` i tym samym spoza klasy niedostępne. Przyjrzyjmy się jak – dla przed chwilą zdefiniowanej klasy `osoba` – możemy wywołać funkcje składowe.

Powiedzmy ściślej – funkcję wywołuje się dla konkretnego obiektu danej klasy. Musimy więc mieć definicje obiektów.

```
osoba student1, student2, profesor, pilot ;
```

Oto wywołanie funkcji składowej dla obiektu `profesor`:

```
profesor.zapamietaj("Albert Einstein", 55);
```

Dla pozostałych obiektów podobnie

```
student1.zapamietaj("Ruediger Schubart", 26);
student2.zapamietaj("Claudia Bach", 25);
pilot.zapamietaj("Neil Armstrong", 37);
```

Wywołanie takie należy rozumieć tak: na rzecz obiektu `profesor` wykonaj funkcję `zapamietaj` z podanymi argumentami.

Składnia jest więc taka: nazwa obiektu, potem kropka, a potem nazwa funkcji składowej z ewentualnymi argumentami

obiekt.funkcja(argumenty) ;

Zapytasz zapewne – Po co ten obiekt i ta kropka? Nie można by wywołać tej funkcji po prostu tak:

```
zapamietaj("Albert Einstein", 55);          // Błąd !
```

Nie, nie można. Zapominasz, że w pamięci są już cztery zestawy danych odpowiadające czterem różnym obiektom klasy `osoba`. Funkcja musi wiedzieć na którym konkretnym obiekcie ma pracować.

Jeśli funkcja `zapamietaj` ma wpisać nazwisko „Albert Einstein" do tego konkretnego obiektu klasy `osoba`, który to obiekt nazwaliśmy `profesor`, to tę nazwę `profesor` stawiamy przed wywołaniem funkcji.

Możemy także wywołać funkcję składową dla tego samego obiektu pokazywanego wskaźnikiem:

```
osoba * wsk ;              // definicja wskaźnika
wsk = &profesor ;          // ustawienie wskaźnika na
                           // obiekcie profesor

wsk -> zapamietaj("Albert Einstein", 55);
```

A oto jak wywołać funkcję dla tego samego obiektu przezywanego referencją:

```
osoba & belfer = profesor ;        // definicja referencji

belfer.zapamietaj("Albert Einstein", 55);
```

10.7.2 Definiowanie funkcji składowych

Do tej pory dużo mówiliśmy o funkcji `zapamietaj`, ale nigdzie nie pojawiła się jeszcze jej definicja, czyli jej treść, ciało – instrukcje składające się na nią.

Gdzie może być zdefiniowana funkcja składowa?

Może się ona znaleźć w dwóch miejscach:

Pierwszy sposób: -Może się znaleźć wewnątrz samej definicji klasy.

Oto taka realizacja:

```
class osoba {
                                        // składniki private
     char nazwisko[80] ;
     int wiek ;
public :                                // składniki publiczne

     //------------------------------- definicje funkcji składowych
     void zapamietaj(char * napis, int lata)
     {
          strcpy(nazwisko, napis) ;
          wiek = lata ;
     }
     //-----------------------------------
     void wypisz()
     {
          cout << nazwisko << " , lat : " << wiek << endl;
     }
};
```

Jak widać funkcja `zapamietaj` przysłane do niej argumenty przepisuje do składników `nazwisko` oraz `wiek`. Przy przepisywaniu nazwiska (jest to string) posługuje się funkcją `strcpy` (string copy) z biblioteki standardowej.

Jest też **drugi sposób** definiowania funkcji składowych:
W definicji klasy umieszcza się tylko same deklaracje tych funkcji, natomiast definicje są napisane poza ciałem klasy:

```
class osoba {
                                                    // składniki private
     char nazwisko[80] ;
     int wiek ;
public :                                            // składniki publiczne

     //————————————————deklaracje funkcji składowych
     void zapamietaj(char * napis, int lata) ;
     void wypisz() ;
} ;                                        // koniec definicji klasy
/*****************************************************/
void osoba::zapamietaj(char * napis, int lata)
{
     strcpy(nazwisko, napis) ;
     wiek = lata ;
}
/*****************************************************/
void osoba::wypisz()
{
     cout << nazwisko << " , lat : " << wiek << endl ;
}
```

Ponieważ funkcje znajdują się teraz poza definicją klasy dlatego ich nazwa uzupełniona została nazwą klasy, do której mają należeć. Służy do tego widoczny operator zakresu :: .

Taki zapis informuje kompilator, że jest to realizacja tej funkcji `zapamietaj`, którą zadeklarowaliśmy w definicji klasy `osoba`. Jeśli byśmy o umieszczeniu tego przedrostka zapomnieli – kompilator uzna, że jest to jakaś funkcja `zapamietaj` – jedna z wielu zwykłych w programie. Natomiast w trakcie linkowania linker nie znajdzie poszukiwanej przez siebie funkcji `osoba::zapamietaj` i otrzymamy komunikat, że funkcja składowa `zapamietaj` dla klasy `osoba` nie jest nigdzie zdefiniowana.

Nazwa klasy i operator zakresu są rzeczywiście jakby przedrostkiem nazwy funkcji. Mówię o tym dlatego, by ustrzec Cię przed błędem polegającym na tym, że nazwę klasy i operator :: ustawisz na samym początku linijki, a dopiero po tym typ zwracany przez funkcję.

```
     osoba:: void zapamietaj(...)              // Błąd !
     { /*ciało*/   }
```

Błędu tego nie popełnisz jeśli zapamiętasz, że nazwa klasy i operator :: uzupełniają **nazwę** funkcji i jakby czynią ją dłuższą.

Poprawnie więc ma być

```
void osoba::zapamietaj(...)
{ /*ciało*/ }
```

Funkcja zdefiniowana poza klasą przy zastosowaniu tego przedrostka ma dokładnie taki sam zakres, jakby była zdefiniowana wewnątrz klasy. Oba sposoby definiują funkcje o zakresie ważności klasy `osoba`.

W konsekwencji więc: niezależnie czy funkcja składowa zdefiniowana jest tym pierwszym czy drugim sposobem – ma jednakowy dostęp do wszystkich składników swojej klasy.

Jest jednak ogromna różnica dla kompilatora.

Jeśli bowiem funkcję składową zdefiniujemy wewnątrz definicji klasy (sposób 1) to kompilator uznaje, że chcemy, aby ta funkcja była typu `inline` (patrz str. 91)
Definicja funkcji składowej będąca poza definicją klasy (sposób 2) sprawia, że funkcja nie jest automatycznie uznawana jako `inline`.

Kiedy który sposób zatem wybrać ?

Umówmy się tak:

- Jeśli ciało funkcji składowej ma nie więcej niż dwie linijki, to funkcję tę definiujemy wewnątrz definicji klasy (sposób 1). Jest wtedy automatycznie `inline`.
- Jeśli funkcja składowa jest dłuższa niż te dwie linijki, to definiujemy ją poza definicą klasy.

Zapytasz pewnie: – Wobec tego funkcja składowa zdefiniowana poza definicją klasy nie może być nigdy typu `inline`?

Może! Tylko trzeba to wtedy wyraźnie zaznaczyć pisząc słówko `inline`:

```
inline void osoba::wypisz()
{
     //... ciało funkcji
}
```

A oto jak wygląda prosty program z użyciem klasy `osoba`:

```
#include <iostream.h>
#include <string.h>                                          // ❶
//////////////////////// definicja klasy ////////////////////
class osoba {
     char nazwisko[80] ;                                     // ❷
     int wiek ;
public :                                                     // ❸
     void zapamietaj(char * napis, int lata) ;               // ❹
     //————————————
     void wypisz()                                           // ❺
     {
          cout << "\t" << nazwisko << " , lat : "
               << wiek << endl ;
     }
} ;
//////////////////////// koniec definicji klasy ////////////////
```

```
void osoba::zapamietaj(char * napis, int lata)        // ❻
{
     strcpy(nazwisko, napis) ;
     wiek = lata ;
}
/*****************************************************/
main()
{
osoba student1, student2, profesor, pilot ;

     cout << "Dla informacji podaje, ze jeden obiekt "
           "klasy osoba\n ma rozmiar : "
          << sizeof(osoba)                            // ❼
          << " bajty. To samo inaczej : "
          << sizeof(student1) << endl ;

     profesor.zapamietaj("Albert Einstein", 55);      // ❽
     student1.zapamietaj("Ruediger Schubart", 26);
     student2.zapamietaj("Claudia Bach", 25);
     pilot.zapamietaj("Neil Armstrong", 37);

     cout << "Po wpisaniu informacji do obiektow. "
              "Sprawdzamy : \n";
     cout << "dane z obiektu profesor\n";
     profesor.wypisz();

     cout << "dane z obiektu student1\n";
     student1.wypisz();

     cout << "dane z obiektu student2\n";
     student2.wypisz();                               // ❾

     cout << "dane z obiektu pilot\n";
     pilot.wypisz();

     cout << "Podaj swoje nazwisko (tylko nazwisko) : " ;
     char magazynek[80] ;
     cin >> magazynek ;                               //❿

     cout << "Podaj swoj wiek : " ;
     int ile ;
     cin >> ile ;

     pilot.zapamietaj(magazynek , ile);             // ❶❶

     cout << "Oto dane ktore teraz sa zapamietane "
              "w obiektach profesor i pilot \n" ;

     profesor.wypisz() ;
     pilot.wypisz();                                // ❶❷
}
```

Oto co po wykonaniu programu zobaczymy na ekranie

```
Dla informacji podaje, ze jeden obiekt klasy osoba
 ma rozmiar : 82 bajty. To samo inaczej : 82          ❼
```

```
Po wpisaniu informacji do obiektow. Sprawdzamy :
dane z obiektu profesor
     Albert Einstein , lat : 55
dane z obiektu student1
     Ruediger Schubart , lat : 26
dane z obiektu student2
     Claudia Bach , lat : 25                                  ❾
dane z obiektu pilot
     Neil Armstrong , lat : 37
Podaj swoje nazwisko (tylko nazwisko) : Galileusz
Podaj swoj wiek : 60
Oto dane ktore teraz sa zapamietane w obiektach profesor
i pilot
     Albert Einstein , lat : 55
     Galileusz , lat : 60                                     ❶❷
```

Ciekawsze miejsca tego programu

❶ Ten plik nagłówkowy włączany jest z uwagi na to, że w programie posługujemy się funkcją biblioteczną `strcpy`. Jej deklaracja jest właśnie w tym pliku.

❷ Prywatne dane składowe klasy `osoba`. Prywatne przez domniemanie, bo do tej pory nie wystąpiła jeszcze w tej definicji klasy żadna etykieta określająca dostęp do składników.

❸ Etykieta mówiąca, że następujące po niej składniki będą miały dostęp `public`.

❹ Deklaracja funkcji składowej. Sama deklaracja.

❺ Definicja (a więc tym samym jednocześnie deklaracja) funkcji składowej `wypisz`. Funkcja jest tu zdefiniowana wewnątrz definicji klasy, a więc będzie typu `inline`.

❻ Definicja funkcji składowej będąca poza definicją klasy. Ponieważ jest tam operator zakresu `(osoba::zapamietaj)`, więc kompilator wie, że jest to definicja funkcji `zapamietaj`, będącej funkcją składową klasy `osoba`.

❼ Tu się przekonasz jak duży jest obiekt klasy `osoba`. Na ekranie widzisz, że jest to 82 bajty. Nic w tym dziwnego, że właśnie tyle – w klasie składnikiem jest przecież tablica 80 znakowa oraz zmienna typu `int` która ma rozmiar 2 bajty (w moim komputerze). W sumie jest to 82. Zauważ, że operator `sizeof` można zastosować do klasy czyli typu, jak też i do konkretnego egzemplarza obiektu. Podobnie jak dla typu `int`.

```
int licznik ;
sizeof(int)
sizeof(licznik)
```

❽ Wywołanie funkcji składowej `zapamietaj` na rzecz obiektu `profesor` – będącego egzemplarzem obiektu klasy `osoba`. Formę takiego wywołania już omawialiśmy.

❾ Przykład wywołania funkcji `wypisz`. Tu na rzecz obiektu o nazwie `student2`.

❿ Po wypisaniu tego, co jest w programie, program prosi o podanie Twoich danych, by je zapamiętać w obiekcie. Prosi nas tylko o nazwisko po to, by był to jeden wyraz.[††]

❶❶ – Otrzymane dane wkładamy do obiektu o nazwie `pilot`.

❶❷ – Na dowód, że to się udało – wypisujemy za chwilę na ekran.

10.8 Jak to właściwie jest ? (this)

Jak wiemy funkcja składowa może wykonywać operacje na danych składowych. W naszym programie z klasą `osoba` widzieliśmy taką definicję funkcji składowej

```
void osoba::zapamietaj(char * napis, int lata)
{
     strcpy(nazwisko, napis) ;
     wiek = lata ;
}
```

Wiemy też, że istnieje w pamięci kilka egzemplarzy obiektów klasy osoba:

```
osoba student1, student2, profesor, pilot ;
```

Są więc cztery tablice na nazwisko, cztery zmienne typu `int` na przechowywanie informacji o wieku. Jeśli przyjrzymy się funkcji składowej, to widzimy, że jest tam tylko nazwa `wiek` i nazwa `nazwisko`. Funkcja składowa, jak już wspomniałem, jest w pamięci tylko jednokrotnie – skąd więc wie ona do której komórki `wiek` ma w danym momencie coś wpisać?

```
wiek = 44 ;
```

Czy do wieku `studenta1` czy też `pilota`?

Odpowiedź jest prosta. Zwróć uwagę jak wywoływana jest funkcja

```
student2.zapamietaj("Ruediger Schubart" , 26);
```

Jak widać wywołana jest na rzecz jednego konkretnego egzemplarza obiektu `student2`. Bez naszej wiedzy do wnętrza funkcji przesyłany jest wskaźnik do tego konkretnego obiektu. Tym adresem funkcja inicjalizuje sobie swój wskaźnik zwany `this`.[†)] Wskaźnik ten pokazuje funkcji, na którym konkretnym egzemplarzu obiektu tej klasy, ma funkcja teraz pracować.

Jak to się odbywa, że funkcja pracuje rzeczywiście na danym konkretnym obiekcie?

Otóż wnętrze tej funkcji wygląda w rzeczywistości tak

```
void osoba::zapamietaj(char * napis, int lata)
{
     strcpy(this->nazwisko, napis) ;
```

†††) To jest wada tego prostego sposobu wczytywania. O tym, jak się wpisuje całe wielowyrazowe zdania – porozmawiamy w rozdziale: Operacje Wejścia/Wyjścia.

†) this – ang: ten (czytaj: „wys").

```
        this->wiek = lata ;
}
```

Widzimy wskaźnik `this` przed nazwą składników klasy. Ten wskaźnik moglibyśmy sami w tych miejscach tam postawić, ale kompilator oszczędza nam pisania i wstawia to sam. Gdybyśmy to mimo wszystko zrobili, to nie będzie to błędem.

Wskaźnik `this` stojący przed składnikami klasy sprawia, że operacje przeprowadzane są na składnikach tego (this) konkretnego egzemplarza obiektu, dla którego tę funkcję wywołaliśmy. Nie ma żadnej wieloznaczności.

Typ wskaźnika `this`

Zwykły wskaźnik mogący pokazywać na obiekty klasy X ma typ

```
X*
```

Wskaźnik `this` pokazuje na właśnie takie obiekty, ale dodatkowo nie wolno nim poruszać. Zatem wskaźnik ten ma typ

```
X const *
```

10.9 Odwołanie się do publicznych danych składowych

Dane składowe klasy mogą być zasadniczo prywatne `private` i publiczne `public`. (O trzecim typie – `protected` nie będziemy na razie mówić – w naszym przypadku składniki `protected` zachowują się jak `private`).

Oto przykład:

```
class owoc {
     int scisle_tajne ;
public :
     int  pestka ;
     void funkcja1() ;
} ;
```

Są tu dwie dane składowe. Składnik `scisle_tajne` jest prywatny. (Przez domniemanie.) Jako prywatny może być użyty tylko z zakresu klasy – czyli wewnątrz funkcji składowej `funkcja1`.

Natomiast składnik publiczny `pestka` oprócz tego, że może być dostępny w tej funkcji składowej `funkcja1`, dostępny jest także z zewnątrz klasy. Pracując jednak na nim z zewnątrz musimy podać, o który konkretny obiekt chodzi.

```
owoc cytryna, gruszka;                    // def dwóch obiektów klasy owoc

     cytryna.pestka = 16 ;
     gruszka.pestka = 12 ;
     cout << "Moja cytryna ma " << cytryna.pestka
          << " pestek" ;
     cout << "Moja gruszka ma " << gruszka.pestka
          << " pestek" ;
```

Nazwa, jak widać, uzupełniona jest nazwą konkretnego obiektu, o którego `pestke` chodzi.

natomiast gdybyśmy napisali tak

```
cytryna.scisle_tajne = 6 ;                    // Błąd !
```

to kompilator zaprotestuje. Wie on, że `scisle_tajne` jest składnikiem prywatnym klasy owoc, a więc nie wolno na nim robić żadnych operacji spoza zakresu tej klasy.

10.10 Zasłanianie nazw

Ponieważ nazwy składników klasy (danych i funkcji) mają zakres klasy, więc w obrębie klasy zasłaniają elementy o takiej samej nazwie leżące poza klasą.

Z danymi sprawa jest prosta. Zmienna

```
int ile ;
```

będąca składnikiem klasy zasłania w klasie ewentualną zmienną `ile` o zakresie globalnym lub lokalnym.

Po raz kolejny przypomnę, że zbytnie poleganie na zasłanianiu wprowadzi nas w tarapaty. Wcześniej czy później zapomnimy co, kiedy i przez co jest zasłaniane. Najlepiej po prostu wymyślać inne nazwy.

Wewnątrz klasy składnik o nazwie nnn zasłania inne obiekty o tej samej nazwie leżące poza klasą (czy to globalne, czy lokalne). Stają się one wtedy niedostępne. Można jednak mimo wszystko dostać się do zasłoniętej nazwy globalnej za pomocą operatora zakresu.

Możliwe jest jeszcze jedno piętro. Otóż wewnątrz zakresu klasy można także zdefiniować sobie jakiś zakres lokalny i w nim zdefiniować jakąś nazwę. Nazwa taka zasłoni wówczas nazwę składnika klasy. Pokażmy na przykładzie jak to się dzieje i jak można - mimo wszystko - do zasłoniętych nazw się odnieść. (Zauważ użycie operatora zakresu).

```
#include <iostream.h>

int balkon = 77 ;                    // nazwa globalna          // ❶
void spiew() ;                  // deklaracja jakiejś funkcji (globalnej)
//////////////////////// definicja klasy ////////////////////////
class opera {
public:
     int  n ;
     float balkon ;                                  // składnik klasy ❷
     // ...

     void      funkcja() ;
     void spiew()                                                   // ❸
     {
          cout << "funkcja spiew (z opery) : tra-la-la !\n" ;
     }
} ;
////////////////////// koniec definicji klasy ////////////////////
void opera::funkcja()                                               // ❹
```

```
{
                    // jeszcze się nic nie dzieje
     cout << "balkon (skladnik klasy) = "
          << balkon << endl ;                                  // ❺
     cout << "balkon (zmienna globalna) = "
          << ::balkon << endl ;


                    // definicja zmiennej lokalnej (lokalnej dla tej funkcji)
     char balkon = 'M' ;                                       // ❻

     cout << "\nPo definicji zmiennej lokalnej ---\n" ;
     cout << "balkon (zmienna lokalna) = " << balkon << endl;
     cout << "balkon (skladnik klasy) = " << opera::balkon
          << endl ;
     cout << "balkon (zmienna globalna) = "
          << ::balkon << endl ;

     //————————wywołanie funkcji
     spiew();                                                  // ❼

     int spiew ;                                               // ❽

     spiew = 7 ;                                               // ❾
     // spiew() ;          // <— błąd w trakcie kompilacji      ❿
                           //    bo nazwa funkcji - już zasłonięta

     cout<< "Po zasloniecu da sie wywolac "
              "funkcje spiew tylko tak\n" ;
     opera::spiew();                        // tak można        ❶❶
}
/*****************************************************/
main()
{
opera Lohengrin ;

     Lohengrin.balkon = 6 ;                          //   ❶❷
     Lohengrin.funkcja();                            //   ❶❸
     spiew();                                        //   ❶❹
}
/*****************************************************/
void spiew()
{
     cout << "zwykla funkcja spiew (nie majaca nic"
              " wspolnego z klasa)\n" ;
}
```

❒ **Po wykonaniu programu na ekranie zobaczymy**

```
balkon (skladnik klasy) = 6
balkon (zmienna globalna) = 77

Po definicji zmiennej lokalnej ---
balkon (zmienna lokalna) = M
balkon (skladnik klasy) = 6
balkon (zmienna globalna) = 77
```

```
funkcja spiew (z opery) : tra-la-la !
Po zaslonieciu da sie wywolac funkcje spiew tylko tak
funkcja spiew (z opery) : tra-la-la !
zwykla funkcja spiew (nie majaca nic wspolnego z klasa)
```

Skomentujmy ciekawsze miejsca programu

❶ Definicja zmiennej globalnej o nazwie `balkon` oraz deklaracja funkcji globalnej o nazwie `spiew`.

❷ Wewnątrz klasy `opera` definiujemy daną składową o nazwie `balkon`.

❸ Definicja funkcji `spiew` będącą funkcją składową klasy `opera`.

❹ Definicja funkcji `funkcja` będącej funkcją składową klasy `opera`. Dla odmiany definicja ta leży poza definicją klasy.

❺ Wewnątrz funkcji składowej klasy `opera` do składnika klasy odnosimy się po prostu:

```
balkon
```

a do zasłoniętej zmiennej globalnej (o tej samej nazwie)

```
::balkon
```

❻ Na scenie pojawia się jeszcze jeden obiekt o nazwie `balkon`. Tym razem jest to zmienna typu `char` zdefiniowana lokalnie – wewnątrz naszej funkcji `funkcja`. Ta nowa nazwa zasłania tutaj wszystkie pozostałe nazwy `balkon`. W następnych linijkach widzimy, jak należy teraz odnosić się do poszczególnych zmiennych o tej nazwie. Zauważ, że teraz użycie samej nazwy `balkon` dotyczy tego lokalnego obiektu – bo w tym lokalnym zakresie (funkcji) właśnie jesteśmy. Oto jak się do poszczególnych obiektów teraz odnosimy:

- `balkon`-dotyczy zmiennej lokalnej dla funkcji `funkcja` (tej typu `char`),
- `opera::balkon` – dotyczy danej składowej klasy `opera`,
- `::balkon` – dotyczy obiektu globalnego (tego typu `int`).

❼ Nazwa `spiew` określa funkcję składową klasy. Tutaj widzimy wywołanie tej funkcji.

❽ Definicja lokalnego obiektu typu `int` o nazwie `spiew`. Nazwa ta od tej pory zasłania (w tym lokalnym zakresie funkcji) nazwę `spiew` będącą nazwą funkcji składowej klasy `opera`.

> Zauważ ważną rzecz: nazwa zasłania inną nazwę – niezależnie czy ta inna nazwa jest nazwą zmiennej czy funkcji.

❾ Poprzez nazwę `spiew` odnosimy się teraz do obiektu lokalnego (typu `int`).

❿ Próba wywołania funkcji składowej `spiew` uznana zostanie za nielegalną. Kompilator zaprotestuje – informując nas, że nazwa `spiew` nie jest nazwą funkcji. Słusznie, bo nazwa tej funkcji została właśnie zasłonięta i jest niewidoczna. Aby kompilacja się udała – ta linijka musiała zostać umieszczona w komentarzu.

❶❶ Jeśli koniecznie chcemy wywołać tę funkcję składową, to musimy ją wzbogacić o operator zakresu. Zapis

```
opera::spiew()
```

wyraźnie określa, że chodzi o tę nazwę `spiew`, która leży w zakresie klasy `opera`.

❶❷ To jest przykład posłużenia się klasą `opera`. W poprzedniej linijce zdefiniowaliśmy obiekt klasy opera. Nadaliśmy mu nazwę `lohengrin`. Teraz do składnika `balkon` wpisujemy liczbę 6

❶❸ Wywołanie publicznej funkcji składowej klasy `opera`.

❶❹ W tym miejscu znajdujemy się poza zakresem ważności klasy. Odniesienie się w tym miejscu do nazwy `spiew` powoduje więc uruchomienie globalnej funkcji `spiew`. Tylko ta nazwa `spiew` jest w tym momencie ważna.

Ten cały program można w ludzkim języku opowiedzieć tak: Jesteśmy w domu – mówimy o wyjściu na balkon. Wiadomo o który balkon nam chodzi. Wieczorem idziemy do opery. W budynku klasy `opera` powiedzenie „balkon" odnosi się już do zupełnie innego balkonu. To określenie ma zakres ważności tego budynku i tutaj zasłania nasz domowy balkon. Jeśli jednak wejdziemy na scenę i gdzie zbudowana jest dekoracja, to powiedzenie „Julio, proszę wejść na balkon!" – odnosi się do jeszcze innego balkonu. Tego lokalnie zbudowanego na scenie. Gdy kurtyna spada kończy się zakres ważności lokalnego balkonu scenicznego i znowu oznacza on balkon na widowni- ten z klasy `opera`.

> Jeśli w swojej praktyce pogubisz się co, kiedy i przez co zostaje zasłaniane to miej pretensje do siebie. Ostrzegałem Cię. Jeśli tylko można, to używajmy innych nazw i nie polegajmy na zasłanianiu.

Widzieliśmy, że funkcja składowa klasy, zasłania funkcję globalną o tej samej nazwie. Powtarzam: **zasłania**! Nie ma tu mowy o żadnym przeładowaniu. Przeładowanie nazwy funkcji może nastąpić tylko wtedy, gdy funkcje mają ten sam zakres ważności. Funkcja nie-składowa ma zakres pliku, w którym jest zadeklarowana. Funkcja składowa ma zakres swojej klasy. Nie następuje więc przeładowanie tylko zasłonięcie.

Jeśli to jeszcze Cię nie przekonuje, to przyjrzyj się argumentom funkcji `spiew` tej globalnej i tej składowej. Są identyczne! (pusta lista) Taka sytuacja, że lista argumentów jest identyczna – nie jest możliwa przy przeładowaniu. Kompilator zameldowałby błąd. Ponieważ jednak jest to zasłonięcie, dlatego odmienność list argumentów przestaje być ważna.

10.11 Przeładowanie i zasłonięcie równocześnie

W naszej klasie `opera` jest jedna funkcja składowa `spiew`, która zasłania globalną funkcję `spiew`. Może być kilka funkcji składowych `spiew`. Wtedy – na obszarze klasy – ta nazwa `spiew` jest przeładowana. Wszystkie te funkcje

`spiew` mają bowiem ten sam zakres ważności – zakres klasy `opera`. Ta – przeładowana już teraz nazwa `spiew` zasłania nadal globalną nazwę `spiew`.

Inaczej mówiąc jeśli mamy takie funkcje

```
void spiew() ;                    // globalna

                                  // funkcje składowe klasy opera
void opera::spiew() ;
void opera::spiew(int) ;
void opera::spiew(float *) ;
void opera::spiew(char *) ;
```

to przy jakimkolwiek wywołaniu funkcji `spiew` z obszaru klasy `opera` - przy dopasowaniu wywołania do funkcji – brane będą pod uwagę tylko funkcje składowe. Funkcja globalna jest zasłonięta.

Przeładowanie funkcji składowych klasy to bardzo częsta praktyka. Oswoimy się z tym jeszcze.

10.12 Przesyłanie do funkcji argumentów będącymi obiektami

Jeśli użytkownik zdefiniował klasę, to oczywiście z myślą, że zdefiniuje konkretne egzemplarze obiektów tego typu. Tak jak projektant pralki ma nadzieję, że fabryka na podstawie jego projektu wyprodukuje kilka egzemplarzy obiektów klasy pralka. Definiujemy konkretny obiekt jakiejś klasy i mamy wtedy jakby nową zmienną: zmienną typu zdefiniowanego przez użytkownika.

Cały wysiłek twórców języka C++ streścił się w tym, by posługiwanie się takim obiektem było identyczne jak posługiwanie się obiektem typu `int` czy `float`. Między innymi musimy mieć możliwość wysłania go do funkcji jako argument.

Zaznaczam, że mówić tu będziemy jak argument wysyła się do funkcji w ogóle – niekoniecznie do funkcji składowych.

10.12.1 Przesyłanie obiektu przez wartość

Przez domniemanie zakłada się, że obiekt przesyłany jest do funkcji przez wartość. Czyli tak samo, jak to się odbywa z danymi typu `int` czy `float`.

Pokażemy to na przykładzie klasy `osoba`. Oto przykład programu. Zobaczymy tutaj także, jak postępować w przypadku, gdy program składa się z wielu plików.

Definicję klasy umieszcza się w pliku nagłówkowym.

```
//////////////////////////////////////////////////////////
// Plik osoba.h                                         ////
//////////////////////////////////////////////////////////
#include <iostream.h>
#include <string.h>
//////////////////// definicja klasy ////////////////////
class osoba {
     char nazwisko[80] ;
```

```
     int wiek ;
public :
     void zapamietaj(char * napis, int lata) ;
     //————————
     void wypisz()
     {
          cout << "\t" << nazwisko << " , lat : "
               << wiek << endl ;
     }
} ;
//////////////////// koniec definicji klasy ////////////////////
```

Jest to plik nagłówkowy dlatego, że w naszym programie włączany go do każdego z plików, w którym posługujemy się klasą `osoba`.

Już słyszę protesty: „-Jak to? To znaczy, że widoczna tutaj definicja (ciało) funkcji składowej `wypisz` występuje potem w programie wielokrotnie? – Jest w każdym z plików? Przecież wtedy na etapie łączenia linker zaprotestuje!"

Masz rację. Przypominam jednak, że definicja funkcji składowej, która znajduje się w obrębie definicji klasy – sprawia, że funkcja ta jest traktowana jako typu `inline`. Innymi słowy potem, w trakcie linkowania nie ma nigdzie funkcji `wypisz`. W każdym miejscu, gdzie wywołaliśmy funkcję `wypisz` kompilator wstawił po prostu linijkę

```
cout << "\t" << nazwisko << " , lat : " << wiek << endl ;
```

będącą ciałem tej funkcji.

Gdyby jednak funkcja składowa zdefiniowana była poza definicją klasy, to nie może znaleźć się w tym pliku nagłówkowym. Błędem by było na przykład wstawienie jej tu zaraz za końcem definicji klasy. Tak definiowana funkcja może już w programie pojawiać się tylko raz. Dlatego zwykle tworzy się osobny plik, w którym znajdują się zebrane definicje wszystkich takich funkcji składowych danej klasy.

Oto jak wygląda ten plik u nas:

```
//////////////////////////////////////////////////////////
// plik osoba.c                                         ////
//////////////////////////////////////////////////////////
#include <string.h>
#include "osoba.h"
/*****************************************************/
void osoba::zapamietaj(char * napis, int lata)
{
     strcpy(nazwisko, napis) ;
     wiek = lata ;
}
/*****************************************************/
```

Zwróć uwagę, że na początku włączamy plik nagłówkowy `osoba.h` Nic w tym dziwnego. Kompilator pracując nad tym plikiem musi już znać definicję klasy `osoba`. Inaczej od razu zaprotestowałby widząc zapis `osoba::zapamietaj` – bo co to jest `osoba`? Nic o tym nie wiem.

Wreszcie nasz właściwy plik programowy:

```
//////////////////////////////////////////////////////
// plik progr.c                                     ////
//////////////////////////////////////////////////////
#include <iostream.h>
#include "osoba.h"                                  // ❶
void prezentacja(osoba);                            // ❷
/*****************************************************/
main()
{
     osoba kompozytor, autor ;                      // ❸

     kompozytor.zapamietaj("Fryderyk Chopin", 36);
     autor.zapamietaj("Marcel Proust", 34);

     // wywołujemy funkcje, wysyłając obiekty
     prezentacja(kompozytor);                       // ❹
     prezentacja(autor);                            // ❼
}
/*****************************************************/
void prezentacja(osoba ktos)                        // ❺
{
     cout << "Mam zaszczyt przedstawic panstwu, \n"
              "Oto we wlasnej osobie :" ;
     ktos.wypisz();                                 // ❻
}
```

Po wykonaniu tego programu na ekranie pojawi się

```
Mam zaszczyt przedstawic panstwu,
Oto we wlasnej osobie : Fryderyk Chopin , lat : 36
Mam zaszczyt przedstawic panstwu,
Oto we wlasnej osobie : Marcel Proust , lat : 34
```

Komentarz

❶ Jeśli chcemy używać klasy `osoba`, to włączamy do programu jej definicję znajdującą się w pliku nagłówkowym, a także na etapie linkowania dołączamy do naszego programu skompilowany plik z definicjami funkcji składowych. To wszystko.

❷ Ta funkcja jest właściwie przedmiotem naszego przykładu. Jest to funkcja, do której wysyła się jeden argument będący obiektem klasy `osoba`. Deklarację czytamy: `prezentacja` jest funkcją wywoływaną z jednym argumentem klasy `osoba`. Funkcja zwraca typ `void` (czyli nic nie zwraca).

Zwracam uwagę, że jest to zwykła globalna funkcja – nie jest ona składnikiem żadnej klasy. To tylko jej argument jest dla nas interesujący.

❸ Definicja dwóch konkretnych obiektów klasy `osoba`. Zaraz potem wpisujemy do tych obiektów informacje, które mają przechowywać. To już od dawna znamy.

❹ Wywołujemy funkcję `prezentacja` wysyłając do niej argument `kompozytor` będący obiektem klasy `osoba`. Zauważ, że zapis jest taki sam jakbyśmy wysyłali obiekt typu `int`:

```
int a ;
     funkcja(a);
```

❺ Wysłany do funkcji `prezentacja` argument jest, jak już wiemy, wysyłany sposobem „przez wartość". Znaczy to, że funkcja tworzy sobie na stosie kopię obiektu `kompozytor` i nadaje tej kopii nazwę `ktos`. W tym momencie programu mamy już trzy obiekty klasy `osoba`: `kompozytor`, `autor` i `ktos`. Tak się składa, że ktoś ma identyczną informację jak `kompozytor`.

❻ Teraz na rzecz obiektu o nazwie `ktos` wywołuję funkcję składową `wypisz`. Jest to funkcja składowa klasy `osoba`, a obiekt `ktos` jest właśnie egzemplarzem obiektu klasy `osoba`, więc wszystko jest legalne.
Gdy opuszczamy funkcję `prezentacja` obiekt `ktos` – jako chwilowy przestaje istnieć. Znowu istnieją w programie tylko dwa obiekty klasy `osoba`.

❼ W `main` widzimy, że za chwilę wywołujemy `prezentacja` po raz drugi. Tym razem jako argument wysyłany jest obiekt o nazwie `autor`. Funkcja więc znowu tworzy na stosie obiekt klasy `osoba`. Obiektowi temu nadaje nazwę `ktos`. Obiekt ten mieści informację skopiowaną z obiektu o nazwie `autor`.
Dalej już tego nie będę opowiadał. Sytuacja jest identyczna, jak z wysyłaniem do funkcji obiektu typu `int`. Przypomnij sobie – dawno temu mówiliśmy o fotografii babci.

Wróćmy do sprawy: jeśli mamy jakąś klasę, to jej obiekty przez domniemanie wysłane są do funkcji przez wartość. Bardzo ważne jest uzmysłowienie sobie tego faktu.

Jeśli klasa zawiera trzy składniki – np. dane typu `int`, to wydaje się oczywiste, że przesłanie nastąpi w ten właśnie sposób. Jednakże, gdy weźmiemy pod uwagę klasę, która ma składnik będący tablicą 8192 elementową – to możesz o tym fakcie zapomnieć i zasugerować się, że przesłanie następuje tak, jak dla tablic – przez adres.

Zapamiętaj:

> Przez domniemanie obiekt wysyłany jest do funkcji przez wartość. To znaczy cały obiekt służy do inicjalizacji swojej kopii wewnątrz funkcji.

Wynika z tego ważna konsekwencja: Jeśli obiekt jest duży, to proces kopiowania może trwać dłużej. Wielokrotne wysłanie przez wartość może wyraźnie wpływać na zwolnienie programu.

10.12.2 Przesyłanie przez referencję

Wysyłanie przez wartość nie jest więc dobrym rozwiązaniem. Muszę się przyznać że, gdy kompilowałem ten przykładowy program, to kompilator ostrzegał mnie, iż wysyłam obiekt do funkcji przez wartość. Wolno tak przesyłać, ale nawet sam kompilator to odradza – Co robić?

Jest wyjście i to specjalnie wymyślone w tym celu. Nazywa się: przesyłanie przez referencję. Mówiliśmy o takim przesyłaniu w jednym z poprzednich rozdziałów (str. 83) obiecując sobie równocześnie nie nadużywać tego sposobu. Teraz nadeszła godzina używania referencji. Po to zostały wymyślone.

> Czym jest referencja w wypadku obiektu zdefiniowanego przez użytkownika? Oczywiście tym samym czym jest dla typów wbudowanych - czyli po prostu przezwiskiem. Nie przezwiskiem klasy, ale danego egzemplarza jej obiektu.

Wysyłając taki egzemplarz obiektu do funkcji na zasadzie przesłania przez referencję – sprawiamy, że nie jest on kopiowany. Funkcja ma dostęp do oryginału. Tyle, że w funkcji mówi się na niego używając przezwiska. Oto przykład funkcji `prezentacja` w wersji z przesłaniem przez referencję:

```
void prezentacja(osoba & ktos)
{
     cout << "Mam zaszczyt przedstawic panstwu, \n"
              "Oto we wlasnej osobie :" ;
     ktos.wypisz();
}
```

Przywykłeś już chyba, że jedyna różnica jaką należy zrobić w funkcji, to wstawienie znaku ampersand `&` przy definicji argumentu formalnego.

> *Jeśli już rzuciłeś się do przerabiania poprzedniego programu, to nieśmiało zaznaczę, że poprawkę należy zrobić nie tylko w definicji funkcji* `prezentacja` ❺. *Także w jej deklaracji, czyli w miejscu* ❷ *ma być:* `void prezentacja(osoba &);`
> *(Przepraszam za tak niski poziom tej uwagi)*

Zastosowanie przesłania przez referencję sprawia, że tym samym funkcja pracuje teraz na oryginale i nie przesyła do funkcji całego 82 bajtowego obiektu. (O tym, że obiekt klasy `osoba` ma u nas rozmiar 82 bajty przekonaliśmy się parę stron wcześniej).

Wewnątrz funkcji `prezentacja` nazwa `ktos` nie jest teraz nazwą kopii obiektu, ale po prostu przezwiskiem nadawanym temu – na kogo rzecz tę funkcję wywołano. Na przykład, gdy wywołaliśmy funkcję na rzecz `kompozytora`, to ten `ktos`, to jest `kompozytor`. Istniejąca w funkcji instrukcja

```
ktos.wypisz();
```

jest wywołaniem funkcji `wypisz` na rzecz samego `kompozytora` (a nie na rzecz jego kopii). Nie muszę dodawać, że tym sposobem funkcja `prezentacja` może nawet dokonać zmian na oryginale.

10.13 Konstruktor – pierwsza wzmianka

Załóżmy, że mamy następującą klasę

```
class numer {
     int  liczba ;
public:
     //————————————funkcje skladowe
     void schowaj(int l)
     {
          liczba = l ;
     }
```

```
        //————————
        int zwracaj() { return liczba ; }
} ;
```

W klasie mamy definicje dwu funkcji składowych. Druga definicja jest dla oszczędności miejsca zapisana w jednej linijce. To samo oczywiście można by zapisać w 4 linijkach.

Klasa, jak widać, jest schowkiem na liczbę typu `int`. Funkcja składowa `schowaj` wkłada liczbę do tego schowka, a funkcja składowa `zwracaj` – pokazuje co w schowku jest.

Jeśli chcieliśmy stworzyć sobie obiekt typu `int`, w którym schowamy liczbę 5, to do tej pory wystarczyła nam instrukcja

```
int skrytkaA = 5 ;
```

Gdy to samo chcielibyśmy zrobić posługując się naszą klasą `numer`, to potrzebny jest zapis

```
numer skrytkaB ;
     skrytkaB.schowaj(5);
```

Widzisz, że w momencie definicji obiektu `skrytkaB` obiekt nie jest od razu inicjalizowany. Aby znalazła się w nim wartość 5 musimy w następnej linijce posłużyć się funkcją składową `schowaj`.

Wynika z tego, że klasa `numer` nie jest tak wygodna w użyciu, jak zwykły typ wbudowany `int`.

A jednak celem moim jest przekonanie Cię, że typy definiowane przez użytkownika (klasy) są tak samo władne jak typy wbudowane.

Czy można definicję obiektu i nadanie mu wartości załatwić w jednej instrukcji?

Można. Rozwiązanie jest proste – posługujemy się w tym celu specjalną funkcją składową zwaną **konstruktorem**. Charakteryzuje się ona tym, że nazywa się tak samo jak klasa.

Oto ta sama klasa wyposażona w konstruktor:

```
class numer {
     int  liczba ;
public:
     //————————————————————————funkcje składowe
     numer(int l)   { liczba = l ; }     //<—— konstruktor

     void schowaj(int l) { liczba = l ;     }
     int  zwracaj()      { return liczba ; }
} ;
```

Zwróć uwagę, że przed konstruktorem nie ma żadnego określenia typu wartości zwracanej. Nie może być tam nawet typu `void`. Po prostu nie stoi tam nic. Oczywiście z faktu tego wynika, że w konstruktorze nie może wystąpić instrukcja `return` zwracająca jakąkolwiek wartość.

Oto jak w programie posługujemy się konstruktorem klasy `numer`:

```
numer a = numer(15) ;
```

drugi sposób jest taki

```
numer b(15) ;
```

Pierwszy sposób wydaje się dłuższy, ale za to bardziej zrozumiały. Definiujemy w pamięci obiekt a i dla niego wywołujemy konstruktor z argumentem 15

Druga definicja opuszcza znak równości i nazwę konstruktora – przecież nazwa konstruktora jest identyczna z nazwą klasy.

> W zasadzie w konstruktorze nie ma żadnych cudów – taka sobie funkcja składowa. Osobliwością jest to, że wywoływana jest ona automatycznie ilekroć powołujemy do życia nowy obiekt danej klasy.

Jeśli klasa `numer` wydaje Ci się zbyt prymitywna, to ją rozbudujmy. Przede wszystkim dodamy jeszcze składnik `nazwa` opisujący znaczenie liczby, którą tam przechowujemy.

```
#include <string.h>                    // bo użyjemy: strcpy()
#include <iostream.h>
/////////////////////////////////////////////////////////////
class numer {
     int  liczba ;
     char nazwa[40] ;
public:
     //————————————funkcje skladowe
     // konstruktor -tylko deklaracja
     numer(int l, char *opis) ;                          // ❶

     // dalsze funkcje skladowe
     void schowaj(int l)
     {
          liczba = l ;
          melduj() ;                                     // ❷
     }
     //————————
     int  zwracaj()      { return liczba ; }
     //————————
     void melduj()                                       // ❸
     {
          cout << nazwa << liczba << endl ;
     }
} ;
//////////////////// koniec definicji klasy ////////////////////////
numer::numer(int l, char *opis)                          // ❹
{
     liczba = l ;
     strcpy(nazwa, opis);
}
/*****************************************************/
main()
{
numer samolot(1200, "Biezaca wysokosc ") ;               // ❺
numer atmosfera(920, "Cisnienie atmosferyczne "),        // ❻
      kurs(63, "Kierunek lotu ");
```

```
        // wstępny raport

        samolot.melduj() ;
        kurs.melduj();                                  // ❼
        atmosfera.melduj();

        cout << "\nKorekta lotu ———\n" ;
        samolot.schowaj(1201);                          // ❽

        // zmiana kursu o 3 stopnie
        kurs.schowaj( kurs.zwracaj() + 3) ;             // ❾

        //ciśnienie spada
        atmosfera.schowaj(919) ;
}
```

Po wykonaniu tego programu na ekranie zobaczymy

```
Biezaca wysokosc 1200
Kierunek lotu 63
Cisnienie atmosferyczne 920

Korekta lotu ———
Biezaca wysokosc 1201
Kierunek lotu 66
Cisnienie atmosferyczne 919
```

Komentarz

❶ Ponieważ konstruktor nam się rozbudował, to postanowiliśmy umieścić jego definicję na zewnątrz definicji klasy. Wewnątrz definicji klasy jest tylko jego deklaracja. Raczej dla odmiany niż z konieczności.

❹ Z definicji tego konstruktora widzisz, że jako argument przysyłany jest do niego tekst. Tekst ten wkłada on do tablicy `nazwa` – będącej składnikiem klasy. Zwróć uwagę na nazwę konstruktora – `numer::numer`. To dlatego, że jest on funkcją składową klasy `numer`, a w dodatku sam (jako konstruktor) nazywa się tak, jak klasa – czyli `numer`. Przed nazwą konstruktora nie ma żadnego określenia typu zwracanego (ani `int`, ani `float`, ani nawet `void`).

❷ Funkcja składowa `schowaj` wywołuje teraz dodatkowo inną funkcję składową `melduj`.

❸ Definicja funkcji składowej `melduj`. Funkcja wypisuje informacje o tym, co przechowuje i jaką to ma wartość.

❺ Dla nas najważniejszym miejscem w tym programie jest użycie konstruktora. W sumie definiujemy trzy obiekty klasy `numer`. Robimy to na dwa sposoby.

❻ Drugi sposób – jest jakby odpowiednikiem zapisu

```
int  a = 5, b = 2;
```

❼ Użycie funkcji melduj wywoływanej na rzecz poszczególnych obiektów klasy `numer`.

❽ Do korekty danych używamy funkcji `schowaj`. Wewnątrz tej funkcji jest już wywołanie funkcji `melduj` tak, że nie musimy tego robić osobno.

❾ To jest nieco bardziej skomplikowane wywołanie funkcji `schowaj`. To samo można prościej zapisać jako:

```
int pomocnik ;
     pomocnik = kurs.zwracaj() + 3 ;
     kurs.schowaj(pomocnik) ;
```

Konstruktor jest funkcją, przy której najczęściej spotyka się przeładowanie nazwy

Inaczej mówiąc konstruktor może mieć kilka wariantów na różne ewentualności - różne zestawy argumentów inicjalizujących obiekt. To, który konstruktor zostanie uruchomiony, wynika (jak to zwykle przy przeładowaniu bywa) — z kolejności i typu argumentów wywołania.

Weźmy taką klasę:

```
class pomiar {
     int sec ;
public :

     pomiar(int sekundy) { sec = sekundy ; }
     pomiar(int minuty, int sekundy)
     {
          sec = sekundy + 60 * minuty ;
     }
     pomiar(int godziny, int minuty, int sekundy)
     {
          sec = sekundy + (60 * minuty) + (3600 *godziny);
     }
     //————
     // ... dalsze funkcje składowe

} ;
```

Załóżmy, że klasa ta służy nam do odmierzania czasu jakichś pomiarów. Rzut oka na konstruktory i widzimy, że powodują one wpisanie do składnika `sec` żądanego czasu w sekundach. Możliwe jest ewentualne przeliczanie godzin i minut na sekundy.

Ten zespół konstruktorów istnieje dla wygody, abyśmy używając tej klasy mogli wygodnie zadawać czas. Oto jak tworzymy i inicjalizujemy obiekty tej klasy:

```
pomiar temperat(5, 0) ;        // użyty konstruktor (int, int)
pomiar predkosc(10);           // użyty konstruktor (int)
pomiar widma(2, 30, 0) ;       // użyty konstruktor (int, int, int)
```

Każdy z tych trzech obiektów został inicjalizowany przy użyciu innego konstruktora. Na przykład obiekt `temperat` – konstruktorem `pomiar(int,int)`. Czas, który znajduje się w składniku `sec` tego konkretnego obiektu wynosi

```
5 * 60 = 300 sekund
```

Czy można by obiekt `temperat` zamiast tego inicjalizować jakimś innym konstruktorem? Oczywiście te różne warianty są po to, by sobie życie ułatwić. Inne sposoby inicjalizacji tego samego obiektu tym samym czasem to:

```
pomiar temperat(300);
```

albo

```
pomiar temp(0, 5, 0) ;
```

Oczywiście tylko jedna z tych ewentualności – konstruktor bowiem jest uruchamiany tylko raz – wtedy, gdy obiekt powoływany jest do życia.

Nazwa "konstruktor" może być nieco myląca

Konstruktor nie konstruuje obiektu tylko nadaje mu wartość początkową. W tym sensie może lepiej byłoby go nazywać dekoratorem wnętrz. Bezpośrednio po zbudowaniu domu wkracza, by go odpowiednio według życzeń klienta urządzić.

Czy konstruktor jest obowiązkowy? Nie. Najlepszy dowód, że do tej pory radziliśmy sobie bez niego. Można więc bez niego żyć. Ale po co?

Do sprawy konstruktora jeszcze wrócimy w jednym z najbliższych rozdziałów (str. 336), wtedy przyjrzymy mu się bliżej.

10.14 Destruktor – pierwsza wzmianka

Przeciwieństwem konstruktora jest destruktor: funkcja składowa wywoływana wtedy, gdy obiekt danej klasy ma być zlikwidowany.

Do tego, że niektóre obiekty są likwidowane, już chyba się przyzwyczaiłeś. Widzieliśmy to wielokrotnie w wypadku typów wbudowanych. Jeśli wewnątrz funkcji definiowaliśmy obiekt automatyczny typu `int`, to w momencie, gdy funkcja kończyła pracę – obiekt przestawał istnieć.

Dokładnie tak samo jest w wypadku obiektów typów definiowanych przez użytkownika. Wtedy właśnie automatycznie uruchamiany jest destruktor.

> Destruktor to jakby taka sprzątaczka, która sprząta na kilka sekund przed zlikwidowaniem obiektu. Najważniejsze jest to, że to nie my uruchamiamy tę sprzątaczkę. Przystępuje ona do pracy sama i robi to, co na tę okoliczność ustaliliśmy.

Destruktor to funkcja składowa klasy. Destruktor nazywa się tak samo, jak klasa z tym, że przed nazwą ma znak ~ (wężyk). Podobnie jak konstruktor - nie ma on określenia typu zwracanego.

Oto klasa wyposażona w destruktor. Destruktor ten niczego w zasadzie nie sprząta, tylko dużo mówi. Dzięki temu widzimy kiedy obiekty są likwidowane.

```
#include <string.h>          // bo użyjemy: strcpy()
#include <iostream.h>
//////////////////////////////////////////////////////////
class gadula {
```

```
     int  licz ;
     char tekst[40] ;
public :

     // konstruktor
     gadula(int k, char *opis) ;

     // destruktor - deklaracja
     ~gadula(void) ;                                        // ❶

     // inne funkcje składowe
     int zwracaj() { return licz ; }
     void schowaj(int x) { licz = x ; }
     void coto()
     { cout << tekst << " ma wartosc "<< licz << endl; }
} ;
//////////////////////////////////////////////////////////
gadula::gadula(int k, char *opis)      // konstruktor
{
     strcpy(tekst, opis);
     licz = k ;
     cout << "Konstruuje obiekt " << tekst << endl ;
}
/*******************************************************/
gadula::~gadula()                      // destruktor      ❷
{
     cout << "Pracuje destruktor (sprzata) "
          << tekst << endl ;
}
/*******************************************************/
gadula a(1, "obiekt a (GLOBALNY)");                        // ❸
gadula b(2, "obiekt b (GLOBALNY)");
/*******************************************************/
main()
{
     a.coto() ;
     b.coto() ;                                            // ❹
     {                                  // ←——— !           ❺
          cout << "Poczatek lokalnego zakresu --------\n";
          gadula c(30, "obiekt c (lokalny)");              // ❻
          gadula a(40, "obiekt a (lokalny)"); // zasłania !


          cout << "\nCo teraz mamy :\n" ;
          a.coto() ;                                       // ❼
          b.coto() ;
          c.coto() ;

          cout << "Do zaslonietego obiektu globaln mozna "
                   "sie jednak dostac\n" ;
          ::a.coto() ;                                     // ❽
          cout << "Konczy sie lokalny zakres ---------\n";
     }                                                     // ❾
     cout << "Juz jestem poza blokiem \n" ;
     a.coto() ;                                            // ❿
     b.coto() ;
```

```
    cout << "Sam uruchamiam destruktor obiektu a\n";
    a.gadula::~gadula() ;                          // ❶❶
    cout << "Koniec programu !!!!!!!!\n";          // ❶❷
}
```

Oto co zobaczymy na ekranie po wykonaniu tego programu

(Oznaczone punkty odpowiadają odpowiednio oznaczonym punktom w programie. Dodatkowo „wcięcie" -zostało przeze mnie zrobione sztucznie – po to, by wydruk stał się czytelniejszy.)

```
Konstruuje obiekt obiekt a (GLOBALNY)                ❸
Konstruuje obiekt obiekt b (GLOBALNY)
obiekt a (GLOBALNY) ma wartosc 1
obiekt b (GLOBALNY) ma wartosc 2                     ❹
     Poczatek lokalnego zakresu --------
     Konstruuje obiekt obiekt c (lokalny)            ❻
     Konstruuje obiekt obiekt a (lokalny)

     Co teraz mamy :
     obiekt a (lokalny) ma wartosc 40                ❼
     obiekt b (GLOBALNY) ma wartosc 2
     obiekt c (lokalny) ma wartosc 30
     Do zaslonietego obiektu globaln mozna sie jednak dostac
     obiekt a (GLOBALNY) ma wartosc 1                ❽
     Konczy sie lokalny zakres ---------
     Pracuje destruktor (sprzata) obiekt a (lokalny) ❾
     Pracuje destruktor (sprzata) obiekt c (lokalny)
Juz jestem poza blokiem
obiekt a (GLOBALNY) ma wartosc 1                     ❿
obiekt b (GLOBALNY) ma wartosc 2
Sam uruchamiam destruktor obiektu a
Pracuje destruktor (sprzata) obiekt a (GLOBALNY)     ❶❶
Koniec programu !!!!!!!!
Pracuje destruktor (sprzata) obiekt b (GLOBALNY)
Pracuje destruktor (sprzata) obiekt a (GLOBALNY)
```

Uwagi

❶ Deklaracja destruktora. Zauważ brak określenia typu zwracanego.

❷ Definicja destruktora. Destruktor ten jest funkcją składową klasy `gadula`, a nazwę ma `~gadula`. Razem więc mamy `gadula::~gadula()`
Temu destruktorowi nie dajemy zwykłego zadania sprzątania, ma on tylko informować nas, kiedy pracuje.

❸ Widzimy tu definicje dwóch obiektów klasy `gadula`. Obiekty te są zdefiniowane poza wszelkimi funkcjami, a więc są globalne. Na ekranie widzimy jak pracują ich konstruktory.

❹ Wewnątrz funkcji `main` te dwa obiekty są już gotowe i mogą się przedstawić.

❺ Za pomocą klamer definiujemy sztucznie pewien lokalny zakres.

❻ W tym lokalnym zakresie definiujemy dwa obiekty klasy `gadula`. Jeden ma nazwę `c`, a drugi `a`. Tym samym globalna nazwa `a` zostaje zasłonięta przez nazwę lokalną.

❼ Na dowód tego zasłonięcia – przy wypisywaniu nazwy `a` – widzimy, że pracujemy na obiekcie lokalnym. Tymczasem nazwa globalna `b` jest dostępna, bo w zakresie lokalnym nie zadeklarowano niczego o nazwie `b`, czyli nic nie zasłania jej.

❽ Pokazywałem już kiedyś ten sposób dostania się do zasłoniętej nazwy globalnej. Gdyby jednak zasłonięty obiekt globalny `a` nie był globalny, tylko zdefiniowany w funkcji `main`, to sprawa byłaby stracona – nie moglibyśmy się do tego obiektu tak długo dostać, jak długo trwało by zasłonięcie (czyli do końca lokalnego zakresu).

❾ Koniec lokalnego zakresu. Kończy się życie lokalnych obiektów `a` i `c`. Na ekranie widzisz, że zadziałały ich destruktory. Samodzielnie. Nie uruchamialiśmy ich żadną instrukcją.

❿ Odsłonięty został obiekt globalny `a`. Aby się do niego odwołać, nie trzeba już robić sztuczek z operatorem zakresu.

❶❶ Tu widzisz niezmiernie rzadką rzecz. Destruktor można uruchomić jawnie. Zrobi wtedy sprzątanie, ale oczywiście nie zlikwiduje obiektu. To jest tylko sprzątaczka.

❶❷ Kończy się program. Likwidowane są wszystkie istniejące jeszcze obiekty. Automatycznie uruchamiane są wtedy ich destruktory. Tu widzimy, że po raz drugi ruszył destruktor obiektu `a`. To nas upewnia, że wywołując przed chwilą jawnie destruktor – nie zlikwidowaliśmy tego obiektu, tylko zrobiliśmy sprzątanie.

Czy destruktor jest koniecznie potrzebny? Nie.

Do czego więc destruktor może się przydać?

Właśnie po to, by przed likwidacją obiektu posprzątać śmieci. Zaraz to wyjaśnię

- Przykład 1. Komputer ilustruje przeloty samolotów nad Europą. Obiektami są między innymi pojedyncze samoloty lecące nad danym obszarem. Jeśli samolot ląduje, to obiekt reprezentujący go jest likwidowany. Przed likwidacją należy zadbać o zmazanie go z ekranu.
- Przykład 2. Jeśli obiekt w trakcie swojego istnienia dokonał jakiejś rezerwacji w dostępnym zapasie pamięci, to przed likwidacją obiektu powinniśmy zwolnić tę rezerwację. Zrobić to najlepiej w destruktorze.
- Przykład 3. Jeśli obiektem jest menu narysowane na ekranie, a po wybraniu opcji menu jest likwidowane, to destruktor może posłużyć do odtworzenia poprzedniego wyglądu ekranu.

Do destruktorów powrócimy jeszcze w innym rozdziale (str. 347). Tutaj zasygnalizowaliśmy ich istnienie.

10.15 Składnik statyczny

Jak pamiętamy, każdy obiekt danej klasy ma swój własny zestaw danych. Wyobraźmy sobie klasę, która ma składnik-daną będącą liczbą `int`. Gdy zdefiniujemy tysiąc obiektów tej klasy, wówczas w pamięci będzie oczywiście tysiąc odrębnych miejsc odpowiadających temuż składnikowi. Dzięki temu, każdy egzemplarz obiektu może w tym składniku przechowywać sobie swoją informację.

Są jednak sytuacje, gdy poszczególne egzemplarze obiektów danej klasy powinny posługiwać się tą samą daną.

Potrzeba taka zachodzi wtedy, gdy dana ta dotyczy nie poszczególnych egzemplarzy obiektów tej klasy, ale samej klasy jako całości. (Cena pralki automatycznej danego typu [klasy] jest wspólna dla wszystkich pralek tego typu). Może być to na przykład zmienna określająca ile obiektów danej klasy powołano już do życia, może też być to zmienna określająca ile razy wykonano jakąś funkcję na wszystkich obiektach tej klasy (obrazowo: ile wyprodukowano pralek tego typu, lub ile razy wykonano na tych pralkach napraw gwarancyjnych). Te informacje są jakby bardziej informacją o klasie, a nie o konkretnym jej obiekcie.

Problem taki w klasycznym programowaniu rozwiązywało się za pomocą zmiennej globalnej. Nie jest to jednak rozwiązanie najlepsze, a to z dwóch powodów:

1) Zmienna globalna jest dostępna nie tylko dla obiektów naszej klasy, ale także dla obiektów innych klas ; i w ogóle dla każdego. W każdym bowiem miejscu programu można przecież posłużyć się zmienną globalną. Nie ma więc mowy o tym wspaniałym narzędziu jakim jest ukrywanie informacji.

2) Nawet jeśli informacja zawarta w takiej zmiennej nie jest ściśle tajna – namnażanie zmiennych globalnych nie jest elegancką praktyką. Trzeba wówczas dbać o to, by nazwa jaką nadajemy nowej danej globalnej nie kolidowała z już istniejącymi.

Co robić? Rozwiązaniem jest włączenie tej danej w obręb klasy w postaci *danej statycznej*.

> Dana, która jest określona jako statyczna ma tę ważną cechę, że jest w pamięci tworzona jednokrotnie i jest wspólna dla wszystkich egzemplarzy obiektów danej klasy. Co więcej: istnieje nawet wtedy, gdy jeszcze nie zdefiniowaliśmy ani jednego egzemplarza obiektu tej klasy.

Dzięki statycznym danym składowym można wydatnie zmniejszyć liczbę potrzebnych zmiennych globalnych.

Dana składowa staje się statyczna, gdy przed jej deklaracją w ciele klasy umieścimy słowo `static`.

```
class klasa {
public :
     int x ;
     static int skladnik ;
} ;
```

Deklaracja składnika statycznego w ciele klasy nie jest jego definicją. Definicję musimy umieścić gdzieś tak, by miała zakres pliku. Czyli tak, jakbyśmy umieszczali definicję zmiennej globalnej. Definicja taka może zawierać inicjalizację.

```
int klasa::skladnik = 6 ;
```

Składnik statyczny może być także typu `private`. Ciekawe, że taka inicjalizacja składnika statycznego możliwa jest nawet jeśli jest on typu `private`. Natomiast **po inicjalizacji** prywatny składnik statyczny nie może być przez obcych czytany ani zapisywany.

Są trzy sposoby odniesienia się do składnika statycznego:

- 1) Za pomocą nazwy klasy i operatora zakresu

  klasa::składnik

- 2) Jeśli istnieją już jakieś egzemplarze obiektów klasy, to możemy posłużyć się operatorem '.'

  obiekt.składnik

- 3) Jeśli jest zdefiniowany taki wskaźnik do obiektów klasy
  klasa * wsk ;
  to stosujemy operator ->

  wsk->składnik

  Nie muszę chyba przypominać, że wskaźnik trzeba wcześniej ustawić tak, by pokazywał na jakiś dowolny obiekt tej klasy. (Wszystko jedno na który, bo przecież dana statyczna jest wspólna dla wszystkich obiektów klasy – to tak, jakbyśmy powiedzieli – Ile cylindrów ma **ten** *samochód? – Niezależnie na który egzemplarz Mercedesa 220 pokażemy – i tak dowiemy się tego samego.*

Dwa ostatnie sposoby są więc identyczne jak w przypadku zwykłego składnika (nie-statycznego).

Zilustrujmy to przykładem

Zdefiniujemy sobie klasę opisującą pionki do gry. W klasie definiujemy dwa składniki statyczne. Jeden będzie mówił ile istnieje już egzemplarzy pionków (ten składnik będzie `public`), a drugi będzie zawierał informacją o tym, jaka jest pensja pionków (to będzie oczywiście `private`).

```
#include <iostream.h>
//////////////////////////////////////////////////////////
class pion {
private :
     int pozycja ;
     static int pensja ;                                  // ❶

public:
     static int ile_pionkow ;                             // ❷

     // funkcje składowe
```

```
        //-------
        pion() {                          // konstruktor
             pozycja = 0 ;
             ile_pionkow ++ ;                                  // ❸
        }
        //-------
        int przesun(int ile)                                   // ❹
        {
             return (pozycja += ile) ;
        }
        //-------
        int ile_zarabia() {                                    // ❺
             return pensja - 800;      // oszust !
        }
   } ;
   ////////////////////////////////////////////////////////////
   int pion::pensja = 3000 ;                                   // ❻
   int pion::ile_pionkow ;
   /*********************************************************/
   main()
   {
        cout << "Poczatek programu, teraz jest pionków = "
             << pion::ile_pionkow ;

        pion czerwony, zielony ;          //definicje pionków

        cout << "\nPo definicji pionkow \n" ;

        // odczytywanie informacji zapisanych w danej statycznej (public)

        cout << "Klasa mowi ze pionków jest "
             << pion::ile_pionkow << endl ;
        cout << "czerwony, ze "
             << czerwony.ile_pionkow << endl ;                 // ❼
        cout << "zielony, ze " << zielony.ile_pionkow << endl;

        pion bialy ;                      //definicja pionka

        cout << "Po definicji jeszcze jednego jest ich : "
             << zielony.ile_pionkow << endl ;                  // ❽

        // pionki idą do przodu, ale to nas nie interesuje
        zielony.przesun(2);
        czerwony.przesun(6);
        bialy.przesun(3);

        // interesuje nas ile zarabia pionek (dana
        // statyczna prywatna)
        /* cout << "pionek zarabia "
                  << pion::pensja; */ // błąd !                 ❾

        // jedyna szansa to zapytać pionka funkcją składową
                                                               // ❿
        cout << "Czerwony, ile zarabiacie ? "
             << czerwony.ile_zarabia() ;
        cout << "\nBialy, ile zarabiacie ? "
```

```
            << bialy.ile_zarabia() ;
}
```

Po wykonaniu programu na ekranie zobaczymy

```
Poczatek programu, teraz jest pionków = 0
Po definicji pionkow
Klasa mowi ze pionków jest 2
czerwony, ze 2
zielony, ze 2
Po definicji jeszcze jednego jest ich : 3
Czerwony, ile zarabiacie ? 2200
Bialy, ile zarabiacie ? 2200
```

Komentarz

❶ Deklaracja (tylko deklaracja) prywatnego składnika statycznego.

❷ Deklaracja (tylko deklaracja) publicznego składnika statycznego.

❸ To bardzo ciekawa rzecz. Wewnątrz konstruktora umieszczamy instrukcję inkrementującą daną `ile_pionków`. Konstruktor rusza do akcji wtedy, gdy definiujemy nowy pionek. Dzięki temu załatwiamy sobie licznik liczący ile obiektów klasy `pion` powołano do życia.

❹ Funkcja `przesun` symuluje ruch pionka. Jest dla nas tylko atrapą, nie będziemy się nią zajmowali.

❺ Funkcja, którą dowiadujemy się o wysokość zarobków pionka. Ta dana jest statyczna, ale `private`, więc z zewnątrz nie mamy do niej dostępu. Natomiast z wnętrza funkcji składowej jest ona dostępna tak, jak zwykła dana składowa. (Jak tutaj widzimy pionek oszukuje i nie udziela prawidłowej odpowiedzi).

❻ **Miejsce definicji danych statycznych.** Są one definiowane tu w miejscu o zasięgu globalnym. Poza wszelkimi funkcjami.

Zauważ, że:

- a) Nie ma tu słowa `static` (potrzebne było tylko wewnątrz klasy).
- b) Użyty jest tu operator zakresu – dzięki temu kompilator wie, że są to dane należące do określonej klasy. (Gdyby tego nie było, to pomyślałby, że to zwykłe obiekty globalne).
- c) Mimo, że `pensja` jest przecież składnikiem `private` - jest tutaj (poza zakresem ważności klasy) inicjalizowana. Inicjalizować wolno.
- d) Drugi składnik statyczny (ten publiczny) mógłby być równie dobrze inicjalizowany, ale nie robimy tego. W związku z tym podobnie jak zmienne globalne inicjalizowany jest automatycznie zerem.

❼ Trzy omówione sposoby odczytania składnika statycznego.

❽ Na dowód, że liczenie pionków działa, zdefiniowaliśmy jeszcze jeden pionek i znowu sprawdzamy.

❾ Ta instrukcja jest w komentarzu. Gdyby nie była – nastąpiłby błąd kompilacji. Składnik `pensja` jest statycznym prywatnym, więc kompilator nie udostępni

go niepowołanym. Można na nim pracować tylko za pomocą funkcji składowych klasy `pion`.

⑩ Wywołujemy więc funkcję składową, aby dowiedzieć się o zarobki. Innej drogi nie ma.

Jak widać, składnik statyczny klasy deklarowanej globalnie zachowuje się tak, jakby miał cechę `external` – czyli można do niego sięgnąć z innego pliku składającego się na ten program.

Klasy, które są definiowane lokalnie (czyli na przykład w obrębie jednej funkcji – nie mogą mieć danych statycznych).

Typ składnika statycznego nie jest wzbogacony o nazwę klasy. To znaczy jego typ jest taki, jakby został zdefiniowany jako zwykła zmienna globalna.

Dlatego nasze oba składniki `pensja`, `ile_pionkow`, są po prostu typu `int`. Gdyby nie były statyczne, byłyby typu `pion::int`.

Dla wtajemniczonych:

Składnik statyczny może pojawić się jako argument domniemany jakiejś funkcji składowej tej klasy. Składnik nie-statyczny (czyli zwykły) nie mógłby.

10.16 Statyczna funkcja składowa

Istnienie w klasie takich funkcji wynika z istnienia statycznych danych składowych.

Jeśli w klasie mamy taką funkcję składową, która pracuje tylko na składnikach będącymi składnikami statycznymi tej klasy, to taką funkcję *możemy* zdeklarować jako statyczną.

Co to w zasadzie zmienia ?

Spójrz na poniższą klasę. Jest to jakby prymitywna klasa symulująca ruch piórka plotera piszącego po papierze. Jazda ewentualnych obiektów klasy piórko (np. różnych kolorów) jest uwarunkowana wspólnym dla wszystkich piórek zezwoleniem na pracę.

```
#include <iostream.h>
#include <string.h>
enum tak_czy_nie { nie, tak } ;                          // ❶
//////////////////////////////////////////////////////////
class piorko {
     int poz_x, poz_y ;
     static int zezwolenie ;                             // ❷
     char kolor[30] ;

     // funkcje ————————————————————————————————
public :
     void jazda(int x , int y)
     {
          cout << "Tu "<< kolor << " piórko : " ;
          if(zezwolenie){                                // ❸
```

```
                poz_x = x ;
                poz_y = y ;
                cout << "Jade do punktu ("
                << poz_x << ", " << poz_y << ")\n" ;
            }
            else {
             cout <<
             " Nie wolno mi jechac !!!!!!!!!!!!! \n" ;
            }
        }
        //-----------------------------------------------
        static void mozna(tak_czy_nie odp)
        {
            zezwolenie = odp ;        // składnik statyczny
            // poz_x = 5 ;        // błąd ! bo składnik zwykły        ❹
        }
        // ----------------------------------------------
        piorko(char * kol)
        {
            strcpy(kolor, kol);
            poz_x = poz_y = 0 ;
        }
};
//////////////////////////////////////////////////////////
int piorko::zezwolenie ;
/*****************************************************/
main()
{
    piorko::mozna(tak) ;                                  // ❺

    piorko czarne("SMOLISTE"), zielone("ZIELONIUTKIE") ;
    czarne.jazda(0, 0) ;
    zielone.jazda(1100, 1100) ;

    // zabraniamy ruchu piórkom
    piorko::mozna(nie) ;

    czarne.jazda(10, 10) ;
    zielone.jazda(1020, 1020) ;

    // zezwalamy w taki sposób
    zielone.mozna(tak);                                   // ❻
    czarne.jazda(50, 50) ;
    zielone.jazda(1060, 1060) ;
}
```

Po wykonaniu programu na ekranie zobaczymy

```
Tu SMOLISTE piórko : Jade do punktu (0, 0)
Tu ZIELONIUTKIE piórko : Jade do punktu (1100, 1100)
Tu SMOLISTE piórko : Nie wolno mi jechac !!!!!!!!!!!!!
Tu ZIELONIUTKIE piórko : Nie wolno mi jechac !!!!!!!!!!!!!
Tu SMOLISTE piórko : Jade do punktu (50, 50)
Tu ZIELONIUTKIE piórko : Jade do punktu (1060, 1060)
```

Przyjrzyjmy się ciekawszym punktom programu

❶ Dla wygody definiujemy sobie taki typ wyliczeniowy.

❷ Statyczna dana składowa.

❸ Funkcja symulująca ruch piórka. Korzysta ze zwykłych danych składowych, a także ze składnika statycznego `zezwolenie`.

❹ Statyczna funkcja składowa. Może taką być, bo nie korzysta z nie-statycznych (czyli zwykłych) danych składowych tej klasy.
Jako argument przyjmuje typ wyliczeniowy, ale równie dobrze moglibyśmy ją tak zadeklarować, by przyjmowała liczbę 0 lub 1 będącą typu `int`. Nie ma to nic wspólnego z faktem czy funkcja jest statyczna czy nie.

Jeśli funkcja jest już zadeklarowana jako statyczna, to nie wolno w niej próbować odnosić się do składników zwykłych – kompilator zasygnalizuje błąd.

❺ **Najważniejsze miejsce w tym programie:** funkcję statyczną można wywołać nie tylko na rzecz jakiegoś obiektu danej klasy, ale także na rzecz samej klasy. Tu jest dowód. Wywołujemy funkcję na rzecz klasy, a jeszcze ani jeden obiekt takiej klasy nie został zdefiniowany.

❻ Oczywiście można także wywołać na rzecz konkretnego obiektu, ale to już znamy, do tego funkcja mogłaby być zwykłą funkcją składową.

Zadeklarowanie funkcji składowej jako statycznej sprawia, że nie zawiera ona wskaźnika `this`. Nie dotyczy ona wówczas konkretnego obiektu tylko klasy obiektów.

Co na tym tracimy ?

- Wewnątrz takiej funkcji nie jest możliwe jawne odwołanie się do wskaźnika `this`.
- Nie jest też możliwe odwołanie się do jakiegoś nie-statycznego (czyli zwykłego) składnika tej klasy. To dlatego, że dana będąca składnikiem ma zawsze przed sobą niewidoczny wskaźnik `this`.

Pamiętasz chyba, że:

```
if(zezwolenie){
     poz_x = x ;                    // to: this->poz_x = x ;
     poz_y = y ;                    // to: this->poz_y = y ;
}
```

Inaczej mówiąc: funkcja statyczna – nawet jeśli wywołana jest na rzecz konkretnego egzemplarza obiektu danej klasy – nie otrzymuje wskaźnika `this` do tego obiektu. Zatem nie może próbować pracować na danych składowych charakterystycznych dla tego właśnie egzemplarza. Dlatego błędem było w funkcji statycznej `mozna` użycie zapisu ❹ `poz_x=5;` bo kompilator nie mógł tego zastąpić zapisem

```
this->poz_x=5;
```

Na razie więc widzimy, że składowa funkcja statyczna jest gorszym rodzajem zwykłej funkcji składowej. A jednak nie!

Co zyskujemy przy statycznej funkcji składowej:

- To, że funkcja dotyczy nie tyle konkretnego obiektu, ale całej klasy tych obiektów. Można ją więc wywołać zarówno dla jednego obiektu

  ```
  obiekt1.funkcja() ;
  ```

  jak i dla klasy do której ona należy

  ```
  klasa::funkcja() ;
  ```

  Oczywiście, w wypadku wywołania dla konkretnego obiektu, funkcja interesuje się tylko tym do jakiej klasy wywołujący ją obiekt należy. Nie jest ważne, który to konkretny egzemplarz ją wywołuje.
- Funkcję można wywołać nawet wtedy, gdy nie istnieje jeszcze żaden obiekt tej klasy.

Ten ostatni powód wydaje mi się najważniejszy.

Czy funkcja statyczna rzeczywiście nie może odwołać się do żadnego zwykłego (nie-statycznego) składnika klasy?

Ostatecznie może. Nie ma co prawda wskaźnika `this`, który jej to normalnie umożliwia, ale możemy się do składnika klasy odwołać podając jawnie, o który egzemplarz obiektu nam chodzi i to zastąpi nieobecny wskaźnik `this`.

Wyjaśnijmy to. Jeśli chodzi nam o to, by w funkcji statycznej – odwołać się do składnika obiektu `obl`, a składnik się nazywa się `sss` to wewnątrz funkcji możemy się odnieść do niego jako

```
obl.sss
```

lub jeśli mamy wskaźnik pokazujący na ten obiekt to

```
wskaznik->sss
```

Już słyszę jak się wyśmiewasz: „– Tak to każdy potrafi! Przecież tym sposobem można do składnika odwołać się zawsze – nawet spoza klasy. Funkcja statyczna nie jest tu niczym lepszym niż jakakolwiek funkcja nie związana z klasą!"

Nie. Jest coś co ją odróżnia. Zapomniałeś, że jeśli jesteśmy w funkcji statycznej, to jesteśmy w zakresie ważności klasy. To oznacza, że możemy tym sposobem odnieść się do składnika `private`, co byłoby absolutnie niemożliwe ze zwykłej funkcji.

10.17 Do czego może nam się przydać składnik statyczny w klasie?

Oto kilka zastosowań:

- – Przydaje się, gdy egzemplarze obiektów danej klasy mają się porozumiewać ze sobą. Na przykład przez flagę – jeden obiekt może zawiadomić inny – (albo wszystkie inne) o jakimś fakcie.

- – Gdy wszystkie obiekty mają tę samą cechę, która może się zmieniać. Jeśli przykładowo obiekty danej klasy mają jakąś cenę, to podwyżka ceny polega na operacji zmiany zawartości jednego składnika statycznego. Nie trzeba żmudnie zmieniać w każdym poszczególnym obiekcie oczekującym na sprzedanie.
- – Może to być zmienna określająca ile obiektów danej klasy już istnieje. Licznik taki umieszcza się w obrębie klasy, której obiekty się liczy. Licznik jest jeden mimo, że obiektów mogą być tysiące.
- – Składnik statyczny może też służyć do tego, by wszystkie obiekty danej klasy mogły korzystać z jednego pliku dyskowego – przydzielonego tej klasie obiektów. Składnikiem statycznym jest wtedy kanał transmisji (strumień) do pliku.
- – Składnik statyczny może zaoszczędzić też miejsca. Jeśli mamy przykładowo 700 samolotów klasy "Samolot_z_Mojego_Towarzystwa_Lotniczego" to lepiej jeśli tę długą nazwę firmy (wspólną przecież dla wszystkich egzemplarzy) uczynimy składnikiem statycznym. Nazwa ta – będąc nadal przypisana do klasy – pojawi się w pamięci komputera tylko raz zamiast 700 razy.

Oczywiście innych zastosowań może być *multum*. Jeśli tylko zdobędziesz swobodę w posługiwaniu się składnikami statycznymi – z łatwością w swoich programach wyłowisz sytuacje kiedy uproszczą Ci życie.

10.18 Funkcje składowe typu `const` oraz `volatile`

Funkcje składowe danej klasy mogą być zadeklarowane z przydomkiem `const`. Po co? Sprawa jest prosta:

> Funkcja, która jest `const` to funkcja, która obiecuje, że jeśli się ją wywoła na rzecz jakiegoś obiektu, to nie będzie modyfikować jego danych składowych. Jest to ważne w sytuacjach, gdy zamierzamy w danej klasie definiować sobie obiekty typu `const`.

Jeśli dla typu `float` mogliśmy zdefiniować obiekt typu `const`

```
const float pi = 3.1416 ;
```

to także możemy sobie wyobrazić obiekty, które mają być stałe. Zawierają po prostu dane składowe ustalone raz na zawsze.

Jeśli obiekt danej klasy jest `const`, to na jego rzecz można wywołać jedynie te funkcje składowe, które zagwarantują, że go nie będą zmieniać – czyli tylko te, które mają przydomek `const`.

Po co ten przydomek? Czy kompilator nie wie, która funkcja modyfikuje, a która nie? Czy musimy dodatkowo to pisać? Tak, musimy, bo kompilator rzeczywiście nie wie. W trakcie kompilacji jakiegoś pliku, w którym używana jest klasa `K`, kompilator musi znać deklaracje jej funkcji składowych, natomiast same definicje tych funkcji mogą być – jak wiemy – w osobnym pliku. Po samych deklaracjach kompilator nie wie co robi funkcja w środku. Nie wie więc czy do

właśnie napotkanego stałego obiektu klasy K można jej użyć. Jeśli jednak przy deklaracji stoi słowo const to sprawa jest dla niego jasna.

Oto klasa pozycja opisująca pozycję czegoś w układzie współrzędnych:

```
#include <iostream.h>
//////////////////////////////////////////////////////////
class pozycja {
     int x ,
          y ;
public :
     pozycja (int a, int b ) {x = a ; y = b ; }
     void wypis (void) const ;                          // ❶
     void przesun(int a, int b);
} ;
//////////////////////////////////////////////////////////
void pozycja::wypis() const                             // ❷
{
     cout << x << ", " << y << endl ;
}
/*******************************************************/
void pozycja::przesun(int a , int b)
{
     x = a; y = b ;                                     //❸
}
/*******************************************************/
main()
{
     pozycja   samochod(40, 50),                        // ❹
               pies(30, 80) ;

     const pozycja dom(50, 50) ;                        // ❺

     // zastosowanie funkcji składowej - const

     samochod.wypis() ;
     pies.wypis() ;                                     // ❻
     dom.wypis() ;

     // zastosowanie funkcji nie-const

     samochod.przesun(4,10) ;
     pies.przesun(50, 50) ;
     // dom.przesun(0, 0) ;            // błąd !         ❼
}
```

Po wykonaniu tego programu na ekranie nie zobaczymy nic szczególnie ciekawego

Komentarz

❶ pozycja to klasa reprezentująca pozycję punktu w układzie współrzędnych. Jest w niej funkcja wypis, która tylko wypisuje dane na ekranie, a pozycji nie modyfikuje. Dlatego może być zadeklarowana jako const. Zauważ, że słowo const umieszczone jest na samym końcu deklaracji, tuż przed średnikiem.

❷ Podobnie przy definicji funkcji. Słowo `const` jest tuż przed klamrami otwierającymi definicję ciała funkcji.

❸ W klasie jest też funkcja `przesun`, która modyfikuje obiekt klasy `pozycja`. Skoro modyfikuje to oczywiście nie może być funkcją `const`.

❹ W `main` definiujemy dwa obiekty klasy `pozycja`. Jeden z nich reprezentuje pozycję samochodu, a drugi pozycję psa. Już same nazwy sugerują, że będziemy tymi obiektami „poruszać". W nawiasach widzimy argumenty dla konstruktora klasy `pozycja`. Te argumenty reprezentują wstępną pozycję tych obiektów.

❺ Definicja obiektu o nazwie `dom`. Postanawiamy, że tego obiektu nie wolno poruszać i dlatego stawiamy przy nim słówko `const`.

Dygresja:

Dawno nie czytaliśmy już definicji, więc przeczytajmy to na głos. Czytamy jak zwykle od tyłu: Dom jest obiektem klasy pozycja, a jest w dodatku (obiektem) stałym. Mam nadzieję, że nie dałeś się zwieść nawiasami dla konstruktora i nie zacząłeś czytać: dom jest funkcją... Tak by było, gdyby w nawiasie były nazwy typów, a nie zwykłe liczby – (czy konkretne obiekty).

❻ Wywołanie funkcji składowej `const` na rzecz obiektów zwykłych i obiektu `const`. Nie ma problemu. To, że funkcja obiecuje niczego nie zmieniać w obiekcie, nie przeszkadza obiektom zwykłym (`samochód`, `pies`), natomiast obiekt z przydomkiem `const` (dom) wprowadza w bardzo dobry nastrój.

❼ Inaczej wygląda sprawa w wypadku zwykłej (nie-`const`) funkcji składowej. Wywołanie jej na rzecz zwykłych obiektów jest zwykłą znaną nam rzeczą, natomiast kompilator nie dopuści do wywołania jej na rzecz obiektu `const`. Do przesuwania domu nie dojdzie.

Wszystko, co powiedzieliśmy o funkcjach składowych typu `const`, dotyczy także przydomka `volatile`

Jeśli obiekt klasy `K` zdefiniowaliśmy jako `volatile` to znaczy, że wymagamy dla niego troskliwszego traktowania. Dlatego kompilator odda go w ręce tylko takich funkcji składowych, które swoim przydomkiem `volatile` to lepsze traktowanie zapewnią.

Nie muszę chyba wyjaśniać, że `volatile` określa tu te funkcje, które zrezygnowały z optymalizacji i rzetelnie pracować będą zawsze na prawdziwych danych składowych sięgając po nie za każdym razem — nawet do najodleglejszych komórek pamięci. Mimo, że robiły to już linijkę wcześniej i dobrze pamiętają co tam jest.

Oczywiście taka troskliwa funkcja jest równie dobra dla obiektu zwykłego (nie-`volatile`).

Przykładu nie będę podawał – weź poprzedni przykład i zamień tylko słówko `const` na `volatile`. Pozycja słówka w deklaracji i definicji jest identyczna.

Czy funkcja składowa może być równocześnie `const` i `volatile`? Czy to ma sens?

Ma, bo to nie obiekt ma być `const` i `volatile` (co nie miało by sensu) ale funkcja. Jest to po prostu funkcja, która ma służyć do pracy na obu typach obiektów.

Z jednej strony obiecuje ona nic nie pisać, tylko czytać treść danych (to słowo `const`). Z drugiej strony obiecuje, że jeśli już będzie czytać, to zrobi to rzetelnie (`volatile`). Podwójna troskliwość.

Konstruktory i destruktory nie mogą mieć przydomków `const` i `volatile`, a mimo to nadają się do pracy na takich obiektach. Widać to nawet w naszym przykładzie. To oczywiście zrozumiałe. Konstruktor, aby zapisać wartość początkową do obiektu `const`, musi mieć możliwość pisania do niego. To jeszcze jeden dowód na to, że konstruktory i destruktory godne są tego, by im poświęcić osobny rozdział.

10.18.1 Przeładowanie a funkcje składowe `const` i `volatile`

W obrębie klasy funkcja może być przeładowana i niektóre wersje funkcji mogą mieć przydomek `const` lub `volatile`, a inne nie. Obowiązuje jednak stara zasada: funkcje o jednej nazwie – aby być przeładowane - muszą się różnić listą argumentów. Kropka. Obecność przydomków nie ma tu znaczenia.

Natomiast obecność przydomków ma znaczenie przy dopasowaniu wywołania do konkretnych wersji funkcji. Oto zasada: na rzecz obiektu `const` może być wywołana tylko funkcja składowa `const`. Wszystkie funkcje, które tego przydomka nie mają są od razu odrzucane. Dopiero spośród tych, które zostają (wszystkie `const`) próbuje się coś dopasowywać.

To samo dotyczy przeładowania funkcji z przydomkiem `volatile`.

11 Funkcje zaprzyjaźnione

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

Wyobraź sobie taką sytuację. W Twoim domu jest dużo roślin. Rośliny te są prywatnym składnikiem obiektu klasy dom.

Pewnego dnia wyjeżdżasz na wakacje na Majorkę. Chcesz jednak, by kwiatki Ci nie „zdechły". Masz dwa wyjścia:

1) 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.

2) Ewentualność druga: masz zaufanego przyjaciela. Dajesz mu klucze do swojego mieszkania i prosisz go, by kwiatki podlewał. Przyjaciel ma dostęp do wszystkich Twoich prywatnych składników (np. kolia brylantowa w komodzie), 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 wykopując przed drzwiami trzymetrowy dół i nakrywając go gałęziami.

Przyjaciel wpada w zasadzkę. Policja zaciera ręce, że złapała złodzieja. Co prawda przyjaciel z dna dołu krzyczy coś o przyjaźni, ale nikt go nie słucha. I słusznie, prawdziwy złodziej też to samo by krzyczał.

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 zatem wyciągnąć z dołu, bo jeszcze kwiatki zwiędną".

Przełóżmy 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 jednak sytuacjach 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` (ang. – przyjaciel [czytaj: „frend"]). Dzięki temu ta zwykła funkcja ma prawo dostępu do prywatnych składników klasy. Tak, jakby składniki te stały się dla niej publiczne.

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

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

Przykład funkcji zaprzyjaźnionej

Oto klasa pionek wzbogacona o następującą deklarację funkcji:

```
class pionek {
int x, y ;
     // dotychczasowe deklaracje
     // .........

     friend void raport(pionek &) ;
} ;
```

Sama funkcja jest gdzieś w programie zdefiniowana następująco:

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

}
```

Funkcja `raport` wywoływana jest z argumentem będącym referencją do obiektu typu `pionek`. (Moglibyśmy równie dobrze wysłać obiekt przez wartość, więc jeśli jeszcze nie oswoiłeś się z referencjami, to skreśl sobie znak & w pierwszej linii definicji funkcji – wtedy będzie to przesłanie przez wartość). Co jednak jest w tym wszystkim najważniejsze?

To mianowicie, że:

> wewnątrz tej zaprzyjaźnionej funkcji `raport` odwołujemy się do prywatnych składników obiektu klasy `pionek`. Tak, jakby te składniki były publiczne. To wszystko.

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

```
pionek niebieski ;          // def 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 wypadku. Lepiej mieć funkcję jako składnik, bo łatwiej wtedy panować na nią. Tak samo, jak lepiej by domownik podlewał kwiatki, niż przychodził w tym celu ktoś obcy.

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

Oto te cechy:

- Funkcja może być przyjacielem więcej niż jednej klasy. Czyli może mieć dostęp do prywatnych składników kilku klas.
- Funkcja zaprzyjaźniona może na argumentach jej wywołania dokonywać konwersji zdefiniowanych przez użytkownika.

  O tym wspaniałym mechaniźmie jeszcze nie mówiliśmy, więc brzmi to tajemniczo. Sprawą zajmiemy się w jednym z następnych rozdziałów. Tu wystarczy przyjąć, że funkcja zaprzyjaźniona jest bardziej uniwersalna niż zwykła funkcja składowa.

  Po prostu: Funkcję składową można wywołać na rzecz jakiegoś obiektu jej klasy, ale nie na rzecz np. liczby 5. Funkcja zaprzyjaźniona może przyjąć jako argument nawet tę liczbę 5. Pod warunkiem, że użytkownik określił jak się liczbę 5 zamienia na obiekt danej klasy.
- Dzięki funkcjom zaprzyjaźnionym możemy nadać dostęp do prywatnych składników klasy nawet takim funkcjom, które nie mogłyby być funkcjami składowymi z powodów zasadniczych. Na przykład dlatego, że są napisane w zupełnie innym języku programowania – np. w assemblerze, Pascalu czy FORTRANIE.

Zilustrujmy przykładem pierwszą cechę: możliwość przyjaźni z kilkoma klasami

Mamy dwie klasy. Klasa `punkt` opisuje współrzędne jakiegoś punktu. Klasa `kwadrat` opisuje współrzędne lewego dolnego rogu prostokąta i długość jego boku. Obie klasy przyjaźnią się z funkcją `sedzia`.

```
#include <iostream.h>
#include <string.h>
//-------------------------
class kwadrat ;                 // deklaracja zapowiadająca          ❶
//////////////////////////////////////////////////////////////////////
class punkt {
     int x, y ;
     char nazwa[30] ;
public :
     punkt(int a, int b,char *opis) ;
     void ruch(int n, int m) { x += n ; y += m ; }
     // ... może coś jeszcze ...

     friend int sedzia(punkt & p, kwadrat & k) ;     // ❷
} ;
```

```
//////////////////////////////////////////////////////////
class kwadrat {
     int x, y,
          bok ;
     char nazwa[30] ;
public :
     kwadrat(int a, int b, int dd, char *opis) ;
     // ... może coś jeszcze ...

     friend int sedzia (punkt & p, kwadrat & k) ;    // ❸
} ;
//////////////////////////////////////////////////////////
punkt::punkt(int a, int b,char *opis)        // konstruktor
{
     x = a ; y = b ;
     strcpy(nazwa, opis) ;
}
/*********************************************************/
kwadrat::kwadrat(int a, int b, int dd, char *opis)
                                             // konstruktor
{    x = a ; y = b ;
     bok = dd ;
     strcpy(nazwa, opis) ;
}
/*********************************************************/
// z tą funkcją przyjaźnią się obie klasy
int sedzia (punkt & pt, kwadrat & kw)                // ❹
{
     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 ;
     }
}
/*********************************************************/
main()
{
     kwadrat   bo(10,10, 40, "boiska") ;
     punkt     pi( 20, 20, "pilka");

     sedzia(pi, bo );                                    // ❺
     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 punktom programu

❷ Wewnątrz definicji klasy `punkt` widzimy deklarację przyjaźni. Klasa `punkt` stwierdza tutaj, że ma zaufanie do takiej funkcji.
Bardzo ważna uwaga:

> zauważ, że na liście argumentów jest nazwa klasy `kwadrat`. Do tej pory klasa ta jeszcze nie została zdefiniowana. Pamiętamy jednak, że w C++ każda nazwa zanim zostanie użyta po raz pierwszy musi zostać zdeklarowana.

Jak ten problem rozwiązać?

❶ Oto rozwiązanie. Jest to tak zwana **deklaracja zapowiadająca** (zwiastująca). Mówi ona: „Jakby co, to nazwa `kwadrat` jest nazwą klasy". To wszystko. Nie ma tu nic więcej na temat wewnętrznej struktury klasy, ale to nie szkodzi, bo w momencie deklaracji przyjaźni te detale nie są jeszcze potrzebne.

❸ Deklaracja przyjaźni w drugiej klasie. Podobnie: klasa `kwadrat` stwierdza tutaj, że ma zaufanie do funkcji `sedzia`.

❹ Definicja funkcji sędzia. Jest zdefiniowana jak najzwyklejsza funkcja. Różnica polega tylko na tym, że funkcja ta pracuje sobie na prywatnych składnikach obu klas tak, jakby były one publiczne. Czym zajmuje się funkcja `sedzia`? Łatwo się zorientować, że po prostu sprawdza czy obiekt klasy `punkt` leży na tle obiektu klasy `kwadrat`. Robi to przez porównanie współrzędnych punktu z obszarem zajmowanym przez obiekt klasy `kwadrat`.

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

```
obiekt.funkcja( )
```

bowiem funkcja – nie jest funkcją składową żadnej klasy.

Trzeba pamiętać o kilku jeszcze sprawach:

Funkcja zaprzyjaźniona nie jest składnikiem klasy, dlatego też nie ma wskaźnika `this`. Dla nas oznacza to, że funkcja ta, aby odnieść się do składnika klasy, z którą się przyjaźni – musi posłużyć się operatorem '.'
lub operatorem ->

> W naszym ostatnim przykładzie celowo współrzędne w obu klasach nazwałem identycznie, by ani przez myśl Ci nie przeszło, że

możesz odnieść się do nich bezpośrednio. Bez określenia obiektu nie wiadomo byłoby czy chodzi o składnik `x` z obiektu klasy `kwadrat` czy z obiektu klasy `punkt`. Musimy to określić stosując zapis

obiekt.składnik czyli u nas: `pt.x` lub `kw.x`

Nawet, gdyby nazwy składników byłyby zupełnie różne, to trzeba stosować ten zapis. Nie można pominąć nazwy obiektu skoro nie ma wskaźnika `this`.

Powtarzam więc wniosek:

> Funkcja zaprzyjaźniona to zwykła funkcja, którą jedynie wyjątkowo nie obowiązują słowa `private` i `protected` w klasach, które się z nią przyjaźnią.

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

Może się zdarzyć, że wewnątrz klasy będzie nie tylko deklaracja funkcji zaprzyjaźnionej, ale wręcz jej definicja (czyli całe ciało funkcji). Ma to następujące konsekwencje:

- funkcja ta jest typu `inline`
- funkcja jest nadal tylko przyjacielem, a nie składnikiem
- tak definiowana funkcja leży w zakresie *leksykalnym* deklaracji tej klasy ; leksykalnym – czyli oznacza to, że można w definicji tej zaprzyjaźnionej funkcji:
  a) korzystać z właśnie obowiązujących (wewnątrz deklaracji tej klasy) instrukcji `typedef`,
  b) możemy używać zdefiniowanych w tej klasie typów wyliczeniowych `enum`.

W wypadku funkcji przeładowanych, przyjacielem klasy `K` jest tylko ta wersja funkcji, która odpowiada liście argumentów widocznej w deklaracji przyjaźni w definicji danej klasy `K`

```
class K {
     // ...
     friend void alarm(K obj, int k) ;
} ;

void alarm(float*, K obiekt);
void alarm(void) ;
void alarm(K obiekt, int i);        // to jest przyjaciel klasy K
```

Funkcja zaprzyjaźniona może być zwykłą funkcją, a może być też funkcją składową zupełnie innej klasy. Oczywiście ma wtedy dostęp do prywatnych składników swojej klasy, a także tej, z którą się przyjaźni.

Oto tak zmodyfikowany poprzedni przykład, że funkcja sędzia jest składnikiem klasy `kwadrat`, a przyjaźni się z klasą `punkt`

```
#include <iostream.h>
#include <string.h>
//---------------------------
class punkt ;              // deklaracja zapowiadająca                ❶
//////////////////////////////////////////////////////////////
class kwadrat {
     int x, y,
          bok ;
     char nazwa[30] ;
public :
     kwadrat(int a, int b, int dd, char *opis) ;
     // ... może coś jeszcze ...

     int sedzia (punkt & p) ;                                  // ❷
} ;
//////////////////////////////////////////////////////////////
class punkt {
     int x, y ;
     char nazwa[30] ;
public :
     punkt(int a, int b,char *opis) ;
     void ruch(int n, int m) { x += n ; y += m ; }
     // ... może coś jeszcze ...

     friend int kwadrat::sedzia(punkt & p) ;                   // ❸
} ;
//////////////////////////////////////////////////////////////
punkt::punkt(int a, int b,char *opis)        // konstruktor
{
     x = a ; y = b ;
     strcpy(nazwa, opis) ;
}
/*************************************************************/
kwadrat::kwadrat(int a, int b, int dd, char *opis)
                                                  // konstruktor
{    x = a ; y = b ;
     bok = dd ;
     strcpy(nazwa, opis) ;
}
/*************************************************************/
int kwadrat::sedzia (punkt & 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 ;
        }
}
/*****************************************************/
main()
{
    kwadrat   bo(10,10, 40, "boiska") ;
    punkt     pi( 20, 20, "pilka");
    bo.sedzia(pi);                                  // ❻
}
```

Po wykonaniu programu na ekranie pojawi się

```
pilka lezy na tle boiska
```

Komentarz

❷ Deklaracja, zwykłej funkcji składowej w klasie `kwadrat`. Argumentem jest obiekt klasy `punkt`, stąd też konieczna była deklaracja zapowiadająca tę klasę ❶.

❸ Deklaracja przyjaźni. Tutaj jednak ważna uwaga. Jeśli przyjacielem ma być *funkcja składowa z innej klasy*, to ta klasa musi być już w tym momencie kompilatorowi znana. Dlatego najpierw w programie jest deklaracja klasy `kwadrat` (z funkcją `sedzia`), a potem dopiero deklaracja klasy punkt i niniejsze ogłoszenie przyjaźni. Odwrotna kolejność byłaby błędna.

❹ Oto definicja funkcji `sedzia`. Już na pewno przy deklaracjach zauważyłeś, że zmieniła się lista argumentów. Teraz argumentem jest tylko obiekt klasy `punkt`. A co z obiektem klasy `kwadrat`, funkcja go przecież także potrzebuje?! Zapominasz, że teraz funkcja jest funkcją składową klasy kwadrat, a więc jest wywoływana na rzecz obiektu klasy kwadrat. Zresztą spójrz poniżej.

❻ Tak właśnie w `main` wywołujemy funkcję sędzia. Obiekt klasy `punkt` wysyłany jest jako argument, a obiekt klasy `kwadrat` przez ukryty wskaźnik `this`.

❺ Z faktu, iż funkcja sędzia jest funkcją składową klasy `kwadrat` 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 obiektu klasy zaprzyjaźnionej `punkt` stosujemy tu zapis

```
pt.x
```

ale w stosunku do swojej własnej klasy

```
x,
bok          (dawniej bylo konieczne kw.x,  kw.bok)
```

To dlatego, że przecież gdy piszemy `x`, to jest tam naprawdę `this->x`
„–Coś kręcisz!" – zawołałeś zapewne – „parę stron wcześniej wmawiałeś mi, że funkcja zaprzyjaźniona z klasą nie zawiera wskaźnika `this`!"

> Podtrzymuję to! Funkcja `F` zaprzyjaźniona z klasą `K` nie zawiera wskaźnika `this` do klasy `K`, z którą sama się przyjaźni (-bo nie jest składnikiem tej klasy). Jeśli jednak sama funkcja `F` jest zwykłą

funkcją składową jakiejś innej klasy, to wskaźnik `this` do obiektu swojej klasy zawiera. Za jego pomocą pracuje przecież na swoich własnych prywatnych składnikach.

Brzmi to zawile, więc posłużmy się analogią z podlewaniem kwiatków

Załóżmy, że to Ty jesteś osobą podlewająca kwiatki i Twoja znajoma Perfidia zadeklarowała przyjaźń z Tobą. Masz jeszcze także innego znajomego Onufrego, który prosił Cię o to samo. (To sytuacja, kiedy funkcja globalna jest zaprzyjaźniona z dwoma klasami).

Gdy określasz swoje czynności to mówisz: „Idę podlać kwiatki Onufrego" albo „idę podlać kwiatki Perfidii". Jak do tej pory rzeczywiście nie ma wskaźnika `this`. Mówisz przecież o składnikach tych klas

```
Onufry.kwiatki
Perfidia.kwiatki
```

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

Mówisz „podlewam kwiatki Onufrego" albo „podlewam kwiatki Perfidii"

```
Perfidia.kwiatki
Onufry.kwiatki
```

Możesz jednak powiedzieć też „podlewam kwiatki" myśląc o podlewaniu swoich własnych kwiatków,

`kwiatki` czyli `this->kwiatki`

gdzie `this` oznacza wskaźnik do obiektu „mój_dom"

> Podsumujmy: funkcja zaprzyjaźniona nie zawiera wskaźnika `this` do obiektów klas, z którymi się przyjaźni. Do swojej własnej może. Nie jest się przecież przyjacielem swojego własnego domu, tylko po prostu domownikiem (składnikiem).

Jeśli projektujesz program i widzisz, że przydało by się, żeby funkcja miała dostęp do składników prywatnych dwu klas, to masz do wyboru:

- – funkcja jest przyjacielem dwu klas
- – funkcja jest składnikiem jednej, a przyjacielem drugiej

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. (O przeładowaniu operatorów – str. 422).

Klasy zaprzyjaźnione

Klasa `K` może deklarować przyjaźń z więcej niż jedną funkcją składową klasy `M`. Może nawet deklarować przyjaźń ze wszystkimi funkcjami tej klasy `M`. Jest to

tak, jakbyś wytrwale zadeklarował przyjaźń klasy K z każdą funkcją składową klasy M. Sprawa jest prosta, ale dużo pisania. Zamiast tego możemy zadeklarować, że cała klasa M jest uznawana za przyjaciela klasy K

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

Jak widać chodzi o umieszczenie takiej deklaracji:

```
friend class nazwa_klasy ;
```

w ciele klasy, która przyjaźń deklaruje. Od tej pory wszystkie funkcje składowe klasy M mają dostęp do prywatnych składników klasy K deklarującej tę przyjaźń. Przyjaźń jest oczywiście jednostronna. Wyraża ją klasa K, natomiast klasa M 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. Jedynym sposobem zadeklarowania takiej przyjaźni jest właśnie deklaracja przyjaźni z klasą jako całością – sposobem jaki właśnie pokazaliśmy.

> Nie ma możliwości zdeklarowania w jednej klasie, że przyjaźni się ona z funkcjami innej klasy, a w tej innej klasie – przyjaźni 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.

Żebyśmy się nie wiem jak gimnastykowali, to zawsze definicja jednej klasy będzie znana wcześniej od drugiej, bo tak przecież piszemy tekst programu. Siłą rzeczy ta 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 znać wnętrze klasy.)

Błędne koło? I tak i nie. W sposób, o którym mówimy rzeczywiście nie da się tego zrobić, ale problem rozwiązuje właśnie deklaracja przyjaźni nie z konkretnymi funkcjami klasy, ale z całą klasą.

```
class druga ;                    // ← dekl. zapowiadająca
//////////////////////////////////////////////////
class pierwsza {
     friend class druga ;
          // ... reszta ciała klasy pierwszej
};
//////////////////////////////////////////////////
class druga {
     friend class pierwsza ;
          // ... 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 – należy w klasie `A` zamieścić deklarację takiej przyjaźni

```
friend class C ;
```

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

Słowo o zakresie

Zastanówmy się jak to jest, gdy deklarujemy przyjaźń ze zwykłą funkcją, nie będącą składnikiem żadnej klasy. Jeśli funkcja taka w momencie deklaracji przyjaźni jest jeszcze kompilatorowi nieznana, wówczas zakłada on, że jest to funkcja leżąca w tym samym zakresie, w którym jest definicja klasy. Czyli jeśli definicja klasy jest globalna, to funkcja – zakłada się – jest też globalna.

Jeśli więc w dalszej części pliku programu okaże się funkcja, o której mowa, zdefiniowana jest jako statyczna – czyli znana jest tylko w zakresie ważności jednego pliku – wówczas linker zaprotestuje, że funkcji nie znalazł.

Konwencja umieszczania deklaracji przyjaźni w klasie

Spotyka się często konwencje 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.

Na koniec jeszcze jedna uwaga. Zapamiętajmy, że:

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

Uwaga dla wtajemniczonych

Jak wiesz klasa może mieć „potomstwo" (klasy pochodne). Przyjaźń nie jest dziedziczna. (Podobnie jak nie jest przechodnia). Jeśli jakaś klasa chce mieć przyjaciela, to powinna to powiedzieć wyraźnie sama.

12 Struktury, Unie, Pola bitowe

W rozdziale tym mówić będziemy o rzeczach, które są jakby obok właściwego toku tej książki. Potraktuj ten rozdział jako rodzaj odprężenia.

12.1 Struktura

To będzie chyba najkrótszy z paragrafów.

> Struktura to po prostu klasa, w której przez domniemanie wszystkie składniki są publiczne.

W zwykłej klasie – jeśli za pomocą słów `private`, `protected`, `public` nie określiliśmy tego inaczej – składniki mają dostęp typu `private`. Tutaj jest odwrotnie. Jeśli nie określamy tego inaczej – składniki są `public`.

A zatem definicja

```
struct nazwa {
        // lista składników
};
```

odpowiada definicji

```
class nazwa {
public:
        // lista składników
} ;
```

Uwaga dla programistów klasycznego C

Jak wiesz w klasycznym C mieliśmy również struktury. Struktury w C++ są tak zrobione (i po to zrobione), żeby ułatwić przejście z C do C++. Oczywiście – jak już zdążyłeś się zorientować – struktura (czyli klasa) w C++ jest o wiele mądrzejsza niż w klasycznym C, choćby dlatego, że może mieć funkcje składowe. Nic to jednak nie przeszkadza starym strukturom. Mogą być przecież także klasy bez funkcji składowych.

Krótko mówiąc jest to dobra wiadomość: Twoje programy w C posługujące się strukturami są najłatwiejsze do przerobienia na C++.

12.2 Unia

Przypomnij sobie jak to było w dzieciństwie: Miałeś takie pudełko, w którym trzymałeś swój największy skarb. Czasem był to opalizujący kamyk, czasem samochodzik, czasem zdechła żaba. Pudełko to służyło Ci do pomieszczenia tylko tego jednego przedmiotu. Często zmieniałeś zdanie i coraz to inny przedmiot był Twoim największym skarbem. Jedno było pewne: pudełko było na tyle duże, by pomieścić największy z przedmiotów.

> Ta analogia ilustruje pojęcie unii: Unia w języku C++ to właśnie takie pudełko do przechowywania jednego przedmiotu. Cokolwiek by to nie było.

Jeśli masz w pamięci operacyjnej jakiś obszar przeznaczony na trzymanie liczby typu `float`, to nie możesz tam normalnie wpisać znaku czy liczby `int`. Jeśli jednak posłużysz się unią to możesz: do pudełka wkłada się (w to samo miejsce) równie dobrze gumę do żucia jak i zdechłą żabę.

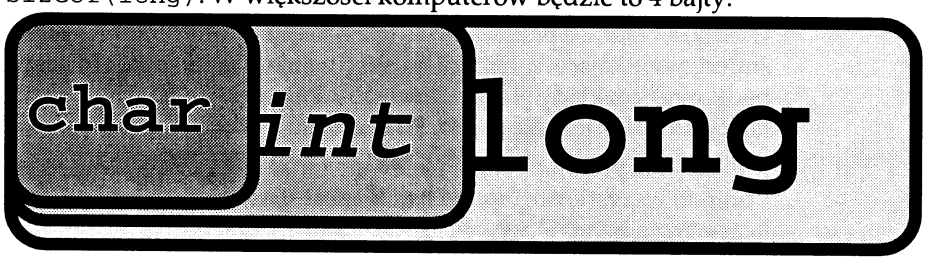
Dzięki unii możemy sprawić, że w tym samym miejscu w pamięci trzymane będą obiekty różnego typu. Oczywiście tylko jeden w danym momencie. Unia więc pozwala zaoszczędzić pamięć. Rozmiar unii wynika z rozmiaru największego z obiektów, do których przechowywania ma służyć.

Oto przykład:

```
union skarbiec {
     int m ;
     long l ;
     char z ;
} ;
```

Zdefiniowaliśmy unię, w której można przechowywać albo liczbę typu `int`, liczbę typu `long`, albo znak. Jedno z trojga.

Unia ta ma rozmiar odpowiadający rozmiarowi największego składnika - czyli `sizeof(long)`. W większości komputerów będzie to 4 bajty.



Zdefiniujmy sobie egzemplarz obiektu takiej unii

```
skarbiec sss ;
```

Od tej chwili w pamięci począwszy od tej samej komórki może zostać zapisana dana typu `int`, `long` albo `char`.

Odnoszenie się do składników unii przypomina odnoszenie się do składników klasy. Robimy to więc za pomocą operatora '.' (kropka) – który właśnie zapewnia nam odnoszenie się do składników obiektów.

```
sss.i = 5 ;                    // wpis
cout << sss.i ;                // odczyt (np. na ekran)
```

Tak zapisaliśmy lub odczytaliśmy liczbę typu `int`. Następna instrukcja:

```
sss.z = 'm' ;
```

powoduje zapisanie do unii litery `'m'`, przy czym dotychczas będąca tam liczba 5 zostaje przez ten zapis zniszczona. Literę tę możemy odczytać na ekran tak:

```
cout << sss.z ;
```

Oczywiście odczytanie w celu wypisania na ekran jest tylko przykładem. Można zrobić z tym co się chce: można przypisać innej zmiennej lub wysłać jako argument wywołania funkcji

```
char c ;
funkcja(char);

     c = sss.z ;
     funkcja(sss.z) ;
```

Trzeba pamiętać, jaki typ danej zapisuje się do unii, i taki sam typ odczytać

To zrozumiałe: jeśli się zapisuje do pudełka daną typu „martwy płaz" to nie należy go odczytywać jako typu „przedmiot jadalny", bo w rezultacie takiej pomyłki żuć będziemy zdechłą żabę. Oto ilustracja:

```
sss.z = 'x' ;
cout << "liczba long z unii = " << sss.l ;
```

Do unii zapisaliśmy znak. Jak pamiętamy rozmiar znaku to 1 bajt. Tam właśnie odbył się zapis. W rezultacie mamy tam na poszczególnych bitach tego bajtu taki rozkład zer i jedynek, który odpowiada znakowi 'x'. Nasza unia ma, jak wiemy, 4 bajty. Te pozostałe 3 są teraz nie używane. Co tam jest? Obecnie są tam śmieci wynikające z historii posługiwania się tą unią. To jest jakiś – (przypadkowy) rozkład bitów. Chociażby kawałek danej `int`, a spod niego wystający kawałek jeszcze starszej danej typu `long`.

Jeśli teraz zapomnimy, że wpisaliśmy tam znak i odczytamy to jako liczbę typu `long`, to wszystkie te bity z 4 bajtów zostaną zinterpretowane jako zakodowana liczba typu `long`. Oczywiście otrzymamy bezsens.

Gdybyśmy chcieli odczytać znak – to bity pierwszego bajtu zostaną zinterpretowane jako znak – co da nam wynik poprawny.

Do składników unii możemy także odnosić się przez wskaźnik (operatorem ->) tak samo, jak do składników klasy.

Unia jest bardzo podobna do struktury z tym, że o ile składniki struktury umieszczane są w kolejnych miejscach w pamięci – w unii umieszczane są w tym samym miejscu. Jeden **na** drugim.

> Napisałem „podobna do struktury" a nie: „do klasy", bo domniemywa się, że składniki unii są publiczne, o ile nie określimy tego inaczej.

Dla wtajemniczonych:

Kilka szczegółów wybiegających w przyszłe zagadnienia:

- Tak jak w klasach i strukturach, możemy tu używać słów określających dostęp: `private`, `public`. Słowo `protected` jest niedopuszczalne. To dlatego, że służy ono celom dziedziczenia, a unia nie może mieć dziedzica.
- Składnikiem unii nie może zostać obiekt klasy, która ma konstruktor lub destruktor.
- Składnikiem unii nie może zostać także obiekt klasy, która ma operator przypisania ('=') zdefiniowany przez użytkownika.
- Unia nie może mieć funkcji wirtualnej. (Bo po co, skoro i tak nie może mieć potomstwa?)

12.2.1 Inicjalizacja unii

Jeśli unia nie ma konstruktora, to można ją inicjalizować, pamiętając jednak o zasadzie, że inicjalizuje się ją tylko daną odpowiadającą typowi pierwszego składnika z jej listy składników.

```
union skrytka {
     char c ;
     float pi ;
} ;

skrytka moja = { 'z' } ;
skryka twoja = { 'x' } ;

skrytka jego = 3.14 ;    // <——jest błędem
```

Ostatnia linijka to błąd, bo pierwszy składnik jest przecież typu `char`.

12.2.2 Unia anonimowa

Unia anonimowa jest to taka unia, która jako „klasa" nie ma swojej nazwy, a także nie ma nazwy jedyny jej egzemplarz.

```
union {
     int licz ;
     float     wspol ;
     char      znak ;
     int       *wsk ;
} ;
```

Do składników takiej unii odnosimy się po prostu podając nazwę tego składnika. Nie trzeba operatora '.' (kropka).

```
licz = 4 ;
cout << licz ;
wspol = 6.26 ;
```

Posługujemy się więc tymi składnikami tak, jakby były to zwykłe nazwy zmiennych. (Trzeba jednak zawsze pamiętać, że mamy do czynienia z unią, zatem możemy przechowywać tam tylko jedną daną w określonej chwili).

Co na tym zyskujemy ?

Oszczędność miejsca (wynika to z unii) oraz prostotę notacji. Odnosząc się do składnika nie musimy pisać

obiekt.składnik

tylko po prostu

składnik

(to wynika z faktu anonimowości).

Jasne jest, że skoro teraz do nazwy składników tej unii odnosimy się tak samo, jak do najzwyklejszych nazw zmiennych, to te nazwy nie mogą być identyczne. Jeśli istnieje już zmienna `aaa`, to składnik jakiejś anonimowej unii (w tym samym zakresie ważności) nie może mieć już nazwy `aaa`.

```
int aaa ;
union {
     float aaa ;          // błąd - redefinicja nazwy aaa
     char zzz ;
} ;
```

Z faktu, iż do odnoszenia się do składników unii anonimowej nie używamy notacji z kropką wynika, że nie są sprawdzane prawa dostępu do danego składnika. Wszystkie składniki mają więc dostęp `public`. Dlatego nie można deklarować składnika takiej unii jako `private`.

Z tego samego powodu nie może być w takiej unii funkcji składowych. Funkcje składowe wywoływane są przecież na rzecz jakiegoś obiektu. Tu nie ma ani nazwy obiektu ani nazwy samej unii.

Jeśli zdefiniujemy unię tak

```
union {
     int i ;
     float pi ;
} egz, *wsk ;
```

to unia nie jest już anonimowa. Co prawda typ tej unii nie ma nazwy, ale jest konkretny egzemplarz obiektu tej unii – i ma on nazwę `egz`. Dodatkowo zdefiniowaliśmy wskaźnik `wsk` mogący pokazywać na tak określoną unię.

> Unia, która mimo, że nie ma nazwy typu, ale ma jakiś obiekt, albo ma jakiś wskaźnik mogący na nią pokazywać – nie jest już unią anonimową.

Przy odnoszeniu się do takiej unii obowiązują zwykłe reguły:
operator kropka '.' – odniesienie się do składnika obiektu, albo
operator -> czyli odniesienie się do składnika pokazywanego wskaźnikiem.

12.3 Pola bitowe

Pola bitowe to specjalny typ składnika klasy polegający na tym, że informacja jest przechowywana na określonej liczbie bitów. Oto przykład

```
class port {
     // ...
     unsigned int odczyt : 1 ;
     // ...
} ;
```

Odczyt jest składnikiem klasy `port`. Można w nim zapisać informację 1 bitową czyli, jak się łatwo domyślić, informację w rodzaju: tak/nie. Typ składnika jest `unsigned int`.

W ogólności pole bitowe może mieć jakikolwiek typ całkowity, a więc `char`, `short`, `int`, `long` – w obu wariantach : `signed` albo `unsigned`.

Potem następuje nazwa tego pola, a potem dwukropek i liczba określająca na ilu bitach ma być przechowywana ta informacja.

Oto przykład klasy, w której składnikami jest kilka pól bitowych:

```
class portA {
     // ...
     unsigned odczyt    : 1 ;
     unsigned tryb      : 2 ;

     unsigned gotow     : 1 ,
              dana      : 4 ;
     // ...
} ;
```

Są jak widać cztery składniki będące polami bitowymi. W sumie więc całość informacji, która ma być zapisana wymaga 8 bitów (1+2+1+4).

To, jak rozmieszczone są w pamięci komputera te dane, zależy od implementacji. Mogą znaleźć się po prostu w jednym bajcie, ale nie muszą. Mogą się także „przełamywać" z jednego bajtu na następny. To znaczy np. z czterech bitów przypadających jednemu polu: jeden bit jest w jednym bajcie, a pozostałe w następnym.

Także zależne od implementacji jest to, czy przydzielone nam bity zajmą kolejno miejsca od począwszy najbardziej znaczącego bitu w słowie, czy od najmniej znaczącego.

Jeśli napiszemy, że pole bitowe ma być typu `int` bez określenia czy `signed/unsigned` to to, czy otrzymamy `signed` czy `unsigned` – również zależy od implementacji.

Skoro tyle niepewności to co za korzyść ?

Najczęściej autorzy piszący o polach bitowych przekonują, że pozwalają ono gęsto upakować dane, co daje oszczędność pamięci. Zapomnij o takiej argumentacji. Rzeczywiście informacja jest upakowana gęsto, ale spróbuj wyobrazić sobie, że potrzebujesz odnieść się w naszej klasie do składnika `tryb`. Komputer pobiera więc określony bajt, w którym ten bit tkwi, po czym musi – że tak powiem – „odcedzić" żądane 2 bity od pozostałych 6 bitów.

Sprawa nie jest trudna – wykonuje się na przykład jedną operację bitowego iloczynu logicznego (operator &), a wynik przesuwa się o pewną liczbę miejsc w prawo. Niby to proste, ale trzeba to zrobić. To trwa, a kod tej akcji „odcedzania" może zająć więcej bajtów niż zaoszczędziliśmy pakując informację do pól bitowych.

Oszczędności pamięci więc raczej nie ma, zatem gdzie zalety ?

Wyobraź sobie, że do twojego komputera podłączony jest układ sprzęgający z jakimś zewnętrznym urządzeniem (ang: interface). Na przykład sterem kierunku samolotu, który oprogramowujesz.

Taki układ jest w pamięci naszego komputera reprezentowany przez kilka komórek. W tych komórkach interface właśnie tak gęsto upakowana jest informacja. Upakowanie jest takie po to, by zminimalizować komunikację z urządzeniem – (szybciej się to prześle).

Poszczególne bity w tych komórkach mogą oznaczać na przykład to, czy ster ma się wychylić w lewo czy w prawo, następne bity o ile stopni i tak dalej.

Przez fakt pracy z takim urządzeniem zewnętrznym jesteśmy skazani na pracę z danymi, które zajmują określone bity. Nie ma problemu: **do tego właśnie przydają się pola bitowe!**

Zapytasz pewnie: „–No, ale co mi po narzędziu, które nie wiadomo jak się zachowa, bo w tylu punktach jest *zależne od implementacji* !"

Rzeczywiście, ale przecież możesz łatwo ustalić jak postępuje Twój komputer. Wystarczy jeśli przyglądniesz się temu obszarowi pamięci.

„–Da mi to program niemożliwy do przeniesienia na inny typ komputera!" – zawołasz pewnie.

Kochany, jak będziesz chciał pomachać sobie sterem kierunku za pomocą innego komputera, to najpierw musisz zamówić do niego nowy, właściwy układ sprzęgający za parę tysięcy dolarów. Poprawka w programie przy polach bitowych to przy tym pestka! Piszesz przecież programy do współpracy z jakimiś zewnętrznymi układami elektronicznymi. Nie wymagaj od takiego programu, żeby mógł ruszyć bez zmian na każdym typie komputera.

A tak naprawdę, to jest zawsze odwrotnie: najpierw masz problem „ster kierunku", a dopiero potem do niego dobierasz komputer, który zrobi to najlepiej. Wtedy już piszesz program na **ten** ster kierunku i na **ten** komputer. Koniec kazania.

Rzeczywistość nie jest jednak taka czarna.

Mimo dowolności implementacji zwykle obowiązują jakieś rozsądne reguły

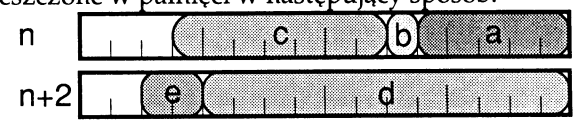
A więc:

- Dane z pól bitowych pakowane są jedna obok drugiej, w kolejności w jakiej deklarujesz je w klasie.
- To, czy pakowanie zaczyna się od prawego czy lewego brzegu słowa (najmniej- i najbardziej znaczące bity słowa), rzeczywiście zależy od implementacji. Z mojego doświadczenia wynika, że najczęściej od prawego brzegu słowa.
- Jeśli jakieś pole bitowe np. 5 bitowe nie mieści się już w całości w słowie, do którego pakowano do tej pory – wówczas nie jest ono przełamywane – lecz raczej jest w całości umieszczane od początku następnego słowa. Ostatnie bity tamtego słowa pozostają niewykorzystane.

Jeśli Twój komputer postąpi według takich reguł, to pola bitowe z poniższej klasy:

```
class sprzeg {
     unsigned  a : 5 ,
               b : 1 ,
               c : 7 ,
               d : 12,
               e : 2 ;
} ;
```

zostają rozmieszczone w pamięci w następujący sposób:

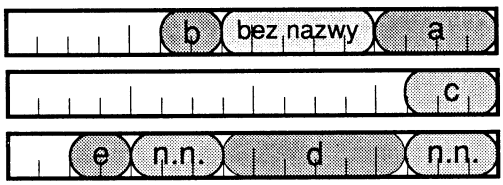


(zakładam, że posługujemy się komputerem 16 bitowym).

Do dokładnego rozmieszczania pól bitowych w słowach pamięci posłużyć się można polami bez nazwy. Służą one jako „wypełniacz" nadający odstęp dwu sąsiednim polom bitowym.

Natomiast takie nienazwane pole bitowe, które ma szerokość 0 bitów - jest sugestią, by następne pole bitowe znalazło się w następnym słowie.

Jeżeli więc w wypadku tego samego komputera chcemy otrzymać następujący rozkład bitów:



to tak definiujemy sobie pola bitowe:

```
class wzor {
public:                                          // ❶
     unsigned        a : 4 ,
                       : 5 ,
                     b : 2 ,
                       : 0 ,
                     c : 3 ,
                       : 0 ,
```

```
                    : 3 ;

        unsigned    d : 6 ,                    // ❸
                    : 3 ,
                    e : 2 ;
    } ;
```

Pola bitowe są składnikiem klasy, więc odnosimy się do nich identycznie, jak do innych składników klasy.

```
wzor obj ;          // definicja obiektu tej klasy
obj.c = 2 ;         // wstawienie liczby 2 do pola o nazwie c      // ❷
```

To tylko komputer musi się gimnastykować z ich „odcedzaniem" – nas to nie obchodzi.

To słowo `public` ❶ nie było obowiązkowe, ale skoro w ❷ chciałem podstawić tam coś z zewnątrz klasy, to musiałem uczynić pole `c` składnikiem publicznym.

Oczywiście składnikiem klasy jest każde pole osobno, a nie cała ta litania jako całość. Aby to pokazać w punkcie ❸ bez powodu tę litanię przerwałem. To tak, jakbym w wypadku zwykłych składników napisał:

```
int  n,
     o,
     p ;

int  q,
     r ;
```

Pole jest więc zwykłym składnikiem klasy, z tą różnicą, że nie można pobrać adresu pola bitowego (jednoargumentowym operatorem & – adres). To wydaje się oczywiste – w komputerze adresowane są słowa lub bajty - nie adresuje się pojedynczych bitów.

> *Mówimy: „bit 4 w słowie o adresie 146728", a nie: „bit numer 8457291663456 w tym komputerze".*

Z faktu, iż nie można określać adresu pola bitowego wynika, że nie można też na to pole pokazać wskaźnikiem.

Pole bitowe nie może też mieć referencji (przezwiska) – bo referencja, to jakby rodzaj adresu.

Na zakończenie dodam, że czasem rzeczywiście pola bitowe pozwalają nam zaoszczędzić pamięć. Mimo tego, że - jak to powiedziałem - kod "odcedzania" żądanych bitów zajmuje dodatkowo pamięć.

Wyobraź sobie sytuację, gdy potrzebujemy pracować z ogromną liczbą danych jednego rodzaju. Na przykłąd tysiace próbek - a w każdej z nich jakieś zjawisko zaszło lub nie zaszło. Informację taką możemy zapisać jednym bicie.

Danych tych mają być setki, czy nawet tysiące. Oczywiście wówczas sprytnie umieścimy je w polach bitowych. Ogromna ilość danych zostanie wówczas gęsto upakowana.

A co ze stratą pamięci, którą pochłonie kod wydobywający tak upakowane informacje ? Żaden problem, przecież napiszemy sobie funkcję, która obsłuży nam to zadanie. Próbka numer 5281 ? - proszę bardzo, wywołujemy funkcję i mamy odpowiedź. W tej funkcji kod wydobywania informacji zajmie oczywiście jakieś komórki pamięci, ale przecież ta funkcja będzie tylko jedna, a danych - tysiące. Zysk jest wtedy oczywisty.

13 Zagnieżdżona definicja klasy

Do pisania tego rozdziału przystępuję bez przekonania. Mam bowiem opisać rzecz, która moim zdaniem wydaje się mało przydatna. Z drugiej jednak strony trzeba o tym powiedzieć, choćby dlatego, że nastąpiła w tym zagadnieniu zmiana w stosunku do wersji **języka** wcześniejszych niż C++ 2.0

Także tu jest wyraźna różnica w stosunku do analogicznej sytuacji w klasycznym języku C. (W analogii dla struktur w C).

Mówić tu będziemy o rzeczy bardzo specyficznej, której znajomość nie jest konieczna do zrozumienia dalszych rozdziałów tej książki. Dlatego proponuję rozdział ten przy pierwszym czytaniu tej książki opuścić i przeskoczyć od razu do rozdziału o konstruktorach.

Jeżeli wewnątrz definicji klasy `A` zamieścisz definicję innej klasy `W` to mówimy, że definicja klasy `W` jest zagnieżdżona w definicji klasy `A`.

```
class A {                                   // klasa zewnętrzna
     ...                                    //
     class W {              // klasa wewnętrzna
          ...               //
     } ;                    //
                                            //
     ...                                    //
                                            //
};                                          //
```

Podkreślam jednak: to tylko definicja jest zagnieżdżona, a nie żaden obiekt.

Taka definicja klasy wewnętrznej jest lokalna dla klasy zewnętrznej. Czyli ma ona zakres ważności ograniczony do wnętrza klasy w której tkwi.

Tu jest właśnie różnica:

Dawniej, w starszych wersjach języka C++ klasa taka miała zakres ważności taki sam, jak ta klasa zewnętrzna. Zatem tak, jakby nie tkwiła w środku, lecz była na zewnątrz – po prostu obok. Zagnieżdżenie wówczas było więc tylko jakby kwestią notacji. Nic więcej.

Teraz – definicja klasy wewnętrznej znana jest tylko w obrębie klasy wewnętrznej. Zagnieżdżenie jest prawdziwe.

Co to oznacza ?

Znaczy to, że tylko w obrębie wnętrza klasy `A` można kreować obiekty klasy `W`.

Jeśli w obrębie klasy powstanie obiekt klasy zagnieżdżonej, to obowiązują mimo wszystko zwykłe zasady dostępu. Żaden z obiektów jednej z tych klas nie ma żadnych przywilejów w stosunku do obiektu tej drugiej klasy. Składniki prywatne obu klas są dla nich nawzajem niedostępne.

Z faktu, że definicja klasy wewnętrznej `W` jest na zewnątrz klasy `A` zupełnie nieznana – wynika, iż nie może być wskaźnika, który pokazałby z zewnątrz na obiekt klasy `W`.

Jak wygląda definicja klasy zagnieżdżonej ?

Tak, jak definicja zwykłej klasy z tym, że znajduje się w środku innej klasy. To już mówiliśmy.

A teraz zagadka. Gdzie wobec tego są definicje funkcji składowych tej klasy zagnieżdżonej? Wiemy, że normalnie można to robić na dwa sposoby: albo we wnętrzu definicji klasy – są one wtedy typu `inline`, albo na zewnątrz definicji klasy.

Jak to jest w wypadku definicji klasy, która jest zagnieżdżona w innej? Gdzie konkretnie są definicje tych funkcji składowych? Na zewnątrz obu klas, czy też tylko na zewnątrz tej zagnieżdżonej `W`, ale w wewnątrz klasy `A`?

Odpowiedź jest taka: Na zewnątrz obu.

Jeśli definicje (nie: deklaracje) funkcji składowych klasy zagnieżdżonej nie mają być bezpośrednio w definicji klasy (jak te `inline`), to wówczas należy je umieścić jak zwykłe funkcje w zasięgu globalnym. Kompilator i tak rozpozna dla której klasy są przeznaczone. Nie można próbować definiować funkcji zaraz za końcem zagnieżdżonej klasy `W`, czyli jeszcze wewnątrz klasy `A`.

To złamałoby bowiem przyjętą w języku C++ (w klasycznym C także) zasadę, że:

> Funkcje nie mogą być definiowane lokalnie wewnątrz innych funkcji i bloków.

Oto przykład:

```
class zewn {            // ========================== ❶
     int a ;
     class wewn {            // *****************      ❷
               float x ;                        //***
     public :                                   //***
          void funjeden() ;                     //***   ❸
```

```
        void fundwa() ;                    //***          ❹
    } ;                          // ****************

    // ponizej jest błąd                                   ❺
    void wewn::funjeden()
    {
    }
} ;        //=========================================
/*****************************************************/
void zewn::wewn::fundwa()                                  ❻
{    // ...
}
/*****************************************************/
```

Komentarz

❶ Klasa zewnętrzna o nazwie `zewn`.

❷ Definicja klasy `wewn`, która jest zagnieżdżona w definicji klasy ❶.

❸ i ❹ deklaracje dwóch funkcji składowych klasy `wewn`.

❺ Próba definiowania funkcji ❸ jeszcze wewnątrz klasy `zewn` to błąd.

❻ Poprawne miejsce definicji funkcji składowej ❹. Zauważ, że teraz nazwa funkcji zawiera niemal ścieżkę do funkcji – dwa operatory zakresu

```
void zewn::wewn::fundwa()
```

czytamy to – funkcja `fundwa` jest funkcją składową klasy `wewn`, która ma definicję zagnieżdżoną w zakresie ważności klasy `zewn`.

Klasa o definicji zagnieżdżonej ma zakres ważności ograniczony do klasy, w której się znajduje

Z tego wynika, że:

Klasa wewnętrzna może używać zdefiniowanych w klasie zewnętrznej

- – nazw typów (instrukcja `typedef`),
- – typów wyliczeniowych (`enum`),
- – publicznych składników statycznych – bo do takich nie musi się docierać podając nazwę konkretnego obiektu klasy zewnętrznej.

Do innych składników klasy zewnętrznej w klasie wewnętrznej musimy się odnosić tak, jak z każdej innej obcej klasy:

obiekt.składnik
referencja.składnik
wskaźnik->składnik

Jeśli klasa wewnętrzna deklaruje swą przyjaźń z jakąś dowolną obcą funkcją, to wcale nie oznacza, że klasa zewnętrzna też się z tą funkcją automatycznie przyjaźni.

Jeśli w klasie wewnętrznej jest jakaś deklaracja `typedef`, to klasę zewnętrzną to nie interesuje. Ta deklaracja nie rozciąga się na nią – ma ona tylko zakres klasy wewnętrznej.

Skoro tyle ograniczeń to jaka korzyść z zagnieżdżenia definicji klasy?

Moim zdaniem niewielka.

> Polega ona chyba tylko na tym, że zagnieżdżenie definicji klasy sprawia, iż jest ona nieznana gdzie indziej. Nie można gdzie indziej kreować obiektów tej klasy.

Ukrywanie czegoś przed sobą samym wydaje się mało przydatne, jednak zdarza się, że piszemy klasę dla innych użytkowników. Dla nich to, jak w środku zrobiona jest klasa – nie jest interesujące. Oni chcą tylko tej klasy używać. Nic ich nie obchodzi czy my po drodze nie definiujemy sobie jakiejś „roboczej" klasy, której obiekty ułatwią nam osiągnięcie jednego z celów. Zagnieżdżona klasa sprawia, że ta „robocza" klasa jest zupełnie niewidoczna dla świata zewnętrznego. Nie ma więc ryzyka, że nastąpi kolizja nazw, gdy użytkownik przypadkowo użyje sobie identycznej nazwy jak nasza „robocza" klasa.

To tyle. Jeśli masz jeszcze jakieś pytania, to najlepiej od razu dodam, że jeszcze ani raz w żadnym poważnym programie nie użyłem zagnieżdżenia definicji klas.

13.1 Lokalna definicja klasy

Tutaj mówić będziemy o zagnieżdżeniu definicji klasy, ale nie w innej klasie, tylko w jakiejś funkcji. Jeśli definicję klasy umieścimy w funkcji, to ma ona zakres ważności lokalny, ograniczony do bloku tej funkcji.

Podkreślam: wewnątrz funkcji jest definicja klasy, a nie konkretny egzemplarz jej obiektu.

Nazwa takiej klasy widziana jest tylko w zakresie, w którym jest zdefiniowana, czyli poza zakresem tej funkcji nie da się kreować obiektów tej klasy, ani nawet na nich działać.

Funkcje składowe takiej klasy

muszą być zdefiniowane wewnątrz ciała klasy (będą więc `inline`).

Nie można definicji tych funkcji umieścić poza ciałem klasy bo:

- – zaraz za definicją klasy, (czyli jeszcze w funkcji, w której jest ona lokalną) nie można, bo łamałoby to wspomnianą już w poprzednim paragrafie zasadę C++, że definicje funkcji nie mogą być zagnieżdżane w innych funkcjach,
- – definicji funkcji składowych tej lokalnej klasy nie można umieścić w zakresie globalnym, bo tam nazwa tej klasy jest zupełnie nieznana. Z tych ograniczeń wynika następująca zasada praktyczna: Skoro funkcje składowe będą `inline` – to powinny być krótkie.

Klasa lokalna przydaje się wtedy, gdy mamy do czynienia z klasą bardzo prostą i gdy jej użycie ogranicza się tylko do wnętrza tej jednej jedynej funkcji w programie.

Lokalna klasa ma zakres ważności wnętrza funkcji, w której się znajduje, więc może używać zdefiniowanych w niej nazw typów (`typedef`), typów wyliczeniowych (`enum`), zdefiniowanych w niej zmiennych statycznych, a także nazw zadeklarowanych w niej jako `extern`.

Strasznie to zawiłe, co? Nie przejmuj się, łatwo to zapamiętać tak:

W klasie tej można używać tego wszystkiego, co już istnieje w czasie kompilacji oraz linkowania.

- Definicje typów wyliczeniowych (`enum`) wtedy już istnieją? Tak, są przecież napisane czarno na białym.
- Instrukcje `typedef`? – także – kompilator je poznał.
- Nazwy zdeklarowane jako typu `extern`? Także – kompilator się z nimi zapoznał i w czasie linkowania już będzie dokładnie wiadomo, które komórki w pamięci one zajmą.

Skoro tak, to czego nie ma w tym zestawie ?

Zmiennych (obiektów) automatycznych, które najczęściej definiujemy sobie w funkcjach. One będą bowiem leżały na stosie, więc ich adres jest na etapie kompilacji i linkowania jest nawet w przybliżeniu nieznany. Spójrz na stos książek na Twoim biurku – jego wygląd zależy nie tylko od tego, co robisz teraz, ale także od tego, co robiłeś wczoraj i przedwczoraj.

Kompilator nie może więc wygenerować dla naszej klasy lokalnej kodu, który sprawi, że zmienną z tej komórki stosu doda się do tamtej, a rezultat złoży się w jeszcze innej. Zatem:

> Klasa lokalna w funkcji – nie może używać jej zmiennych automatycznych

Ciekawostki o zasłanianiu

Jeszcze jedna ciekawa rzecz: Jeśli mamy zmienną globalną o nazwie `xyz`, a w funkcji zdefiniujemy sobie zmienną automatyczną o takiej samej nazwie `xyz`, to zwykle nazwa zmiennej lokalnej automatycznej zasłania nazwę globalną.

Tutaj jednak, w wypadku klasy lokalnej, znowu jest inaczej. Gdy kompilator pracuje nad definicją lokalnej klasy – nie da się nawet w przybliżeniu określić wyglądu stosu – zatem kompilator daje to, co może: obiekt globalny. On nie jest dla klasy lokalnej zasłonięty obiektem automatycznym. Jakiekolwiek odniesienie się od obiektu `xyz` będzie odniesieniem się do **globalnego** obiektu `xyz`.

Próba odniesienia się do jakiejś nazwy, która jest tylko lokalna, uznana zostanie za błąd w czasie kompilacji.

I jeszcze jedno: lokalna klasa nie może mieć swoich składników statycznych.

Pokażmy to wszystko na przykładzie

```
#include <iostream.h>
int xyz = 10 ;                          // zmienna globalna          ❶
void zwykla() ;
/*****************************************************/
main()
{
     zwykla() ;
     // lokalik BBB ;                                                ❷
}
/*****************************************************/
void zwykla()
{
int xyz = 15 ;                                                  // ❸
int lokal_autom ;                                               // ❹
static float lokal_stat = 77 ;                                  // ❺

     class lokalik {
     public :
          // static int sss ;      // błąd - klasa nie może       ❻
                                   // mieć składników statycznych
          void lok_funskl()
          {
               cout << "to jest inline"
                    << "xyz= " << xyz << endl                   // ❼
                    // << lokal_autom // błąd !                 // ❽
                         << lokal_stat  // o.k. !               // ❾
                    << endl ;
          }
     } ;

     cout << "jestem w zwyklej funkcji \n" ;
     lokalik A ;                                                // ❿
     A.lok_funskl() ;                                          // ❶❶

}
```

Spróbuj sam to skompilować

Wydruku z ekranu nie zamieszczam, gdyż większość z dostępnych mi kompilatorów zachowuje się tutaj po swojemu i zwykle nie respektuje omawianych zasad.

Mimo to skomentujmy to, co widzimy w programie

❶ Definicja obiektu globalnego `xyz`.

❸ Wewnątrz funkcji `zwykla` definiujemy obiekt automatyczny o tej samej nazwie.

❹ Także definicja obiektu automatycznego, ale tym razem nazwa jest nigdzie indziej nie używana.

❺ Definicja obiektu lokalnego, ale statycznego.

❻ Definicja klasy. Ta definicja jest lokalna dla funkcji `zwykla`. Próba zdefiniowania składnika statycznego w lokalnej klasie jest błędem. Trzeba ująć to w znaki komentarza, by kompilacja się powiodła.

❼ Odniesienie się do obiektu o nazwie `xyz`, jest odniesieniem się do globalnego obiektu o tej nazwie. Można się o tym przekonać patrząc na to, co wypisane zostaje na ekranie - wartość 10 jest w obiekcie globalnym (natomiast w lokalnym jest 77).

❽ Próba odniesienia się do obiektu automatycznego o nazwie `lokal_autom` byłaby błędem. Globalnego obiektu o takiej nazwie nie ma, a do lokalnego automatycznego odnosić się nam nie wolno.

❾ Natomiast do lokalnego statycznego wolno.

❿ Tak definiuje się egzemplarz obiektu klasy `lokalik`. Wewnątrz funkcji `zwykla` nam to wolno. Nie wolno nam natomiast nigdzie poza tą funkcją – bo ta klasa jest tam nieznana. Dlatego błędem byłaby definicja ❷.

❶❶ Wywołanie funkcji składowej klasy `lokalik` dla obiektu tej klasy.

Na koniec jeszcze jedna oczywista chyba uwaga – klasa nie musi być lokalna z powodu faktu zamieszczenia jej definicji w bloku funkcji. Równie dobrze może być lokalna z powodu zagnieżdżenia jej w jakimkolwiek zwykłym lokalnym bloku oznaczonym w programie dwoma klamrami.

13.2 Lokalne nazwy typów

Instrukcja `typedef` definiująca nową nazwę dla jakiegoś istniejącego już typu może być umieszczona w zakresie ważności lokalnego bloku lub wewnątrz definicji klasy. Wówczas jej działanie ogranicza się tylko do zakresu tej klasy lub bloku. Poza tym zakresem ważności jest nieznana.

Natomiast jeśli taka instrukcja istniała już na zewnątrz lokalnego bloku, to wówczas wewnątrz bloku możemy skorzystać z efektu działania tej definicji.

```
void funkcja()
{
typedef int czas ;
     {                                  //--------- lokalny blok -------------
     typedef unsigned char byte ;

          czas bbb ;
          byte aaa ;
          // ... zwykłe instrukcje

     }                                  // ------koniec lokalnego bloku --

     czas ccc ;
     // byte ddd ; błąd ! - nieznane tutaj

     // ... zwykłe instrukcje
}
```

Widzimy, że jest tak, jakby to było definiowanie zwykłych zmiennych. Podobieństwo jest jeszcze większe: Otóż jeśli wewnątrz naszego lokalnego bloku instrukcją `typedef` zdefiniowalibyśmy nazwę typu `czas`, to ta definicja zasłoni w lokalnym bloku poprzednią definicję o tej nazwie.

Ten chwyt jednak można zastosować tylko wtedy, jeśli w lokalnym bloku ani raz jeszcze nie skorzystaliśmy z dotychczasowej definicji (tej zewnętrznej). Jeśli już skorzystaliśmy – to przepadło.

Podobnie jest z nazwą zewnętrzną: jeśli jest raz użyta wewnątrz definicji klasy, to nie można się już nagle rozmyślić i zastosować ją do instrukcji `typedef`.

Podobnie jak i w poprzednim paragrafie – zwracam uwagę, że i te sprawy różnie respektowane są przez różne kompilatory. Tutaj więc mówiliśmy o tym, jak być powinno, a nie o tym, jak to w istocie jest.

14 Konstruktory i Destruktory

Aby klasa, czyli typ definiowany przez użytkownika – przypominała swoim zachowaniem typy wbudowane, wymyślono trzy specjalne rodzaje funkcji składowych:

1) konstruktor i destruktor,

2) funkcje składowe przeładowujące operatory,

3) operatory konwersji.

Punktom 2) i 3) poświęcone są dalsze rozdziały. Tutaj zajmiemy się szczegółowo punktem pierwszym. Co prawda o konstruktorach i destruktorach napomknęliśmy w jednym z poprzednich rozdziałów, jednak teraz czas przyjrzeć im się bliżej.

14.1 Konstruktor

Konstruktor to specjalna funkcja składowa, która nazywa się tak samo jak klasa. W ciele tej funkcji (konstruktora) możemy zamieścić instrukcje nadające składnikom obiektu wartości początkowe. W trakcie definiowania obiektu, przydziela mu się miejsce w pamięci, a następnie uruchamiany jest dla niego konstruktor.

Zwróć uwagę, że:

> Konstruktor sam nie przydziela pamięci na obiekt. On może ją tylko zainicjalizować. Zatem używając analogii: nie jest to budowniczy obiektu – raczej dekorator wnętrz.

W samej treści konstruktora nie ma nic nadzwyczajnego: ot, taka sobie funkcja składowa. Najważniejszym aspektem konstruktora jest to, że jeśli klasa ma odpowiedni konstruktor, to jest on **automatycznie** uruchamiany przy definiowaniu każdego obiektu tej klasy.

Cechy konstruktora

Konstruktor może być przeładowywany. Jest to bardzo częsta praktyka, w definicjach klas widzi się zwykle kilka wersji konstruktora (różnią się oczywiście listą argumentów).

Konstruktor nie ma wyspecyfikowanego żadnego typu wartości zwracanej. Nie zwraca nic – nawet typu `void`! Jeśli więc w ciele konstruktora jest instrukcja `return`, to nie może przy niej stać żadna wartość. Tylko średnik.

Konstruktor może być wywoływany dla tworzenia obiektów z przydomkami `const` i `volatile`, ale sam nie może być funkcją typu `const` i `volatile`.

(Pamiętamy, że z innymi funkcjami składowymi jest tak, że na rzecz obiektów takiego typu mogą pracować tylko te, które obiecują, że także są odpowiednio: `const` lub `volatile`. Jak widać konstruktora to zastrzeżenie nie obowiązuje.)

Konstruktor nie może być także typu `static` – między innymi dlatego, że ma pracować na niestatycznych składnikach klasy. Jako `static` - nie miałby do tego prawa. (Musi mieć przecież wskaźnik `this`.)

Dla wtajemniczonych przypomnę, że konstruktor nie może być także typu `virtual`.

Nie można posłużyć się adresem konstruktora.

Jeśli obiekt ma być składnikiem unii, to jego klasa nie może mieć żadnego konstruktora. Łatwo to zrozumieć: jeśli jest konstruktor, to startuje on do pracy automatycznie. W wypadku gdyby w unii było kilka obiektów klas z konstruktorami, to przecież nie ma sensu, by wszystkie ruszyły do pracy zwalczając się nawzajem. Skoro zgody być nie może, to lepiej niech unia konstruktorów nie ma.

14.1.1 Przykład programu zawierającego klasę z konstruktorami

Ilustruje on niektóre z omówionych cech. Problem jest taki: mamy na ekranie narysować kilka przyrządów pokładowych. Wszystkie mają wygląd miernika ze wskazówką (albo wyświetlacza cyfrowego). Oczywiście od razu nasuwa się, że każdy taki miernik jest obiektem *klasy* przyrząd. Taka właśnie klasa zdefiniowana jest w naszym przykładzie.

Do operacji pisania w odpowiednich miejscach ekranu użyłem funkcji z biblioteki dla kompilatora Borland C++. Nie musisz rozumieć tych instrukcji. To, o czym mówię, jest wewnątrz funkcji składowej `narysuj()`. Ważne jest tylko, że funkcja ta pracując w tzw. trybie alfanumerycznym rysuje na ekranie coś, co przy odrobinie fantazji przypomina wyświetlacz cyfrowy.

Oczywiście byłoby o wiele ładniej, gdybym posłużył się tak zwanymi znakami semigraficznymi, które pozwalają na narysowanie ładniejszej ramki. Mogłem też przejść do trybu graficznego i narysować piękne instrumenty pokładowe jak w symulatorach lotu. Byłyby to jednak rzeczy zbyt związane z jakimś konkretnym typem komputera, kompilatora i określoną biblioteką graficzną. Nie o to chodziło mi w tym przykładzie. Ilustruje on tylko różne aspekty konstruktorów.

```
#include <string.h>                    // strcpy          ❶
#include <stdlib.h>                    // itoa
#include <conio.h>                     // gotoxy, cprintf
#include <dos.h>                       // sleep

class przyrzad {
     char nazwa[20] ;
     char jednostki[10] ;
     int pokazuje ;
     int x, y ;          // gdzie jest na ekranie ten przyrząd
     static int ile_nieznanych ;                        // ❷

public :

     // konstruktory
     przyrzad (int, int, char *, char *, int = 0) ; // ❸
     przyrzad (void) ;

     // zwykłe funkcje składowe
     void zmien (int w ) ;
     void narysuj (void);
} ;
//////////////////////////////////////////////////////////////
// definicje konstruktorów
//*********************************************************
przyrzad::przyrzad(int xx, int yy, char * nnn,
                              char * jedn, int w)
{
     strcpy(nazwa, nnn);
     strcpy(jednostki, jedn);
     pokazuje = w ;
     x = xx ;
     y = yy ;
     narysuj() ;                                        // ❹
}
//*********************************************************
przyrzad::przyrzad(void)                                // ❺
{
char tmp[20] ;

     ++ile_nieznanych ;        // jeszcze jeden nieznany wskaźnik

     strcpy(jednostki, " ");

     // wypełnienie tablicy tekstem - "Wskaznik nr n"
     strcpy(nazwa, "Wskaznik nr ");
     itoa(ile_nieznanych, tmp, 10) ;
     strcat(nazwa, tmp);

     // wymyślenie dla przyrządu jego pozycji na ekranie
     x = 45 ;
     y = 1+ (ile_nieznanych-1) * 4 ;                    // ❻

     pokazuje = 0 ;        // co ma on pokazywać

     // narysowanie go na ekranie
```

```
        narysuj();
}
//                    ******** dalsze funkcje składowe ******************************
//*****************************************************
void przyrzad::zmien(int w)
{
        pokazuje = w ;
        narysuj();
}
//*****************************************************
void przyrzad:: narysuj()
{
        // tej funkcji ładnie nie definiujemy. Zależy ona od
        // tego jaką mamy bibliotekę graficzną.
        // Markujemy wobec tego wyświetlaczem cyfrowym

        // przepraszam za poniższe instrukcje !                    //❼
        gotoxy(x, y);
        cprintf("  ______________________________");
        gotoxy(x, y+1);
        cprintf("I %-20s I" , nazwa);
        gotoxy(x, y+2);
        cprintf("I %5d %10s I" , pokazuje, jednostki);
        gotoxy(x, y+3);
        cprintf("I______________________________I");
}
//=====================================================
int przyrzad::ile_nieznanych ;                          //❽
//=====================================================
main()
{
        clrscr();                                       // ❾

        // definicje obiektów ---------------------
        przyrzad Pred(2, 1, "Predkosc", "wezlow", 110);// ❿
        przyrzad Vari(2, 6, "Wzoszenie", "stopy/sec");// ❶❶
        przyrzad A ;                                   // ❶❷
        przyrzad B ;
        volatile przyrzad C ;                          // ❶❸

        const przyrzad Udzw (2, 11, "Udzwig maksymalny",
                        "ton", 15000 );                // ❶❹

        // symulacja normalnego ciągłego wyświetlania ----
        for(int i = 0 ; i < 30 ; i++){                 // ❶❺
                Vari.zmien(i);
                Pred.zmien(110+i);
                A.zmien(-i);
                B.zmien(i % 4) ;
                // C.zmien(i % 3);       // obiekt volatile  ❶❻
                // Udzw.zmien(270);      // obiekt const     ❶❼

                delay(500) ;
        }
}
```

Kompilacja i wykonanie programu na komputerze IBM PC zaowocuje takim (w przybliżeniu!) wyglądem ekranu.

```
 ______________________
I Predkosc             I
I   139       wezlow   I
I______________________I

 ______________________
I Wznoszenie           I
I     29  stopy/sec    I
I______________________I

 ______________________
I Udzwig maksymalny    I
I 15000        ton     I
I______________________I
```

```
 ______________________
I Wskaznik nr 1        I
I   -29                I
I______________________I

 ______________________
I Wskaznik nr 2        I
I     1                I
I______________________I

 ______________________
I Wskaznik nr 3        I
I     0                I
I______________________I
```

Wskazania poszczególnych przyrządów pokładowych będą się zmieniać w czasie.

Przyjrzyjmy się samemu programowi

❶ Włączamy kilka plików nagłówkowych zawierających deklaracje odpowiednich funkcji bibliotecznych. W komentarzach umieszczone są nazwy funkcji, o które nam tu chodziło.

❷ Definicja klasy przyrząd. Jej składnikami-danymi są dwie tablice do przechowywania tekstów, następnie trzy dane typu `int` określające bieżące wskazania miernika i pozycję miernika na ekranie (`x` – to u nas numer kolumny, `y` – to numer rzędu). Widzimy też tutaj deklarację składnika statycznego tej klasy. Przypominam, że składnik statyczny klasy musi być zdefiniowany na zewnątrz. Jego definicję widzimy w miejscu ze znaczkiem ❸. Zauważ operator zakresu z nazwą klasy oraz to, że w tym miejscu nie ma już słowa `static`.

❸ Klasa ma kilka wersji konstruktorów - jest to, jak wiadomo, przeładowanie. Widzimy tu dwa konstruktory, ale ponieważ jeden z nich ma argument domniemany, więc to tak, jakbyśmy mieli trzy następujące konstruktory:

```
przyrzad (int, int, char *, char *, int) ;
przyrzad (int, int, char *, char *) ;
przyrzad (void) ;
```

Przypominam, że argument domniemany funkcji określa się w deklaracji wewnątrz klasy, a jeśli sama definicja takiej funkcji (tutaj: konstruktora) jest gdzieś na zewnątrz definicji klasy, to przy jej definicji już się o tym nie wspomina po raz drugi.

❹ Wewnątrz definicji konstruktora może nastąpić wywołanie innej funkcji składowej tej klasy.

❺ Oto definicja konstruktora

```
przyrzad::przyrzad(void) ;
```

wywoływanego, jak widać, bez żadnych argumentów. Taki konstruktor nazywa się także *konstruktorem domniemanym*. (Nie ma to nic wspólnego z argumentem domniemanym wspomnianym przed chwilą.)

W następnych linijkach widzimy jak w tym konstruktorze wpisuje się dane do określonych składników.

Zamiast opisu mówiącego co pokazuje wskaźnik, wpisuje się tam napis: "Wskaźnik nr ... " gdzie numer jest kolejnym numerem przyrządu. Nie numerujemy jednak wszystkich przyrządów tylko te, które kreujemy konstruktorem domniemanym. Informację o tym, ile już takich nienazwanych przyrządów powstało, przechowujemy w składniku statycznym tej klasy. Jak pamiętamy, ten składnik jest wspólny dla wszystkich obiektów tej klasy. Każde wykonanie tego właśnie konstruktora – powoduje zwiększenie owego składnika o 1 (zauważ inkrementację).

Ponieważ chcę liczbę nieznanych przyrządów zamienić na string będący jej wizerunkiem alfanumerycznym, to muszę się posłużyć taką funkcją biblioteczną: `itoa` – skrót od: integer **to** ASCII.

Sam jestem sobie winien. To wszystko można zrobić o wiele prościej, ale niestety rozdział na temat klas odpowiadających za łatwą i wygodną pracę z ekranem musi znaleźć się na samym końcu książki. Nie przerażaj się więc tym, że aby dokonać tu prostego wypisu na ekran drapię się prawą ręką za lewe ucho.

❻ Nawet bliżej nieokreślony przyrząd musi mieć na ekranie określone miejsce. Trzeba je jakoś wybrać i robimy to tutaj na przykład w taki sposób.

❼ Narysowanie przyrządu na ekranie jest rzeczą bardzo specyficzną dla określonej implementacji. Tutaj jest kilka instrukcji, których nie musisz wcale rozumieć. Rysują one bardzo prymitywne pudełko w miejscu ekranu określonym przez współrzędne `x` oraz `y`. Wewnątrz wpisywany jest tekst.

❾ W `main` rozpoczynając pracę programu wywołujemy funkcję biblioteczną `clrscr()` odpowiadającą za wyczyszczenie ekranu. (Skrót od: *clear screen*).

❿ Przystępujemy do definicji obiektów klasy `przyrzad`. Tutaj widzimy definicję z wywołaniem konstruktora z argumentami.

❶❶ Tutaj robimy to samo dla innego obiektu, ale ostatni argument jest pominięty. Ponieważ ten argument jest określony jako domniemany, więc kompilator mniema, że programiście chodzi o wartość = 0.

❶❷ Tutaj w definicji nie ma żadnej listy argumentów. Uruchamia to zatem konstruktor

```
przyrzad::przyrzad (void) ;
```

❶❸ Definicja obiektu z przydomkiem `volatile`. Jak pamiętamy, na rzecz takiego obiektu mogą być uruchomione tylko te funkcje składowe, które także są `volatile`. Nie dotyczy to jednak konstruktorów. Konstruktor zresztą nie może mieć przydomka `volatile`.

❶❹ Definicja obiektu z przydomkiem `const`. Wszystkie powyższe uwagi powtarzają się w wypadku przydomka `const`. Mimo, że konstruktor nie jest (bo być nie może) `const`, to jednak może pracować na rzecz takiego obiektu.

❶❺ Mamy tu pętlę, w której symuluje się zmiany wskazań poszczególnych przyrządów. Na dole pętli widzimy funkcję biblioteczną `delay` (ang.- zwłoka), która sprawia, że wskazania na przyrządach aktualizuje się co około 500 milisekund.

❶❻ Błędem byłoby wywołanie funkcji `zmien` na rzecz obiektu z przydomkiem `volatile`. To dlatego, iż funkcja ta, nie mając sama przydomka `volatile`, nie gwarantuje tym samym, że z naszym obiektem będzie się obchodziła ze specjalną troską (należną obiektom tego typu).

❶❼ Błędem byłoby wywołanie funkcji `zmien` na rzecz obiektu typu `const`, ponieważ funkcja `zmien`, nie mając przydomka `const`, nie gwarantuje, że nie będzie zmieniać składników obiektu. (My zresztą wiemy, że ona właśnie chciałaby je zmieniać !).

W `main` widzieliśmy między innymi takie definicje obiektów:

```
przyrzad Vari(2, 6, "Wzoszenie", "stopy/sec");
przyrzad A ;
```

Mam nadzieję, że przyzwyczaiłeś się już do tego zwięzłego zapisu. Jeśli nie, to przypomnę, że te dwie linijki można zapisać również tak:

```
przyrzad Vari = przyrzad(2,6, "Wznoszenie", "stopy/sec");
przyrzad A    = przyrzad() ;
```

Pułapka

Uwaga: Początkujący programiści często się mylą i tę ostatnią definicję piszą tak:

```
przyrzad A();
```

Jest to błąd. Zamiast definicji obiektu mamy tu deklarację... czego ? Przeczytajmy tę deklarację:

> `A` – jest funkcją wywoływaną bez żadnych argumentów, a zwracającą w rezultacie obiekt klasy `przyrzad`.

Krótko mówiąc zupełnie nie to, o co nam chodziło. Przypomnijmy dwie formy poprawnej definicji.

```
przyrzad A ;
przyrzad A = przyrzad() ;
```

Pułapka ta grozi nam wtedy, gdy chcemy użyć konstruktora bez żadnych argumentów. Jeśli konstruktor ma argumenty to oczywiście zapis

```
przyrzad A(2, 6, "Wzoszenie", "stopy/sec");
```

jest poprawny. Nie ma tu mowy by kompilator pomyślał, że deklarujemy tu jakąś funkcję. W nawiasie są przecież nie *typy*, a konkretne *wartości*. Gdyby nic nie było (jak przedtem) – to wtedy zaczyna się problem.

14.2 Kiedy i jak wywoływany jest konstruktor

Podobnie jak w wypadku obiektów typów wbudowanych, tak i w wypadku obiektów typów zdefiniowanych przez użytkownika, możemy mieć kilka rodzajów obiektów – zależnie od tego, jak i gdzie je definiujemy. Przyjrzyjmy się jak w takich różnych sytuacjach pracuje konstruktor.

14.2.1 Konstruowanie obiektów lokalnych

Obiekty lokalne automatyczne

(czyli tworzone na stosie w trakcie wykonywania programu) – powstają w momencie, gdy program napotyka ich definicję, a przestają istnieć, gdy program wychodzi poza blok, w którym zostały powołane do życia.

```
...
    {                    <------- otwarcie lokalnego bloku

        przyrzad M ;             // definicja obiektu M
        ...
    }                    <------- zamknięcie bloku, obiekt M jest likwidowany
... tu obiektu M już nie ma
```

Konstruktor takiego obiektu uruchamiany jest w momencie, gdy program napotyka definicję tego obiektu.

Tak zdefiniowany obiekt jest obiektem lokalnym automatycznym.

Obiekty lokalne statyczne

Jeśli przy definicji stałoby słowo `static`, to obiekt byłby **obiektem lokalnym statycznym**. Znaczy to, że istniałby od samego początku programu, aż do momentu zakończenia programu, ale dostępny byłby tylko w zakresie ważności tego bloku. (Wszystko jest tak samo, jakby to było w wypadku statycznego obiektu typu: `int`. Różnica jest tylko ta, że obiekt typu `int` nie ma konstruktora).

Zagadka:

> *skoro nasz obiekt statyczny klasy `przyrzad` istnieje przez cały czas wykonywania programu, to kiedy startuje do pracy jego konstruktor ?*

Zaskoczę Cię! Konstruktor ruszy do pracy jeszcze przed rozpoczęciem wykonywania `main`. Ponieważ to Ty sam piszesz konstruktor, więc możesz w nim wykonać jakąś akcję jeszcze przed tym, jak `main` zacznie się w ogóle wykonywać.

14.2.2 Konstruowanie obiektów globalnych

Jeśli jakieś obiekty klasy `przyrzad` zdefiniowane będą poza wszystkimi funkcjami, to będą obiektami globalnymi.

```
przyrzad GGG ;
main()
{
```

```
        // ...
}
```

Zakres ważności takiego obiektu to plik. Jeśli z innych plików chcemy się odnieść do takiego obiektu, to jego nazwa musi być w tych plikach zadeklarowana.

Czas życia: cały czas wykonywania programu. Konstruktor rusza do pracy jeszcze przed rozpoczęciem wykonywania funkcji `main`.

14.2.3 Konstrukcja obiektów tworzonych operatorem new

Obiekt może zostać zdefiniowany (powołany do życia) przez użycie operatora `new`. Jest to bardzo pożyteczna rzecz, bo umożliwia tworzenie w trakcie wykonywania programu takiej liczby obiektów, o której się nam nawet nie śniło w momencie, gdy program pisaliśmy.

Oto przykład takiej definicji – dokonanej na przykład w środku jakiejś funkcji:

```
void f()
{
    przyrzad *nitka ;          // najpierw musimy mieć wskaźnik
    nitka = new przyrzad(1,1, "Waga", "kg");

    // ...
}
```

Najpierw definiujemy sobie wskaźnik mogący pokazywać na jakiś obiekt klasy `przyrzad`. Wskaźnik ten nazywamy `nitka`. Następnie za pomocą operatora `new` kreujemy obiekt klasy `przyrzad` - dzieje się to w zapasie dostępnej pamięci. Rezerwuje się tam dla niego pamięć i natychmiast rusza do pracy konstruktor wpisując tam między innymi słowa "Waga" i "kg". Kiedy obiekt jest gotowy wskaźnikowi `nitka` przekazuje się jego adres. Od tej pory możemy się tym obiektem posługiwać. Nie ma on co prawda nazwy, ale ma wskaźnik, który na niego pokazuje. To wystarcza.

Czas życia obiektu: od momentu kiedy program wykonał linijkę z tą instrukcją zawierającą operator `new` – aż do momentu, gdy sami nie zlikwidujemy obiektu operatorem `delete`.

```
delete nitka ;
```

Zakres ważności: Jeśli tylko jest w danym momencie dostępny choć jeden wskaźnik, który pokazuje na ten obiekt, to obiekt jest dostępny.

Jest tu jednak niebezpieczeństwo:

> Jeśli przez nieuwagę stracimy wskaźnik, który pokazuje na ten obiekt (zmienimy, przestawimy go na coś innego, albo po prostu przestanie on istnieć) wówczas stracimy wszelki kontakt z tak utworzonym obiektem. Obiekt będzie istniał nadal, ale już nie odszukamy go.

Przypomnij sobie scenkę z chłopczykiem kupującym balonik w parku. (str. 188) Wszystko to obowiązuje także i w stosunku do obiektów klas zdefiniowanych przez użytkownika.

Co jest nowe – to oczywiście zachowanie konstruktora. Jeśli operator `new` nie potrafi wykreować obiektu (bo na przykład nie ma już więcej miejsca w pamięci)

wówczas konstruktor tego obiektu nie jest uruchamiany. To chyba zrozumiałe: jeśli nie dostaliśmy przydziału na mieszkanie, to nie wzywamy dekoratora wnętrz.

14.2.4 Jawne wywołanie konstruktora

Obiekt może być też stworzony przez jawne wywołanie konstruktora. W efekcie otrzymujemy obiekt, który nie ma nazwy, a czas jego życia ogranicza się do wyrażenia, w którym go użyto.
Robi się to według składni

nazwa_klasy(argumenty)

Zauważ, że wywołujemy konstruktor – czyli funkcję składową, a nie stosujemy notacji

obiekt.funkcja_składowa(argumenty)

Konstruktor jest co prawda funkcją składową, ale specjalną. Nie jest wywoływany na rzecz jakiegoś obiektu, bo ten obiekt jeszcze nie istnieje. Zadaniem konstruktora jest go utworzyć. Stąd w wywołaniu nie ma jeszcze zapisu z kropką.

Jeśli takie zastosowanie nie jest jasne, to posłużmy się przykładem. Wyobraź sobie, że mamy jakąś prostą klasę `kl` oraz mamy jakąś funkcję, taką zwykła, nie należącą do żadnej klasy. Jest to jednak funkcja, której argumentem jest jakiś obiekt klasy `kl`.

```
#include <iostream.h>

class kl {
public :
     int       a ;
     float     b ;
     char      c ;
     // ----------------------------konstruktor
     kl(int i, float x, char z )
     { a = i ; b = x ; c = z ;}
} ;
/////////////////////////////////////////////////////////
void wypis(kl ) ;
main()
{
     kl obiektA(1, 3.14, 'x'),
          obiektB(2, 1.41, 's') ;

     wypis(obiektA);
     wypis(obiektB);

     wypis( kl(3, 7.77, 'l') ) ;                    // ❶
}
//*******************************************************
void wypis(kl sk)
{
     cout << " a= " << sk.a << " b= " << sk.b
```

```
                << " c= " << sk.c << endl ;
}
```

Spójrzmy na ekran

```
a= 1 b= 3.14 c= x
a= 2 b= 1.41 c= s
a= 3 b= 7.77 c= l
```

Komentarz

Zauważ w `main` dwa wywołania funkcji `wypis` – zrobione w zwykły sposób. Wywołujemy jednak tę funkcję także w inny sposób ❶. Widzimy, że argumentem jest tu wyrażenie będące jawnym wywołaniem konstruktora. Na użytek tego właśnie wyrażenia zostaje chwilowo wytworzony nienazwany obiekt klasy `kl` i wysłany do funkcji jako argument. Obiekt istnieje tylko w tej linijce. Po przejściu do następnej linijki już go nie ma.

Tak naprawdę, to z tym sposobem tworzenia obiektów (metodą jawnego wywołania konstruktora) – już się spotkaliśmy. Porównaj dwie wersje tej samej definicji obiektu P:

```
przyrzad P(5, 6, "Predkosc", "km/h" ) ;
przyrzad P = przyrzad (5, 6, "Predkosc", "km/h" ) ;
```

W drugiej wersji, za znakiem '=' wywoływany jest właśnie jawnie konstruktor. Tworzy się nienazwany obiekt klasy `przyrzad`. Po lewej stronie znaku przypisania jest utworzenie drugiego obiektu klasy przyrząd. Ten obiekt nosi nazwę `P`. Chwilowo więc istnieją te dwa obiekty. Znak przypisania oznacza, że obiekt P ma mieć dokładnie tę samą treść (składników danych) co obiekt nienazwany. Po wykonaniu tego kopiowania †) obiekt P ma także wpisane słowa "Predkosc" czy "km/h" itd. Program po skończeniu pracy nad tą linijką przechodzi do następnej linijki i w tym momencie likwidowany jest obiekt nienazwany. W rezultacie pozostaje tylko obiekt P o wyżej wspomnianej zawartości.

Rezultat jest więc taki sam, jak w wypadku pierwszej wersji definicji.

14.2.5 Dalsze sytuacje, gdy pracuje konstruktor

Wymienimy je tylko w punktach, bo dokładniej zajmiemy się nimi w stosownym czasie.

- Konstruktor wywoływany przy tworzeniu obiektów chwilowych (tej klasy).
- Konstruktor jest wywoływany także jeśli powstaje obiekt jakiejś klasy, który zawiera w sobie obiekt innej klasy. Urucha-

†) O tym jak dokonuje się kopiowania będziemy mówić w paragrafie o konstruktorze kopiującym.

miany jest wtedy konstruktor tego składnika.
(O tym za chwilę).

Dla wtajemniczonych:

- Konstruktor klasy podstawowej wywoływany jest przy kreacji obiektu klasy pochodnej.

14.3 Destruktor

O destruktorze już napomknęliśmy w rozdziale o klasach (str. 291). Tutaj zajmiemy się nim jeszcze raz – bliżej.

> Destruktorem klasy `K` jest jej funkcja składowa o nazwie `~K` (wężyk i nazwa klasy). Funkcja ta jest wywoływana automatycznie zawsze, gdy obiekt jest likwidowany.

Funkcja jak funkcja. Najważniejsze jest tu właśnie to *automatyczne* uruchamianie. Co do treści destruktora – to już zależy od nas samych.

Klasa nie musi mieć obowiązkowo destruktora. Destruktor nie likwiduje obiektu, ani nie zwalnia obszaru pamięci, który obiekt zajmował. Destruktor przydaje się wtedy, gdy przed zniszczeniem obiektu trzeba jeszcze dokonać jakichś działań. Po prostu trzeba posprzątać.

- Jeśli na przykład obiekt reprezentował okienko na ekranie, to możemy chcieć, by w momencie likwidacji tego obiektu okienko zostało zamknięte, a ekran wyglądał jak dawniej.
- Destruktor jest potrzebny, gdy konstruktor danej klasy dokonał na swój użytek rezerwacji dodatkowej pamięci (operatorem `new`) - na przykład zarezerwował sobie miejsce na dużą tablicę. Wtedy w destruktorze umieszcza się instrukcję `delete` zwalniającą ten już nie potrzebny obszar pamięci.
- Destruktor może się też przydać, gdy liczymy obiekty danej klasy. W konstruktorze podwyższamy taki licznik o jeden, a w destruktorze zmniejszamy o jeden. Coś na kształt dokładnej statystyki urodzin i zgonów.
- Skoro już ta analogia – to destruktor może się przydać po to, by obiekt mógł spisać na dysku (lub ekranie) swój testament. Poważnie! Obiekt ma być zlikwidowany, więc pisze na dysku w specjalnym pliku: „Byłem obiektem klasy o nazwie Boeing 747 o numerze identyfikacyjnym .... Zostałem zlikwidowany (data) po dokonaniu `n` napraw, przeleceniu `m` tysięcy kilometrów.." i tak dalej.

Destruktor jako funkcja nie może zwracać żadnej wartości (nawet typu `void`). Destruktor nie jest wywoływany z żadnymi argumentami. W związku z tym nie może być także przeładowany.

Destruktor jest automatycznie wywoływany, gdy obiekt automatyczny lub chwilowy wychodzi ze swojego zakresu ważności.

Jeśli obiekt lokalny jest statyczny, to mimo, że kończy się jego zakres ważności – nie jest likwidowany – więc także nie uruchamia się jego destruktora.

Likwidacja następuje dopiero przy zakończeniu programu i wtedy też rusza do pracy jego destruktor.

Do obiektu, który kreowaliśmy operatorem `new`, destruktor wywoływany jest, gdy zastosujemy operator `delete` wobec wskaźnika pokazującego na ten obiekt. Jeśli wskaźnik taki ma wartość NULL, to destruktor (mimo życzenia) nie jest uruchamiany.

Jeśli kończy się zakres ważności referencji (przezwiska) obiektu — destruktor nie jest wywoływany.

> *Bo: jeśli na Tomka przez pewien czas mówiono „łysy", to z faktu, że przezwisko zapomniano – (przezwisko umarło) – nie wynika, że umarł sam Tomek.*

Analogicznie: destruktor nie jest automatycznie wywoływany, gdy wskaźnik do jakiegoś obiektu wychodzi ze swojego zakresu. To, że wskaźnik przestaje istnieć, nie oznacza, że obiekt (na który do tej pory pokazywał) również ma przestać istnieć.

Obie zasady wydają się oczywiste.

Podobnie jak konstruktor – destruktor także może wywołać jakąś funkcję składową swojej klasy.

Niemożliwe jest pobranie adresu destruktora.

Obiekt klasy mającej destruktor nie może być składnikiem unii.

> *Powody wydają mi się te same, jak w wypadku konstruktora: gdyby w unii były dwa obiekty klas z destruktorami – to który destruktor przy likwidacji należałoby uruchomić. Dwa – to bez sensu. Jeden? A który? Według przebywającego właśnie w unii obiektu? Zapominasz, że unia nie wie jaki obiekt w danej chwili mieści. (To my mamy pamiętać czy jest tam guma do żucia czy zdechła żaba.)*

Destruktor nie może być ani `const` ani `volatile`, ale może pracować na obiektach swej klasy z takim przydomkiem. Wyjątkowo, bo oczywiście zwykła funkcja składowa nie mogłaby. Musiałaby sama być także `const` lub `volatile`.

W paragrafie „Destruktor – pierwsza wzmianka" widzieliśmy już przykład programu zawierającego prosty destruktor. Obserwowaliśmy jak zostaje automatycznie uruchamiany.

Jawne wywołanie destruktora

Destruktor można wywołać jawnie. Należy wówczas podać całą jego nazwę. To dlatego, żeby nie było nieporozumienia w interpretacji wężyka – który jest, jak pamiętamy, także jednoargumentowym operatorem bitowej negacji (patrz str. 63).

Zapamiętaj sobie taką zasadę:

> Jawne wywołanie destruktora nie może się zacząć od '~' (wężyka).
> Wcześniej musi być albo obiekt, na rzecz którego jest wywoływany i kropka '.', albo wskaźnik do obiektu i '->'

```
obiekt.~klasa();
wskaznik->~klasa() ;
```

Zapytasz może: A co zrobić jeśli destruktor uruchamiamy z wnętrza klasy? Przecież wówczas odnosząc się do funkcji składowych nie trzeba specyfikować, o który obiekt chodzi. Wiadomo, że destruktor wywołujemy wówczas na rzecz tego właśnie obiektu. Zatem nic nie stoi przed nazwą uruchamianej funkcji składowej (tutaj: destruktora). Czy zatem w tym wypadku wolno wywołanie destruktora zapisać tak:

```
~klasa() ;
```

Nie. Wówczas musimy napisać tam to, co tam naprawdę jest, ale jest niewidoczne: wskaźnik `this`

```
this->~klasa() ;
```

Przypominam, że jawne wywołanie destruktora nie likwiduje obiektu. To tylko jakby wezwanie sprzątaczki. Co ona zrobi, to już nasza sprawa. Po jej wyjściu sam obiekt nadal istnieje. Może trochę posprzątany, zmodyfikowany, ale istnieje.

Wywołanie nieistniejącego destruktora

Może się zdarzyć, że klasa, którą się posługujemy, nie ma destruktora. Jeśli mimo to jawnie go wywołamy, to wywołanie takie zostanie zignorowanie.

Co ciekawsze można również wywołać destruktor dla typu wbudowanego. Także i takie wywołanie jest dopuszczalne, ale ignorowane.

Oto wywołanie destruktora na rzecz obiektu `int`.

```
int a ;
a.~int() ;
```

(Uwaga: Z niewiadomych mi powodów powyższy zapis przez Borland C++ jest uznawany jako błędny).

Destruktor ma jeszcze inne ciekawe właściwości – poznamy je w następnych rozdziałach.

14.4 Konstruktor domniemany

Konstruktor domniemany to taki konstruktor, który można wywołać bez żadnego argumentu.

Oto przykład klasy z konstruktorem domniemanym

```
class boss {
     // ...
public :
     // konstruktory
     boss(int);
     boss(void);                    // <-- domniemany
     boss(float*) ;
     // ...
} ;
```

Zauważ, że nie mówimy

> „konstruktor bez argumentów"

tylko

> „konstruktor, który można wywołać bez żadnych argumentów."

W świetle tej definicji konstruktorem, który można wywołać bez żadnych argumentów jest konstruktor ze wszystkimi argumentami domniemanymi†)

```
class don {
     // ...
public:
     // konstruktory
     don(int);
     don(float) ;
     don(char *s = NULL, int a =4, float pp = 6.66);
} ;
```

Konstruktorem domniemanym jest ten ostatni, najdłuższy. To dlatego, że można go wywołać bez żadnych argumentów.

Klasa może mieć oczywiście tylko jeden konstruktor domniemany. Wynika to z istoty przeładowania funkcji.

Jeśli klasa nie ma w ogóle **żadnego** konstruktora, wówczas sam kompilator wygeneruje sobie dla tej klasy konstruktor domniemany. Będzie on `public`. Podkreślam: to automatyczne generowanie konstruktora domniemanego zajdzie tylko wtedy, gdy klasa nie ma ani jednego konstruktora.

14.5 Lista inicjalizacyjna konstruktora

Pamiętasz zapewne takie definicje

```
const int stal = 44 ;
```

Dzięki temu w programie powstaje obiekt stały o nazwie `stal` i wartości 44. Pamiętasz też zapewne zasadę, że obiekty stałe mogą być tylko inicjalizowane, nie wolno do nich nic przypisywać. (Przypominam, że inicjalizacja to nadanie wartości w momencie narodzin). Zatem gdybyśmy w powyższej instrukcji nie nadali obiektowi `stal` wartości 44, to potem już nie można by tego zrobić.

To były stare sprawy. A teraz wyobraź sobie taką klasę:

```
class abc {
     const int stal ;
     // ...
} ;
```

†) We wczesnych wersjach języka C++ konstruktor domniemany nie mógł rzeczywiście mieć żadnych, nawet domniemanych argumentów.

Widzimy w niej składnik `stal` z przydomkiem `const`. Pytanie: Jak nadać temu składnikowi wartość początkową? Jak wiemy w ciele klasy nie wolno nam napisać inicjalizacji w taki sposób

```
class abc {
     const int stal = 44 ;          // źle !!!
     // ...
} ;
```

Odpowiedź: Do tego, by zainicjalizować składnik stały, służy właśnie konstruktor. Konkretnie: lista inicjalizacyjna konstruktora. Tym teraz się zajmiemy.

Oto schematycznie pokazany konstruktor. Zauważ dwukropek oddzielający listę inicjalizacyjną

```
klasa::klasa(argumenty) : lista_inicjalizacyjna
{
     // ciało konstruktora
}
```

Pokażmy listę inicjalizacyjną na przykładzie naszej klasy `abc`, którą teraz nieco rozbudujemy. Dodamy tam jeszcze inne składniki – tym razem już nie `const` – oraz rzeczony konstruktor z listą inicjalizacyjną

```
class abc {
     const int stal ;
     float x ;
          char c ;
     //--------- deklaracja konstruktora
     abc(float pp, int dd, char znak) ;
} ;
//////////////////////////////////////////////////////////
// ------------ definicja konstruktora
abc::abc(float pp, int dd, char znak) : stal(dd), c(znak)
{
     x = pp ;
}
```

Przyjrzyjmy się pierwszej linijce definicji konstruktora.

Zauważ, że po zamknięciu nawiasu z argumentami formalnymi zamiast zwykłego otwarcia klamry '{' rozpoczynającej ciało funkcji – widzimy dwukropek.

Klamra, o której myślimy, jest za to niżej. To, co następuje po dwukropku, nazywa się właśnie **listą inicjalizacyjną**. Na niej jest kilka pozycji oddzielonych przecinkami. Zauważ, że lista inicjalizacyjna pojawia się tylko przy definicji konstruktora – przy deklaracji jej nie było.

Łatwo to zapamiętać tak: Lista inicjalizacyjna to nie „cecha" konstruktora, ale lista konkretnych prac, które ma on wykonać.

Lista inicjalizacyjna specyfikuje jak należy zainicjalizować niestatyczne składniki klasy.

Wykonanie konstruktora składa się bowiem z dwóch oddzielnych etapów.

- Etap 1: inicjalizacja składników.
  - wykonywany jest właśnie dzięki liście inicjalizacyjnej.

- Etap 2: przypisania i inne akcje
  - to instrukcje wykonywane w ciele konstruktora. Tam możemy dać instrukcje przypisania (a także inne, które wydają się nam przydatne w konstruktorze).

Ten etap drugi jest nam znany – robiliśmy to już wielokrotnie.

Przyjrzyjmy się inicjalizacji analizując pozycje na liście. Pozycję

```
stal(dd)
```

rozumieć należy jako:

składnik `stal` zainicjalizować wartością wyrażenia w nawiasie (czyli u nas wartością argumentu `dd`).

Potem następuje kolejna pozycja na liście: `c(znak)` – oczywiście już wiesz, że chodzi o inicjalizację składnika `c` wartością wyrażenia `(znak)`

Ta druga pozycja na liście nie była konieczna. Składnik `c` nie ma przydomka `const`, dlatego nie musi być koniecznie inicjalizowany. Chcąc nadać mu wartość, można to uczynić w ciele konstruktora – tak, jak to zrobione jest w wypadku składnika `x`.

```
x = pp ;
```

Dla składnika nie-`const` obie formy są dopuszczalne. Przyznasz jednak, że ten sposób zapisu w liście inicjalizacyjnej jest krótszy.

Podsumujmy:

Składnikowi nie-`const` możemy w konstruktorze nadać wartość na dwa sposoby:
– przez listę inicjalizacyjną konstruktora,
– przez zwykłe podstawienie w ciele konstruktora,
Składnikowi `const` można nadać wartość początkową tylko za pomocą listy inicjalizacyjnej.

Ważne:

Lista inicjalizacyjna nie może inicjalizować składnika `static`

(przypominam, że składnik statyczny klasy, to składnik, który jest wspólny dla wszystkich obiektów tej klasy).

Ciekawostką jest to, że nie musimy wcale postępować tak grzecznie, jak to zrobiliśmy teraz. Inicjalizować możemy nie tylko argumentem konstruktora, ale nawet jakimś wyrażeniem, w którym możemy wywołać jakąś funkcję składową lub zwykłą. Oto przykłady wyrażeń, które mogłyby się znaleźć na liście.

```
stal(dd)
stal(dd +4)
stal( funkcja(dd) )
stal( x+ funkcja(dd*dd) +2)
```

14.6 Konstrukcja obiektu, którego składnikiem jest obiekt innej klasy

Wspominaliśmy już, że daną składową jakiejś klasy może być nie tylko obiekt typu `int`, czy `float`, ale nawet obiekt innej klasy. Jest to sytuacja bardzo często spotykana w życiu codziennym:

> Składnikiem obiektu klasy odbiornik radiowy są obiekty klasy tranzystor, obiekty klasy kondensator itd. Oczywiście składnikiem takiej klasy są także zwykłe liczby np. cena, waga, rok produkcji.

Zastanówmy się nad tym, jak konstruowany jest taki obiekt. Zanim zbuduje się ten główny obiekt (radio) trzeba najpierw skonstruować obiekty składowe (tranzystory). Nie można postąpić odwrotnie i najpierw zbudować obiekt główny, a potem zabrać się do budowy obiektów składowych. Chociażby dlatego, że nie wiemy wówczas jak dużo zarezerwować miejsca na główny obiekt. (Jaka duża ma być obudowa tego radia).

Cała ta praca wykonywana jest automatycznie, a jeśli o niej mówimy, to tylko po to, byś był świadom faktu, że konstruktory obiektów składowych wykonują się najpierw, a dopiero potem rusza do akcji konstruktor głównego obiektu.

Stanie się to jasne, gdy przyjrzymy się przykładowi

Załóżmy, że chcemy tu zbudować klasę, która reprezentować będzie panel, na którym znajduje się kilka przyrządów.

Przyrządy te już kiedyś przerabialiśmy i mamy klasę służącą do ich budowania. Załóżmy że jej definicja jest w pliku `przyrzad.h`

Nie chodzi mi tu o to, byś od razu zaczął sobie przypominać jak wyglądała realizacja klasy `przyrzad`. Zrobiliśmy to raz, działało – i od tej pory więcej nas to już nie interesuje. Aby z klasy korzystać interesuje nas tylko deklaracja publicznych składników klasy `przyrzad`. Przypomnę więc tylko ten fragment

```
public :                          // publiczne funkcje składowe
     przyrzad (int, int, char *, char *, int = 0) ;
     przyrzad (void) ;

     void zmien (int w ) ;
     void narysuj (void);
```

Od tej pory cała nasza praca z tą klasą polega na wywoływaniu tych funkcji.

Kilka słów o klasie `panel`

Panel jest jakby tablicą przyrządów. Na tym naszym panelu znajdują się dwa przyrządy. Są one więc składnikami obiektu klasy `panel`.

Jeśli zdefiniuję obiekt klasy `panel`, to na ekranie pojawi mi się tablica rozdzielcza składająca się z dwóch przyrządów. Jeśli zdefiniuję następny obiekt klasy panel, to w żądanym miejscu ekranu pojawi mi się następna tablica z dwoma przyrządami.

Oto program, w którym definiujemy klasę `panel` wykorzystując już istniejącą klasę `przyrzad`. Zakładam, że definicja klasy przyrząd znajduje się w pliku nagłówkowym `przyrz.h`

```
#include <conio.h>                          // gotoxy, cprintf
#include <dos.h>                            // delay
#include "przyrz.h"                                          // ❶
/*****************************************************/
inline void napisz(int x, int y, char *co)                   // ❷
{
     gotoxy(x,y) ; cputs(co);
}
//////////////////////////////////////////////////////////////
class panel {                                                // ❸
          int poz_x ;
     const int poz_y ;

     przyrzad predkosciomierz ;
     int *wsk1 ;

     przyrzad drugi ;
     int *wsk2 ;

 public :
     // konstruktor
     panel(    char *nazw,                                   // ❹
               int x, int y,
               int * zrodlo_sygnalu1,
               int * zrodlo_sygnalu2,
               char *opis2, char *jedn2) ;
     // destruktor
     ~panel();
     // zwykła funkcja składowa
     void aktualizuj() ;
};
//////////////////// koniec def klasy panel /////////////////

// ----------------definicja konstruktora
panel::panel( char *nazw, int x, int y,                      // ❺
              int * zrodlo_sygnalu1,
              int * zrodlo_sygnalu2,
              char *opis2, char *jedn2)
                   : poz_x(x),          // <---lista incjalizacyjna.
                     poz_y(y),
                     predkosciomierz(x, y+3,
                                  "Predkosc", "km/h"),
                     drugi(x, y+7, opis2, jedn2)
{
     wsk1 = zrodlo_sygnalu1 ,                                // ❻
     wsk2 = zrodlo_sygnalu2 ;
     napisz(poz_x, poz_y, nazw);
}
/*****************************************************/
panel::~panel()                                              // ❼
{
     napisz(poz_x, poz_y, " - ZLOMOWANY - ");
}
/*****************************************************/
void panel:: aktualizuj()
{
```

```
     predkosciomierz.zmien(*wsk1) ;                    // ❽
     drugi.zmien(*wsk2) ;
}
/*****************************************************/
main()
{
int  predkosc = 0,
     azymut = 270 ;

     clrscr(); // skasowanie dotychczasowej treści ekranu

     // definicja obiektu klasy panel                  // ❾
     panel kabina("Panel pilota", 1,2,
                       &predkosc, &azymut,
                       "kurs ", "stopni");

     for(int i = 0 ; i < 50 ; i++)
     {
          // imitujemy zmianę w czasie lotu
          predkosc = 60 + i ;
          azymut = 270 + (i % 3) ;

          // panel to pokazuje
          kabina.aktualizuj() ;                        // ❿
          delay(200) ;
     }
     // --- bawimy się dalej -----

     int  paliwo = 500 ;
     // definiujemy drugi panel                         ⓫⓫

     panel maszynownia("Panel mechanika", 40,2,
                       &predkosc, &paliwo,
                       "Zuzycie paliwa", "litrow");

     for( ; i < 100 ; i++)
     {
          // imitujemy zmianę w czasie lotu
          predkosc = 60 + i ;
          azymut = 270 + (i % 3) ;
          paliwo -- ;

          kabina.aktualizuj() ;
          maszynownia.aktualizuj() ;
          delay(200);
     }
}
```

❒ **Ekran pod koniec wykonywania programu będzie wyglądał** *mniej więcej* **tak**

```
Panel pilota                    Panel mechanika

 _____________________           _____________________
I Predkosc            I         I Predkosc            I
I   159        km/h   I         I   159 km/h          I
I_____________________I         I_____________________I

 _____________________           _____________________
I kurs                I         I Zuzycie paliwa      I
I    270     stopni   I         I   450        litrowI
I_____________________I         I_____________________I
```

Ciekawsze punkty programu

❶ Włączenie pliku z definicją klasy `przyrzad` znajdującego się w bieżącym katalogu. (Bieżącym, bo nazwa ujęta w cudzysłów).

❷W różnych wersjach kompilatorów różnie odbywa się wypisywanie na ekran w określone miejsca. Dlatego tutaj zdefiniowałem funkcję `napisz`, która odpowiada za napisanie stringu na ekranie w miejscu o współrzędnych x, y. Nie musisz wiedzieć jak się to odbywa. Treść tej funkcji dostosowana jest do kompilatora Borland C++, jeśli masz inny, to musisz zmienić tylko tę funkcję. Funkcja zdefiniowana jest jako `inline` – żeby wykonywała się szybciej. Dlatego więc musi być w programie na samej górze, jeszcze przed pierwszym jej wywołaniem.

❸Definicja klasy panel. Składnikami jej są:

- dwa obiekty typu `int`. Drugi jest typu `const`, bo będę chciał coś ciekawego pokazać. W tych dwóch składnikach przechowujemy pozycję panelu na ekranie ;
- obiekty klasy `przyrzad`. Jeden egzemplarz obiektu nazywamy `predkosciomierzem`, a drugi nazywa się po prostu `drugi`. Na razie jeszcze nie wiemy co konkretnie będzie pokazywał. Ani jedna, ani druga nazwa nie ma znaczenia. Oba przyrządy równie dobrze mogłyby się nazywać A oraz B ;
- dwa wskaźniki `int*`. Posłużą mi one do pokazywania na te zmienne, które mają być wyświetlane przez przyrząd;
- funkcje składowe, te jednak omówimy niżej.

❹ Deklaracja konstruktora. Nie ma w niej nic nadzwyczajnego, ale wytłumaczę się ze znaczenia poszczególnych argumentów:

`nazw` – tak przysyłamy string z tytułem panelu,

`x, y` – to współrzędne panelu na ekranie – konkretnie lewego górnego rogu,

`zrodlo_sygnalu1`, `zrodlo_sygnalu2`, to dwa wskaźniki pokazujące na te zmienne typu `int`, które mają być wyświetlane przez przyrządy,

`opis2`, `jedn2` – ponieważ nie byłem zdecydowany jaką tabliczkę ma mieć wymalowany `drugi` przyrząd, dlatego w ostatniej chwili, w momencie konstrukcji panelu przyślę żądane napisy.

❺ **To jest najważniejsza linijka w tym przykładzie.** Widzimy tu pierwszą linijkę definicji konstruktora. Nie jest to byle jaki konstruktor, ale konstruktor klasy, która zawiera w sobie obiekty innej klasy.

Zauważ, że konstruktor ma bardzo rozbudowaną listę inicjalizacyjną.

Przyjrzyjmy się inicjalizacji analizując pozycje na tej liście

```
poz_x(x)
```

Oznacza to, że składnik `poz_x` należy zainicjalizować wartością argumentu `x`. Składnikowi temu moglibyśmy równie dobrze przypisać wartość początkową już w ciele konstruktora. To dlatego, że ten składnik nie ma przydomka `const`.

Dalej na liście widzimy

```
poz_y(y)
```

Tutaj zajmujemy się składnikiem, przy którym stoi przydomek `const`. Takiemu obiektowi można nadać wartość tylko w momencie jego narodzin. Potem już przepadło.

Czyli: w wypadku składnika `poz_y` nie moglibyśmy przypisać mu wartości w ciele konstruktora. To już byłoby za późno. Inicjalizacja odbyć się musi tutaj, za pomocą listy inicjalizacyjnej.

Dalej na liście inicjalizacyjnej widzimy następujące wyrażenie

```
predkosciomierz(x, y+3, "Predkosc", "km/h")
```

Mam nadzieję, że rozpoznajesz. Jest to wywołanie konstruktora dla obiektu `predkosciomierz` będącego składnikiem panelu. To wywołanie konstruktora musimy umieścić na liście inicjalizacyjnej.

Zapamiętaj

> Obiekt jakiejś klasy – będący składnikiem innej klasy może być inicjalizowany jedynie za pomocą listy inicjalizacyjnej (czyli w etapie 1).

> Nie można próbować uruchomić jego konstruktora takiego obiektu składowego z wnętrza (z ciała) konstruktora klasy, w której obiekt zawiera (Etap 2). Wtedy już za późno.

Co zrobić jeśli obiekt składowy nie ma konstruktora?

Rzeczywiście – może być taka sytuacja, posiadanie konstruktora nie jest przecież obowiązkowe. Wówczas po prostu na liście inicjalizacyjnej nie umieszcza się go.

Jeśli klasa ma konstruktor wywoływany bez żadnych argumentów – czyli tzw. konstruktor domniemany – i ten konstruktor chcemy użyć, to można tę pozycję na liście inicjalizacyjnej pominąć.

Jeśli jednak klasa obiektu składowego ma konstruktory (wszystkie z jakimiś argumentami), to pominięcie wywołania na liście spowoduje błąd kompilacji. Kompilator będzie się domagał określenia, którą wersję konstruktora ma uruchomić.

Na naszej liście ostatnią pozycją jest

```
drugi(x, y+7, opis2, jedn2)
```

To jest wywołanie konstruktora drugiego obiektu składowego. Jak widać, argumenty wysyłane konstruktorowi są to opisy dostarczone przez wywołującego konstruktor klasy `panel`. Zwróć uwagę na wywołanie ❶❶.

Kiedy wykonane zostaną wszystkie inicjalizacje z listy, wtedy dopiero zaczyna się wykonywanie ciała konstruktora klasy „zawierającej" czyli klasy `panel`.

❻ Wewnątrz tego konstruktora widzimy dwa przypisania wskaźników. Oczywiście i to mogłem umieścić na liście inicjalizacyjnej, ale nie chciałem Cię przerażać jej długością.
W ciele jest też wywołanie funkcji wypisującej w określonym miejscu na ekranie tytuł panelu.

❼ Działanie destruktora polega na tym, że w miejscu, gdzie był tytuł panelu, pojawia się informacja, że panel jest zezłomowany.
Ciekawsza jest tutaj kolejność zdarzeń. O ile w wypadku *tworzenia* obiektów najpierw ruszły do pracy konstruktory obiektów składowych, a dopiero potem konstruktor klasy zawierającej, to w wypadku destruktorów jest odwrotnie:

Zapamiętaj:

> Najpierw rusza do pracy destruktor klasy zawierającej obiekty, a dopiero potem wykonywane są destruktory obiektów składowych.

To także łatwo zapamiętać: Jeśli chcesz dokonać destrukcji obiektu klasy radio, to najpierw rozwalasz młotkiem obudowę, a dopiero wtedy możesz zabrać się za destrukcję tranzystorów.

W naszym wypadku klasa `przyrzad` nie miała destruktora. Nic nie jest wobec tego aktywowane.

❿ Funkcja składowa `aktualizuj` wywołuje funkcję składową

```
przyrzad::zmien(int i) ;
```

na rzecz obu obiektów składowych. Zauważ w ❽, co wysyłam im jako argument. Wyrażenie:

```
*wsk1
```

jest wartością tej zmiennej typu `int`, na którą pokazuje wskaźnik `wsk1`. Jest to po prostu jakaś liczba całkowita.

❶❶ W funkcji `main` widzimy definicję egzemplarza obiektu klasy `panel`. Egzemplarz ten nazywa się `kabina`. Z analizy argumentów wywołania widzimy, że na panelu ma być napisany tytuł "Panel pilota". Przyrząd pierwszy ma pokazywać stan zmiennej `predkosc`, przyrząd drugi stan zmiennej `azymut`. Wysyłamy też napisy, które mają się pojawić na drugim przyrządzie.
W dalszej części programu widzimy pętlę `for`, która ma imitować nam ciągłe zmiany zmiennych `predkosc` i `azymut`. Na te zmienne spojrzą oba przyrządy z panelu `kabina` w chwili, gdy wywołam funkcję składową `aktualizuj` ❿

Dalej w programie widzimy definicję innego obiektu klasy `panel`. Obiekt ten nazywa się `maszynownia`. Z argumentów konstruktora widzimy, że będzie on pokazywał stan zmiennych `predkosc`, oraz `paliwo`.

Patrząc na argumenty określające jego pozycję na ekranie widzimy, że pojawi on się na prawej stronie ekranu.

Zapamiętaj, zasada jest taka:

> Obiekty składowe wewnątrz klasy, to jakby jej goście. Przy konstruowaniu obiektu klasa najpierw daje pierwszeństwo gościom, a dopiero potem myśli o swoim konstruktorze.

14.7 Konstruktory nie-publiczne ?

Konstruktor jest zwykle deklarowany jako publiczny. Wydaje się to oczywiste, bo przecież obiekty powołuje się do życia będąc w „świecie zewnętrznym" w stosunku do klasy.

A jednak nie jest to takie oczywiste. Konstruktor może być nie-publiczny. Konstruktor jest składnikiem klasy i jako takiego – obowiązują go również zwykłe reguły dostępu ustalane za pomocą słów `public/protected/private`.

Klasa, która nie ma publicznych konstruktorów nazywana jest **klasą prywatną**.

Nasuwa się pytanie:

Jak są tworzone obiekty danej klasy, skoro nie można sięgnąć do konstruktora, bo jest niedostępny?

Bardzo prosto. Konstruktor jest niedostępny dla tzw. szerokiej publiczności, ale jest dostępny dla obiektów tej klasy. Konstruktor jest funkcją składową tej klasy, co prawda prywatną funkcją składową, ale obiekty tej klasy maja dostęp do prywatnych składników.

Słowem: obiekty tej klasy mogą być tworzone przez inne obiekty tej klasy.

„–Mam cię!" – zawołasz – „A skąd się weźmie pierwszy obiekt tej klasy?"

Dobre pytanie. Pierwszego obiektu rzeczywiście w ten sposób nie da się powołać do życia. Zastanów się jednak: kto jeszcze ma dostęp do prywatnych składników klasy, więc mógłby uruchomić prywatny konstruktor ?

Oczywiście — funkcja zaprzyjaźniona, albo klasa zaprzyjaźniona. Oto przykład:

```
class star {
     friend void kreator(char z ) ;
private :
     char znak ;
     star(int i);          // prywatny konstruktor
     // ...
};
```

Definicja samego konstruktora jest teraz nieistotna. Ważna jest natomiast definicja tej funkcji zaprzyjaźnionej.

```
void kreator(char z)
{
     star a('*') ;                    // obiekt automatyczny
     static star b('*') ;             // obiekt statyczny
     /* ... */                        // jakaś praca z nim
     wskaznik = new star('@') ;       // obiekt w „zapasie pamięci"
}
```

Jak widzimy, w funkcji `kreator` można zdefiniować obiekt klasy `star`. Co to znaczy, że *w obrębie bloku funkcji?*

To, że obiekty tam tworzone mogą być:

- – automatyczne lub statyczne – czyli ich nazwy znane są tylko wewnątrz tej funkcji. (Statyczne są dodatkowo nieśmiertelne).
- – tworzone operatorem `new`. Wtedy z takim obiektem można pracować nawet poza funkcją `kreator` i w dowolnym momencie go skasować.

Skoro konstruktory bywają przeładowane, a są one przecież zwykłymi funkcjami składowymi, to za pomocą słów `private/protected/public` możemy różnie określić dostęp do różnych wersji konstruktora. Innymi słowy jeden może być prywatny, inny publiczny itd.

Co to oznacza w praktyce? To, że jeden ze sposobów konstruowania obiektów jest dostępny dla szerokiej publiczności, a inny tylko dla „swoich". Próba utworzenia obiektu za pomocą konstruktora, do którego nie mamy prawa, zostanie odrzucona przez kompilator. (Tak samo, jak próba wywołania innej prywatnej funkcji). Jeśli chcemy – to możemy kreować obiekt, ale tylko w sposób, który jest nam dostępny.

```
class club {
     friend void przyjaciel() ;
     int x ;
     char tekst[20] ;

     club(int n, char *s) ;          // prywatny konstruktor
public:
     club(int n) ;                   // publiczny konstruktor

} ;
```

Jeśli w programie, wewnątrz funkcji `przyjaciel`, chcielibyśmy zdefiniować obiekt klasy `club`, to możemy posłużyć się takimi definicjami:

```
void przyjaciel()
{
     club jacek(5);
     club marek(7, "Kangur") ;
     // ...
}
```

Natomiast w innych miejscach programu pierwsza z tych definicji jest akceptowalna, a druga błędna:

```
club maciek(58) ;            // o.k.
club zosia(3, "Ruda") ;      // źle !!
```

Uwagą dla wtajemniczonych:

> Jeśli konstruktor jest oznaczony dostępem `protected`, to znaczy, że może być wywołany w sytuacjach identycznych jak `private`, a w dodatku może być wywołany z klasy pochodnej od danej klasy.

14.8 Konstruktor kopiujący (albo inicjalizator kopiujący)

Konstruktorem kopiującym w danej klasie `klasa` nazywamy konstruktor, który można wywołać z jednym argumentem poniższego typu

```
klasa::klasa( klasa &)
```

Argumentem jest, jak widać, referencja (przezwisko) obiektu danej klasy.

Konstruktor ten służy do skonstruowania obiektu, który jest kopią innego, już istniejącego obiektu tej klasy. Innymi słowy – zamiast w „zwykłym" konstruktorze wyszczególniać argumenty inicjalizacji – mówimy: chcę, by nowy obiekt był taki sam jak tamten, który posyłam na wzór.

> Zauważ, że w definicji powiedzieliśmy: konstruktor, który **można** wywołać z jednym argumentem. To dlatego, że dopuszcza się, by były jeszcze inne argumenty, ale domniemane.

Konstruktorem kopiującym klasy `K` jest więc

```
K::K(K &)
```

albo

```
K::K(K&, float = 51.45, int * = NULL)
```

Oczywiście „albo-albo". Deklaracja pierwsza jest bowiem jakby szczególnym wypadkiem drugiej, więc w definicji tej klasy nie mogą się one pojawić równocześnie.

Konstruktor kopiujący nie jest obowiązkowy. Jeśli go nie zdefiniujemy wówczas kompilator wygeneruje go sobie sam.

Konstruktor kopiujący inaczej można by nazwać **inicjalizatorem kopiującym**.

To dlatego, że działa on na prawach inicjalizacji, a nie przypisania (podstawienia). Co to oznacza? Oznacza to, że pracuje wtedy, gdy odbywa się inicjalizacja *nowego* obiektu, a nie zwykłe przypisanie.

Kiedy wywoływany jest konstruktor kopiujący ?

W kilku sytuacjach, które można najogólniej podzielić na:

- ❖ – gdy tego jawnie zażądamy,
- ❖ – bez naszej wiedzy.

Wywołanie konstruktora kopiującego na nasze życzenie

następuje wtedy, gdy tego jawnie zażądamy i definiujemy nowy obiekt w następujący sposób:

```
K obiekt_wzor ;                    // wcześniej zdefiniowany obiekt klasy K
// ...
K obiekt_nowy = K(obiekt_wzor) ;   // definicja nowego
```

Jak widać definiujemy tu nowy obiekt, a do konstruktora wysyłamy inny istniejący obiekt jako wzór. Założyłem tu oczywiście, że klasa `K` ma konstruktor kopiujący, i że ten stary obiekt jest godny tego, by go rzeczywiście kopiować.

Niejawne wywołanie konstruktora kopiującego klasy `K` następuje w kilku sytuacjach

a) **Podczas przesłania argumentów do funkcji – jeśli argumentem funkcji jest obiekt klasy `K`, a przesłanie odbywa się przez wartość.** Jak wiemy – nawet z typów wbudowanych – w obrębie funkcji argument jest kopiowany †) i funkcja pracuje na kopii. Jeśli więc argumentem jest obiekt klasy `K`, to musi istnieć narzędzie do zrobienia kopii obiektu tej klasy `K`. Słowem – musi zostać użyty konstruktor kopiujący klasy `K`. Wywołanie takiego konstruktora odbywa się bez naszego udziału. Załatwiają to same służby odpowiadające za przesyłanie argumentów do funkcji.

b) **Podczas, gdy funkcja jako swój rezultat zwraca przez wartość obiekt klasy `K`.** Wytłumaczenie jest identyczne jak wyżej. Także służby specjalne, czyli mechanizm zwrotu rezultatu funkcji przez wartość, posłuży się tym konstruktorem.
To, co stoi przy instrukcji `return`, staje się wzorcem do inicjalizacji obiektu chwilowego będącego wartością tej funkcji. Ten chwilowy obiekt nie jest już lokalny – jest widziany z zewnątrz, z zakresu z którego funkcję wywołaliśmy.

†) casus: fotografia babci

14.8.1 Przykład klasy z konstruktorem kopiującym

Najpierw wyjaśnienie do czego służy nasza klasa

Przy posługiwaniu się aparaturą pomiarową zachodzi konieczność cechowania urządzenia. Mówi się na to także: *kalibracja*. Polega to na tym, że należy sformułować zasadę zamiany jednej wielkości na inną.

Przykładowo:

> *Cechowanie wagi (takiej jak w sklepie spożywczym) to poznanie zależności: jak wychylenie wskazówki (zmierzone w milimetrach lub stopniach) zamienić na kilogramy. Po ustaleniu takiej zależności wychylenia od wagi maluje się na wadze skalę.*

W aparaturze naukowej często posługujemy się tzw. analizatorami amplitudy, które mierzoną wielkość fizyczną – na przykład energię kwantu promieniowania - zamieniają na liczbę od 1 do 8192. Ta liczba przychodzi z urządzenia pomiarowego do komputera. Liczba taka, to jakby podana w milimetrach wartość wychylenia wskazówki. Taką liczbę trzeba zamienić na wielkość, którą ona symbolizuje – czyli ciężar produktu albo energię kwantu promieniowania.

Często wystarczy proste równanie

```
waga      = (a * wychylenie) + b

energia   = (a * liczba) + b
```

Jeśli nic z tego nie zrozumiałeś – nie szkodzi

rozmawiamy przecież o konstruktorze kopiującym. Wystarczy wiedzieć, że nasza klasa składa się z tych dwóch liczb: a, b oraz z funkcji składowej, która oblicza powyższą zależność. Dodatkowo jest oczywiście nasz konstruktor kopiujący.

```
#include <iostream.h>
#include <string.h>
//////////////////////////////////////////////////////////
class kalibracja {
     float a, b ;                    // współczynniki kalibracji
     char nazwa[80] ;                    // nazwa                ❶
public :
     // -----------------konstruktor
     kalibracja(float wsp_a, float wsp_b, char * txt);

     // ---------------konstruktor kopiujący
     kalibracja( kalibracja & wzor) ;                    // ❷

     // --------------inne funkcje składowe
     float energia (int kanal)
     {
          return( (a * kanal) + b ) ;
     }
     char * opis() {return( nazwa) ; }                   // ❸

} ;
//////////////////////////////////////////////////////////
kalibracja::kalibracja(float wsp_a, float wsp_b,
```

```
                char * txt ) : a(wsp_a), b(wsp_b)
{                                                          // ❹
     strcpy(nazwa, txt) ;
}
/*******************************************************/
kalibracja::kalibracja(kalibracja & wzorzec)              // ❺
{
     a = wzorzec.a ;
     b = wzorzec.b ;

     // zamiast : strcpy(nazwa, wzorzec.nazwa) ;
     strcpy(nazwa,                                        // ❶❻
          "-- To ja, konstruktor kopiujacy !!! --");
}
// ------------------------------------------------
void fun_pierwsza( kalibracja odebrana) ;
kalibracja fun_druga(void) ;
/*******************************************************/
main()
{
kalibracja kobalt(1.07, 2.4, "ORYGINALNA KOBALTOWA ");

// Różne warianty tego samego
kalibracja europ(kobalt) ;                                // ❻
//kalibracja europ = kalibracja(kobalt) ;
//kalibracja europ = kobalt ;

     cout << "O ktory kanal widma chodzi ? : " ;

     int kanal ;
     cin >> kanal ;                                       // ❼

     cout << "\nWedlug kalibracji kobalt, \nopisanej jako "
          << kobalt.opis()
          << "\nkanalowi nr " << kanal                   // ❽
          << " odpowiada energia "
          << kobalt.energia(kanal) << endl ;

     cout << "\nWedlug kalibracji europ, \nopisanej jako "
          << europ.opis() //
          << "\nkanalowi nr " << kanal                   // ❾
          << " odpowiada energia "
          << europ.energia(kanal) << endl ;

     cout <<"\nDo funkcji pierwszej wysylam kalibracje "
          << kobalt.opis() << endl ;

     fun_pierwsza(kobalt) ;                               // ❿

     cout << "\nTeraz wywolam funkcje druga, a jej"
               " rezultat\n"
               "podstawie do innej kalibracji \n" ;

     cout << "Obiekt chwilowy zwrocony jako "
               "rezultat funkcji \nma opis "
          << ( fun_druga() ).opis()                       // ❶❷
```

```
            << endl ;
}
/*****************************************************/
void fun_pierwsza( kalibracja odebrana)
{
     cout << "Natomiast w funkcji pierwszej "
                "odebralem te kalibracje\n"
                "\topisana jako "
            << odebrana.opis()                    // ❶❶
            << endl ;
}
/*****************************************************/
kalibracja fun_druga(void)                        // ❶❸
{
     kalibracja wewn(2, 1, "WEWNETRZNA") ;        // ❶❹
     cout << "W funkcji fun_druga definiuje kalibracje"
                " i ma \nona opis : "
            << wewn.opis()
            << endl ;
     return wewn ;                                // ❶❺
}
```

Na ekranie, po wykonaniu tego programu, pojawi się następujący tekst

```
O ktory kanal widma chodzi ? : 100
Wedlug kalibracji kobalt,                          ❽
opisanej jako ORYGINALNA KOBALTOWA
kanalowi nr 100 odpowiada energia 109.400009

Wedlug kalibracji europ,
opisanej jako -- To ja, konstruktor kopiujacy !!! -- ❾
kanalowi nr 100 odpowiada energia 109.400009

Do funkcji pierwszej wysylam kalibracje ORYGINALNA KOBALTOWA
Natomiast w funkcji pierwszej odebralem te kalibracje
     opisana jako -- To ja, konstruktor kopiujacy !!! --

Teraz wywolam funkcje druga, a jej rezultat
podstawie do innej kalibracji
W funkcji fun_druga definiuje kalibracje i ma
ona opis : WEWNETRZNA
Obiekt chwilowy zwrocony jako rezultat funkcji
ma opis -- To ja, konstruktor kopiujacy !!! --
```

Spójrzmy na ciekawsze miejsca programu

❶ Oprócz współczynników równania wprowadzamy też tablicę znakową. Będziemy w niej przechowywać opis słowny.

❷ Konstruktor kopiujący (tylko deklaracja).

❸ Jedną z funkcji składowych jest funkcja, która zwraca adres tablicy znakowej. Dawno nie przypominałem, że nazwa tablicy jest równocześnie adresem jej zerowego elementu.

❹ Definicja konstruktora (takiego zwykłego). Widzimy, że inicjalizację składników `a` i `b` załatwiłem w liście inicjalizacyjnej. Ten zapis jest krótszy niż napisanie w ciele konstruktora

```
a = wsp_a ;
b = wsp_b ;
```

W ciele jest natomiast wywołanie funkcji bibliotecznej, która kopiuje string (przysłany jako argument) do tablicy `nazwa` – będącej składnikiem klasy.

❺ Definicja konstruktora kopiującego. O tym, że to właśnie on – mówi nam argument: jest to referencja do obiektu swojej własnej klasy.
W ciele konstruktora widzimy skopiowanie składników obiektu wzorcowego do tego obiektu, na rzecz którego wywołano konstruktor.

Słyszałeś co powiedziałem? Do: **tego** – a więc `this`. Zatem pierwsze dwie linijki można inaczej zapisać jako

```
this -> a = wzorzec.a ;
this -> b = wzorzec.b ;
```

Dalej też by można przekopiować treść tablicy `nazwa`, ale wtedy kopia niczym nie różniłaby się od wzorca, dlatego wpisujemy tam tekst, który będzie nam udowadniał, że ruszył do pracy ten właśnie konstruktor kopiujący.

❻ Jest to *jawne* uruchomienie konstruktora kopiującego. No, może nie tak całkiem jawne, ale już przyzwyczailśmy się, że poniższe zapisy są jakby tym samym:

```
kalibracja europ = kalibracja(kobalt) ;
kalibracja europ = kobalt ;
```

A jeśli się nie przyzwyczailiśmy, to przypominam jakby to wyglądało dla typów wbudowanych

```
float wzor = 6.66 ;                    // obiekt wzorcowy

float kopia2 = float(wzor) ;           // całkiem jawnie
float kopia1 = wzor ;                  // prawie-jawnie
```

Jeśli linijka z `kopia2` cię dziwi, to przypominam, że operacja rzutowania może mieć dwie formy: `(float)wzor`, oraz `float(wzor)`. Tę drugą formę stosujemy tutaj.

Tym sposobem użyliśmy jawnie (a przynajmniej **świadomie**) konstruktora kopiującego.

❼ Cała kalibracja jest po to, by na żądanie przeliczyć milimetry na kilogramy lub kanały na energię. Tutaj pytamy użytkownika o wartość do przeliczenia.

❽ To, co program odpowiada, nie jest takie istotne w tym momencie. Przyjrzyjmy się tablicy `nazwa` – bo tutaj zorientujemy się kiedy mieliśmy do czynienia z kopią. Patrząc na ekran widzimy, że w tym miejscu mamy oryginał. Przypominam, że wywołanie funkcji składowej `opis` na rzecz obiektu klasy `kalibracja` owocuje adresem tablicy znakowej `nazwa`. Czyli to samo co

```
cout << kobalt.nazwa ;
```

Z tym, że nie możemy tak tego zrobić (jest ona składnikiem prywatnym), więc posługujemy się funkcją składową.

❾ Wywołanie tejże funkcji składowej na rzecz obiektu `europ` pokazuje tkwiącą tam treść – udowadniającą, że obiekt powstał przy użyciu naszego konstruktora kopiującego.

❿ Wywołanie funkcji, która, jak łatwo sprawdzić w definicji, przyjmuje argument będący obiektem klasy `kalibracja`. Obiekt ma być przesłany przez wartość. Linijkę wcześniej wypisaliśmy na ekranie opis tego obiektu, który teraz służy jako argument.

❶❶ W tym miejscu wewnątrz funkcji wypisujemy opis z obiektu, który otrzymaliśmy. Co widzimy na ekranie? Oczywiście - użyty został nasz konstruktor kopiujący. To chciałem pokazać. Najważniejsze: użyty został niejawnie. My go nie uruchamialiśmy.
Przypominam więc:

> Konstruktor kopiujący klasy `K` jest uruchamiany niejawnie w sytuacji, gdy do jakiejś funkcji wysyłamy obiekt klasy `K` przez wartość. Użyty on zostaje jako narzędzie do zrobienia lokalnej kopii przysłanego obiektu.

❶❷ Z tej linijki muszę się gęsto tłumaczyć. Napiszmy to prościej. Co to jest?

```
( fun_druga() ).opis()
```

Najpierw odpowiem, a potem udowodnię. Jest to wywołanie funkcji składowej `opis` na rzecz obiektu chwilowego przysłanego jako rezultat wywołania funkcji składowej `fun_druga`. Inaczej mówiąc coś takiego

```
( obiekt_chwilowy ).opis()
```

Teraz pokażę skąd się bierze obiekt chwilowy. Spójrzmy więc na definicję funkcji `fun_druga` ❶❸. Deklarację tej funkcji czytamy: `fun_druga` jest funkcją wywoływaną bez żadnych argumentów, a która jako rezultat zwraca obiekt klasy (typu) `kalibracja`.

Zwrot obiektu odbywa się przez wartość (na mocy domniemania).

❶❹ Wewnątrz funkcji definiujemy obiekt klasy `kalibracja`. Ten egzemplarz nazywa się `wewn`. Jak nam wiadomo, jest to obiekt lokalny, automatyczny, czyli jego nazwa jest znana tylko w tej funkcji (zakres ważności), a umrze on w chwili, gdy opuścimy tę funkcję (czas życia).
Coś sobie w tej funkcji na tym obiekcie pracujemy – w naszym wypadku jest to wypisanie na ekranie jego opisu. Już nawet nie obliczamy energii promieniowania, bo to jest nudne w porównaniu z naszymi konstruktorami kopiującymi.

❶❺ Nadchodzi moment opuszczenia funkcji. Na moment przed swą zagładą obiekt lokalny stawiany jest obok słowa `return`. To sprawia, że służby specjalne odpowiadające za mechanizm zwrotu rezultatu funkcji – które tworzą obiekt chwilowy klasy `kalibracja` - do niego kopiują obiekt `wewn`.
Nie mogą mu uratować życia, bo został on założony na stosie – czyli jakby na piasku plaży, do której zbliża się przypływ. Kopiują go jednak za pomocą naszego konstruktora kopiującego.

Obiekt ginie, ale funkcja jako wartość zwraca obiekt chwilowy będący jego kopią. Kopia nie jest oczywiście wierna – o to już się postaraliśmy w linijce ❶❻ podmieniając tekst opisu.

Teraz chyba rozumiesz dlaczego wyrażenie

```
( fun_druga() )
```

ma wartość będącą chwilowym obiektem klasy kalibracja. Na rzecz takiego obiektu wywołujemy funkcję składową `opis`, po czym rezultat – string opisu – piszemy na ekranie. Spójrz na ekran. Widzisz, że jest tam opis udowadniający, że pracował nasz własny konstruktor kopiujący. Uruchomiono go bez naszego udziału.

Obiekt jest chwilowy. Po linijce programu, w której powstał, ginie. Żałuję, że w tej chwili jeszcze nie umiemy się (swobodnie) posługiwać operacjami z ekranem, bo wtedy zobaczyłbyś, że ten obiekt chwilowy ma w sobie współczynniki 2, 1 – te które skopiował z obiektu `wewn`.

Zapamiętaj:

> Konstruktor kopiujący klasy `K` jest uruchamiany niejawnie w sytuacji, gdy funkcja zwraca przez wartość obiekt klasy `K`
> Użyty on zostaje jako narzędzie do zrobienia obiektu chwilowego będącego rezultatem funkcji.

Dlaczego konstruktor kopiujący pracuje na argumencie przesłanym przez referencję, a nie na argumencie przysłanym przez wartość?

Konstruktor kopiujący klasy `kkk` ma, jak wiemy, formę

```
kkk::kkk( kkk &) ;
```

Czyli jako argument przyjmuje referencję obiektu klasy `kkk`. Referencja, jak wiemy, daje mu prawo pracowania na oryginale przysłanego argumentu.
Gdyby przysłano mu obiekt przez wartość, to tego prawa by nie miał. Referencja mu je daje. Może on więc nawet zmodyfikować oryginał. Zupełnie nie tego oczekujemy od konstruktora kopiującego. Kopiować powinien nie zmieniając oryginału, to chyba oczywiste.

Zapytasz: „No to – skoro jest takie ryzyko – dlaczego nie każemy temu konstruktorowi odbierać obiekt przez wartość?"

Masz świętą rację! – tak by było najlepiej, niestety nie da się.

Wyobraź sobie, że do naszej funkcji `fun_pierwsza` wysyłamy obiekt klasy `kalibracja`. Ponieważ przesłanie odbywa się przez wartość, dlatego musi zostać zrobiona kopia, więc rusza do pracy konstruktor kopiujący.

Konstruktor jako argument przyjmuje: – wiadomo – obiekt klasy `kalibracja`. Z tym, że teraz przesyła go nie przez referencję, ale – w myśl naszego najnowszego pomysłu – przez wartość. Skoro więc przez wartość, to znaczy, że w tym konstruktorze kopiującym trzeba zrobić kopię obiektu.

W porządku, aby zrobić kopię w dla konstruktora kopiującego uruchamiamy konstruktor kopiujący – tylko, że ten znowu wymaga kopii, więc znowu konstruktor kopiujący czyli robienie kopii...

I tak dalej – błędne koło nieskończonych wywołań konstruktora kopiującego.

Pomysł nie był więc dobry. Zwróć też uwagę, że to całe nieszczęście nie musiało być wynikiem naszej wynalazczości. Ponieważ przesłanie przez referencję różni się od przesłania przez wartość tylko jednym małym znaczkiem `&` (ampersand) w deklaracji funkcji, to jeśli przez nieuwagę o tym znaczku zapomnisz, to...

Nie, nie obawiaj się — Nasz przyjaciel kompilator do tego już nie dopuści.

Poniższy tekst jest przeznaczony tylko dla dociekliwych. Jeśli jesteś nieco zmęczony tym paragrafem - bez wahania opuść poniższych kilka linijek.

Jak dostać piątkę z C++ ?

Pewien profesor jednej z polskich politechnik zwierzył mi się kiedyś, że gdy omawia ze studentami ten paragraf „Symfonii C++" - a konkretnie ten nasz ostatni przykład - zadaje studentom pytanie, obiecując, że za poprawną odpowiedź natychmiast wpisuje do indeksu piątkę. Jeśli chciałbyś łatwo zaliczyć C++ na piątkę — przeczytaj poniższych kilka zdań. `

Otóż w naszym ostatnim przykładzie, pod koniec funkcji `main` występują takie rozkazy:

```
cout << "\nTeraz wywolam funkcje druga, a jej"
         " rezultat\n"
         "podstawie do innej kalibracji \n" ;

cout << "Obiekt chwilowy zwrocony jako "
         "rezultat funkcji \nma opis "
     << ( fun_druga() ).opis()                    // ❶❷
     << endl ;
```

Jak widać są to dwie instrukcje wypisujące na ekran tekst. W drugiej jest dodatkowo wywołanie funkcji składowej `opis` na rzecz obiektu chwilowego (zwracanego przez `fun_druga`). Wykonanie tych instrukcji powoduje wypisanie na ekranie następującego tekstu:

```
Teraz wywolam funkcje druga, a jej rezultat          ← 1
podstawie do innej kalibracji
W funkcji fun_druga definiuje kalibracje i ma        ← !!
ona opis : WEWNETRZNA
Obiekt chwilowy zwrocony jako rezultat funkcji       ← 2
ma opis -- To ja, konstruktor kopiujacy !!! --
```

Profesor K. pyta studentów dlaczego między tekstem **1** (`"Teraz wywolam..."`), a tekstem **2** (`"Obiekt chwilowy zwrocony..."`) - pojawił się tekst `"W funkcji fun..."`.

Oczywiście jest to efekt działania funkcji `opis`, ale dlaczego tekst ten pojawił się między wspomnianymi tekstami, a nie poniżej - czyli tak

```
Teraz wywolam funkcje druga, a jej rezultat          ← 1
podstawie do innej kalibracji
Obiekt chwilowy zwrocony jako rezultat funkcji       ← 2
ma opis -- To ja, konstruktor kopiujacy !!! --
W funkcji fun_druga definiuje kalibracje i ma        ← !!
ona opis : WEWNETRZNA
```

Oto dlaczego. W trakcie wykonywania instrukcji

```
cout << "Obiekt chwilowy zwrocony jako "
         "rezultat funkcji \nma opis "
```

```
<< ( fun_druga() ).opis()                    // ❶❷
<< endl ;
```

było tak:

- Komputer najpierw zaczął przygotowywać wszystko to, czym ta instrukcja ma się zająć. Mówiąc "wszystko to" - mam na myśli wszystkie stringi i wyrażenia, które w tej instrukcji oddzielają trzykrotnie użyte znaczki <<.
- Jednym z tych wyrażeń było wywołanie funkcji `opis`. Jak wiemy - jest w niej inna instrukcja wypisywania na ekran - i to ona posłała na ekran tekst `"W funkcji fun..."`. Po wykonaniu tej funkcji `opis` – przygotowania były dalej kontynuowane. (Czyli komputer zajął się manipulatorem `endl`).
- Gdy wszystko było już przygotowane i czekało zapewne na stosie - odbyło się wypisanie tego przygotowanego tekstu. W naszym wypadku zaczynał się on od słów `"Obiekt chwilowy..."`.

Stąd taki wygląd ekranu. Gdyby Ci to nie odpowiadało - wystarczy ten fragment programu zapisać tak

```
cout << "\nTeraz wywolam funkcje druga, a jej"
         " rezultat\n"
         "podstawie do innej kalibracji \n" ;

cout << "Obiekt chwilowy zwrocony jako "
         "rezultat funkcji \nma opis " << endl ;

// osobno poniżej
cout << (fun_druga()).opis() ;
```

Czyli wywołanie funkcji `opis` umieść w ***osobnej*** instrukcji.

14.8.2 Konstruktor kopiujący gwarantujący nietykalność

Wiemy już, że niemożliwe jest pozbawienie konstruktora kopiującego prawa modyfikowania oryginału. Czyli temu konstruktorowi nie można po prostu ufać. Nie ufa mu też kompilator, przekonajmy się.

Załóżmy, że chcemy tego konstruktora użyć do kreacji obiektu na wzór obiektu z przydomkiem `const`. Niech tym cennym wzorcowym obiektem będzie wzorzec metra z paryskiej dzielnicy Sevres. Kompilator – zaprotestuje.

```
class metr ;                  // klasa z konstruktorem kopiującym
                              // metr::metr(metr & wz) ;

const metr wzorzec_metra ;    // z Sevres (Paryż)
                              // (definicja obiektu const)

metr krawiecki = wzorzec_metra ;  // błąd - konstruktor
                                  //kopiujący nie gwarantuje, że nie uszkodzi wzorca.
```

Skoro konstruktor ma warunki na to, by uszkodzić cenny wzorzec, więc kompilator nie dopuści, by metr krawiecki zrobiono według wzorca tym konstruktorem kopiującym.

To już poważna rzecz: w rezultacie obiekty `const` nie mogą być używane do robienia z nich kopii. Ani do wysyłania ich do funkcji, ani do zwracania ich itd. Impas.

> Skoro nie możemy zabronić konstruktorowi modyfikacji – to pozostaje wyjście dyplomatyczne: pertraktować. To znaczy sam konstruktor powinien obiecać, że oryginału nie zmieni. To już znamy – konstruktor powinien odbierać argument jako referencję (jak do tej pory), ale powinien obiecać, że obiekt traktować będzie jako obiekt stały.

Jak się definiuje taki konstruktor? A co tu definiować? Dopisuje się w dotychczasowym konstruktorze kopiującym jedno słówko `const` i już gotowe!

```
kalibracja::kalibracja(const kalibracja & wzorzec)
```

To znaczy nie tyle, że sam obiekt wysyłany musi być koniecznie stały - (wysyłać będziemy przecież różne obiekty: czasem stałe, czasem nie) - jednak cokolwiek byśmy nie wysłali, to konstruktor gwarantuje nietykalność tego, co dostał.

14.8.3 Współodpowiedzialność

Niby problem rozwiązany, niestety nie zawsze da się tak zrobić. Wyobraź sobie, że to ty jesteś konstruktorem kopiującym. Kiedy możesz gwarantować nietykalność średniowiecznego starodruku, który powierzono Ci do skopiowania w domu? Wtedy, gdy odpowiadasz za siebie. Niestety – jeśli składniki Twojej klasy (Twoi domownicy) nie zagwarantują także nienaruszalności starodruku, to Twoje gwarancje nie wystarczą.

Przekładając to na język komputerowy: jeśli klasa ma składniki będące obiektami innych klas, to sposób nasz da się zastosować tylko wtedy, gdy konstruktory wewnętrznych obiektów można wywołać dla obiektów typu `const`. Czyli wtedy, gdy te konstruktory także zagwarantują nietykalność analogicznym składnikom wzorca. Oczywiście oznacza to, że i w tych konstruktorach kopiujących argumentami formalnymi muszą być referencje do obiektów stałych: `const`.

Pewnie pomyślałeś, że nie ma problemu - w razie czego wrócisz do odpowiednich miejsc w programie i uzupełnisz te kilka brakujących przydomków `const`

Wierz mi - zwykle to ogromna praca, bo każdy jeden dopisany `const` pociąga za sobą konieczność umieszczenia kilku dalszych w innych miejscach. To jest jak reakcja lawinowa. Nawet jeśli się uprzesz i postanowisz mozolnie uzupełnić wszystko co trzeba, to może się okazać, że po paru godzinach dojdziesz do funkcji, przy której powinieneś postawić `const`, a wiesz, że ona być `const` nie może.

Jakie jest wyjście? Tylko jedno: nigdy nie robić tego od tyłu. Jeśli tylko zaczynasz pisać poważny program, to od razu stawiaj `const` w takich miejscach, jak konstruktory kopiujące.

14.8.4 Konstruktor kopiujący generowany automatycznie

W linijce ❶❻ ostatniego przykładu – wewnątrz konstruktora kopiującego dokonywaliśmy w kopii zamiany tekstu. Oczywiście moglibyśmy zamieniać cokolwiek innego, tekst był tutaj przykładem.

Co by było, gdybyśmy nie podmieniali nic, czyli gdybyśmy chcieli robić naprawdę wierne kopie, a nie wariacje na temat oryginału?

- – Po pierwsze nasz przykład nie byłby tak pouczający.
- – Po drugie niepotrzebna byłaby ta cała praca. Bowiem jeśli sami nie zdefiniujemy konstruktora kopiującego, to kompilator postara się go wygenerować automatycznie. Kopiowanie odbywa się wówczas według zasady „składnik po składniku" (ang. memberwise copy) – czyli otrzymalibyśmy obiekt, który byłby idealną kopią naszego wzorca. W takiej sytuacji definiowanie konstruktora nie byłoby więc konieczne.

Wniosek: do identycznych kopii wystarczy konstruktor kopiujący generowany automatycznie.

14.8.5 Kiedy konstruktor kopiujący jest niezbędny?

Wiemy już, że za pomocą generowanego automatycznie konstruktora kopiującego, przy inicjalizacji nowego obiektu obiektem wzorcowym, powstaje dokładna kopia obiektu wzorcowego.

klasa obiekt_nowy = obiekt_istniejący ;

Są jednak sytuacje, kiedy taka dosłowna kopia byłaby wręcz niepożądana. Do tego stopnia, że mogłoby to mieć skutki fatalne.

Oto przykład (negatywny):

```
#include <iostream.h>
#include <string.h>
//////////////////////////////////////////////////////////
class wizytowka {
public :
     char *nazw ;                                        // ❶
     char *imie ;

     // konstruktor
     wizytowka(char * na, char * im) ;
     // destruktor
     wizytowka::~wizytowka() ;

     void personalia() {
          cout << imie << " " << nazw << endl ;
     }
     //-----------
     void zmiana_nazwiska(char *nowe)
```

```
     {
          strcpy(nazw, nowe);
     }
} ;
//////////////////////////////////////////////////////////
// definicja konstruktora
wizytowka::wizytowka(char *im, char *na)
{
     nazw = new char [80] ;                          // ❷
     strcpy(nazw, na) ;                              // ❸

     imie = new char [80] ;
     strcpy(imie, im) ;
}
/*********************************************************/
// ------------ definicja destruktora ----
wizytowka::~wizytowka()
{
     delete nazw ;                                   // ❹
     delete imie ;
}
/*********************************************************/
main()
{
wizytowka fizyk( "Albert", "Einstein") ;             // ❺
wizytowka kolega = fizyk ;                           // ❻

     cout << "Po utworzeniu blizniaczego obiektu oba "
              "zawieraja nazwiska\n" ;

     fizyk.personalia();                             // ❼
     kolega.personalia();

     // mój kolega nazywa się naprawdę Albert Metz

     kolega.zmiana_nazwiska("Metz");                 // ❽

     cout << "\nPo zmianie nazwiska kolegi brzmi ono : ";
     kolega.personalia();                            // ❾

     cout << "Tymczasem niemodyfikowany fizyk"
              " nazywa sie : " ;
     fizyk.personalia();                             // ❿

}
```

Po wykonaniu programu na ekranie zobaczymy

```
Po utworzeniu blizniaczego obiektu oba zawieraja
nazwiska
Albert Einstein
Albert Einstein

Po zmianie nazwiska kolegi brzmi ono : Albert Metz
Tymczasem niemodyfikowany fizyk nazywa sie : Albert Metz
```

Komentarz

❶ Klasa `wizytowka` służy do przechowywania nazwiska i imienia osoby. Jej składnikami nie są tablice znakowe, lecz wskaźniki do takich tablic.

Funkcjami składowymi tej klasy (oprócz konstruktora i destruktora) są:

- – funkcja `personalia`, która wypisuje na ekranie imię i nazwisko przypisane danemu obiektowi,
- – funkcja `zmiana_nazwiska`, która zmienia nazwisko wedle życzenia.

❷ Definicja konstruktora. Widzimy tutaj, że za pomocą operatora `new` rezerwujemy w dostępnym zapasie pamięci tablicę na 80 znaków. Wskaźnik `nazw` ustawiamy tak, by pokazywał na tę tablicę.

❸ Do tablicy kopiujemy przysłany nam jako argument string z nazwiskiem. Poniżej robimy podobnie z imieniem.

❹ Działanie destruktora polega na zwolnieniu rezerwacji pamięci. Dostępnemu zapasowi pamięci oddajemy z powrotem te obszary, na które pokazują wskaźniki `nazw` i `imie`.

❺ Definicja obiektu klasy `wizytowka`. Ta konkretna wizytówka nazywa się `fizyk` i konstruowana jest za pomocą zwykłego konstruktora.

❻ Definicja obiektu klasy wizytówka połączona ze skopiowaniem wizytówki fizyka. Ten znak równości w definicji mówi nam, że odbywa się to za pomocą konstruktora kopiującego. Takiego konstruktora nie definiowaliśmy, więc kompilator wygenerował go sam. Daje on w rezultacie absolutnie wierną kopię obiektu `fizyk`.

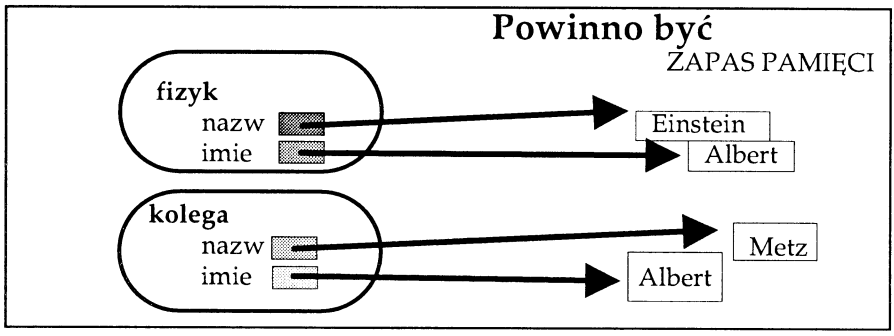
❼ Na dowód, że tak jest w istocie – wypisujemy na ekran treść obu wizytówek.

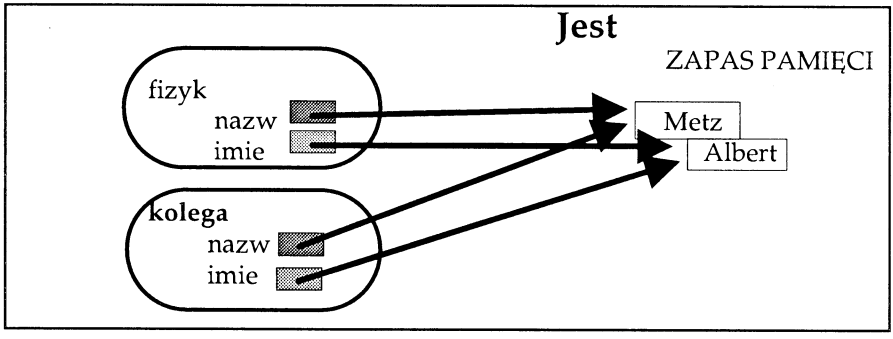
❽ Uruchamiamy funkcję `zmiana_nazwiska` na rzecz obiektu `kolega`, po czym wypisujemy na ekranie personalia z tego obiektu ❾. Widzimy wyraźnie, że zmiana nastąpiła.

❿ Przypadkiem wypisujemy też personalia `fizyka`. I tu niespodzianka – nic z tym obiektem nie robiliśmy, a zmienił on także nazwisko. Dlaczego?

To dlatego, że składnikami klasy były nie tablice, a wskaźniki do tablic. Kopiując „składnik po składniku" mieliśmy wierną kopię wskaźników, które pokazywały na to samo miejsce w pamięci.

Schematycznie można to pokazać tak, jak na zamieszczonych poniżej dwóch rysunkach.





Jeśli z rysunków rozumiesz co się stało, to nie musisz czytać dokładnie wytłumaczenia.

Wytłumaczenie

Obiekt `kolega` został utworzony metodą kopiowania składnik po składniku. Do składników obiektu `kolega` zostały przepisane więc adresy tablic, które obiekt `fizyk` zarezerwował **sobie**, na swój użytek. Nie była kopiowana żadna treść tablic. Dziwi Cię to? Przecież obiekty są kopiowane składnik po składniku – a tablice nie były składnikami. Składnikami są tylko dwa wskaźniki. Tablice są tylko przez obiekt dodatkowo wydzierżawione od zapasu pamięci.

Na skutek tego mamy dwa obiekty, ale tylko jeden zestaw tablic.

- – `fizyk` ma swoje własne tablice, bo je sobie sam wytworzył operatorami `new`
- – `kolega` ma dwa wskaźniki pokazujące na te tablice, które fizyk założył dla siebie. Błąd więc polegał na tym, że dla `kolegi` nie założono żadnych tablic. Kolega pasożytuje na `fizyku`. Zmiana nazwiska `kolegi` nastąpiła, więc przez wpisanie nowego nazwiska do tablic fizyka.

To jeszcze nie wszystko. Spójrz na destruktor. Jeśli jeden z tych obiektów będzie likwidowany (wszystko jedno który), to tablice zostaną zlikwidowane operatorem `delete`.

Drugi obiekt o tym nic nie będzie wiedział i może się zdarzyć, że wpisze coś do tego obszaru, który już jest oddany (może przydzielony komu innemu). Coś wtedy zniszczymy. Tragedia.

Dodatkowo, jeśli drugi obiekt będzie likwidowany, to także uruchomi operator `delete` w stosunku do obszaru, który już do niego nie należy. Pamiętasz chyba, że dwukrotne użycie `delete` w stosunku do tego samego wskaźnika ma skutek fatalny. Co się zdarzy – zależy to od implementacji.

Kto jest winny? My sami

Pamiętajmy, że generowany automatycznie konstruktor kopiujący wykonuje kopię „składnik po składniku". Powinniśmy sobie zadać pytanie, czy to nam odpowiada. Jeśli nie odpowiada – to definiujemy własny konstruktor kopiujący, który skopiuje tak, jak sobie tego zażyczymy. W naszym wypadku – zakładając osobny zestaw tablic.

Oto realizacja takiego konstruktora:

```
wizytowka::wizytowka(wizytowka &wzor)
{
     nazw = new char [80] ;
     strcpy(nazw, wzor.nazw);

     imie = new char [80] ;
     strcpy(imie, wzor.imie);
}
```

Nasz ostatni program wyposażony w taki konstruktor kopiujący zachowa się już poprawnie.

Spójrz zresztą na ekran

```
Po utworzeniu blizniaczego obiektu oba zawieraja
nazwiska
Albert Einstein
Albert Einstein

Po zmianie nazwiska kolegi brzmi ono : Albert Metz
Tymczasem niemodyfikowany fizyk nazywa sie : Albert Einstein
```

Nie tylko dla wtajemniczonych:

Klasa, w której jest
- destruktor

lub
- konstruktor kopiujący

lub
- przeładowany operator przypisania,

najczęściej wymaga istnienia ich wszystkich trzech

15 Tablice obiektów

Jak wiemy, w języku C++ można tworzyć tablice z obiektów typów wbudowanych (np. `int`, `float`), a nawet ze wskaźników do tych obiektów.

```
int  tablica[30] ;        // 30 elementowa tablica obiektów int
float mmm[12] ;           // 12 elementowa tablica obiektów float
char *tablwsk[5] ;        // 5 elementowa tablica wskaźników do char
```

Analogicznie da się tworzyć tablice obiektów jakiejś klasy (czyli tablice obiektów typu zdefiniowanego przez użytkownika).

Oto przykład. Najpierw wymyślmy sobie klasę

```
class stacje_metra {
public:
     float km ;                    // na którym kilometrze trasy
     int glebokosc ;
     char nazwa[40] ;
     char przesiadki[80];
} ;
```

Dla prostoty nie ma w niej funkcji składowych, a składniki-dane są publiczne.

Tablicę obiektów tej klasy definiujemy

```
stacje_metra stacyjka[15] ;
```

Definicję tę czytamy tak: `stacyjka` jest 15 elementową tablicą obiektów klasy `stacje_metra`. Poszczególne obiekty tej tablicy to oczywiście

```
stacyjka[0]
stacyjka[1]
...itd...
stacyjka[14]
```

Ponieważ wszystkie składniki tej klasy są tu akurat publiczne, więc możemy się do nich odnosić bezpośrednio (bez pośrednictwa funkcji składowych) stosując znaną już notację obiekt.składnik czyli

```
stacyjka[4].glebokosc
stacyjka[9].km
```

Zdefiniujmy sobie wskaźnik, który może pokazywać na takie elementy

```
stacje_metra * wsk ;
```

Czytamy: `wsk` – jest wskaźnikiem mogącym pokazywać na obiekty typu (klasy) `stacje_metra`.
Wskaźnik ten można ustawić tak, by pokazywał na którąś konkretną stację

```
wsk = & stacyjka[9] ;
```

Wykonanie operacji

```
wsk++ ;
```

sprawi, że wskaźnik będzie pokazywał na stację następną. A oto jak za pomocą wskaźnika odnosić się do składników klasy

```
wsk->km
```

Jeśli wskaźnik pokazuje właśnie na obiekt `stacyjka[9]`, to wyrażenie powyższe oznacza odniesienie się do składnika `km` w obiekcie `stacyjka[9]`.

> Posługiwanie się tablicami obiektów bardzo upraszcza programowanie. Sama notacja zaś – dla nowicjusza – może wydać się trochę skomplikowana. Wystarczy jednak uświadomić sobie, że obiekt teraz w nazwie ma swój numer.

Dla porównania utwórzmy parę obiektów klasy `stacje_metra`, ale nie róbmy z nich tablicy

```
stacje_metra a, b, c ;
```

Porównaj teraz zapis odniesienia się do składników w wypadku obiektów z tablicy i – że tak powiem – wolnostojących

```
a.glebokosc              stacyjka[0].glebokosc
b.km                     stacyjka[1].km
c.nazwa                  stacyjka[2].nazwa
a.przesiadki             stacyjka[0].przesiadki
```

Kiedy nasza tablica obiektów może się przydać?

Oto przykład. W wagonie metra, w czasie jazdy, wyświetlane są dane o następnej stacji: nazwa i ewentualne przesiadki. Na trasie linii jest 15 stacji. W trakcie jazdy ich opisy muszą się kolejno pojawić na wyświetlaczu. Dzięki temu, że kolejne stacje ułożone są w tablicę - można się po prostu posłużyć pętlą.

```
for(int i = 0 ; i < 15 ; i++)
{
     cout << "Stacja : " << stacyjka[i].nazwa << endl;

     if(stacyjka[i].przesiadki != NULL)
     {
          cout << "Przesiadki :"
               << stacyjka[i].przesiadki
               << endl ;
```

```
        }
    }
```

Nie pytaj mnie skąd w elementach tablicy wzięły się dane. O tym pomówimy za chwilę. Najpierw jednak inny sposób definiowania tablic.

15.1 Tablica obiektów definiowana operatorem `new`

Podobnie jak zwykłe tablice, tak i tablice obiektów danej klasy można zdefiniować w dostępnym **zapasie pamięci** (free store). Oto porównanie: Definiujemy w zapasie pamięci tablicę elementów typu `int` oraz elementów klasy `stacje_metra`. Jak pamiętamy, taka tablica jest zawsze bezimienna, ale to nie szkodzi – i tak posługujemy się wskaźnikami.

```
int *wskint ;
     wskint = new int[100] ;

stacje_metra *wsk_sta ;
     wsk_sta = new stacje_metra[15] ;
```

Dalsza praca na tak zdefiniowanych tablicach wygląda tak samo, jakbyśmy mieli wskaźnik do zwykłej tablicy. Pamiętamy, że posługując się wskaźnikiem możemy stosować w takim wypadku dwa typy zapisów: wskaźnikowy i tablicowy. Odniesienie się elementów o indeksie 8 można więc wyrazić jako

```
*(wskint + 8)          // notacja „wskaźnikowa"
wskint[8]              // notacja „tablicowa"

*(wsk_sta + 8)         // notacja „wskaźnikowa"
wsk_sta[8]             // notacja „tablicowa"
```

Wiem, jestem nudny, ale jeszcze raz przypomnę, że element o indeksie 8 to dziewiąty element tablicy (numerujemy przecież od 0)

Jeśli chcielibyśmy sięgnąć do składnika `przesiadki` w tym elemencie, to stosujemy jeden z poniższych zapisów

```
(*(wsk_sta + 8)).przesiadki      // jakby: obiekt.składnik
  (wsk_sta + 8)->przesiadki      // jakby: wskaźnik->składnik
   wsk_sta[8].przesiadki         // jakby: wskaźnik[8].składnik
```

Przypominam, że wyrażenie `(wsk_sta+8)`, jako całość, jest adresem czegoś, co jest o osiem elementów dalej niż to, na co pokazuje wskaźnik `wsk_sta`. Jako całość więc jest też czymś w rodzaju wskaźnika. Stąd wyrażenie `(*(wsk_sta + 8))` jest odniesieniem się do obiektu tak właśnie pokazywanego.

Trzeci zapis jest, jak powiedziałem, możliwy mimo, że wskaźnik nie jest nazwą tablicy. Wynika to z podobnego traktowania w języku C++ wskaźników i tablic. Przypominam: nazwa tablicy, to jakby **stały wskaźnik** pokazujący na zerowy element tablicy.

Tej klauzuli o stałości nie ma nasz wskaźnik, zatem jeśli go przesuniemy tak, żeby pokazywał na element o indeksie 4, wówczas zapis `wsk_sta[8]` będzie odniesieniem się do elementu o indeksie 12. (4+8 = 12)

Ostrzegam jednak przed zmienianiem wskaźnika pokazującego na tablicę w zapasie pamięci. Pamiętaj, że w momencie, gdy będziesz chciał tablicę kasować operatorem `delete` – musisz mieć wskaźnik pokazujący na początek tej tablicy. W dodatku przy żonglowaniu wskaźnikiem możemy stracić kontakt z tablicą na zawsze. (*Vide* historyjka o baloniku zerwanym z nitki).

Aby się przed takimi ewentualnościami uchronić mam pewien sposób. Polega on na tym, że w takiej sytuacji mam co najmniej dwa wskaźniki pokazujące na tę tablicę. Jednego z nich nigdy nie zmieniam i pokazuje on zawsze na początek tablicy. Żeby się upewnić definiuję go po prostu jako `const`.

Oto taki wskaźnik

```
stacje_metra * const stac ;
```

Definicję czytamy – `stac` jest stałym (`const`) wskaźnikiem `*` do obiektów klasy `stacje_metra`.

Kasowanie tablicy, którą zarezerwowaliśmy w zapasie pamięci, jest również podobne jak w wypadku tablic typów wbudowanych.

```
delete [] wskint ;
delete [] wsk_sta ;
```

Jeśli mimo moich ostrzeżeń przesunąłeś wskaźnik i nie pokazuje on już na początek tablicy – to skutek użycia takiego wskaźnika dla operacji `delete` może być fatalny.

15.2 Inicjalizacja tablic obiektów

Zwracam uwagę, że mówić tu będziemy o inicjalizacji czyli o nadawaniu wartości początkowej w momencie definicji obiektu (narodzin).[†)]

W wypadku inicjalizacji tablic obiektów danej klasy mogą nastąpić takie sytuacje:

- – inicjalizowana jest tablica, która jest agregatem,
- – inicjalizowana jest tablica, która nie jest agregatem,
- – inicjalizowana jest tablica zdefiniowana w zapasie pamięci.

Przyjrzyjmy się poszczególnym sytuacjom.

15.2.1 Inicjalizacja tablic obiektów będących agregatami

Z pojęciem agregatu już się spotkaliśmy (w rozdz. o tablicach str. 132). Tutaj przyjrzymy mu się dokładniej.

†) Natomiast nadawanie wartości obiektowi już wcześniej istniejącemu nazywamy przypisaniem (czyli inaczej: podstawieniem).

Agregatem – (czyli skupiskiem danych) jest

tablica obiektów klasy K lub obiekt klasy K,

gdy owa klasa K:

- – nie ma składników danych prywatnych lub zastrzeżonych (czyli `private` lub `protected`),
- – nie ma konstruktorów,

ani: (o czym pomówimy w następnych rozdziałach)

- – nie ma klas podstawowych,
- – nie ma funkcji wirtualnych.

Przypomnij sobie naszą ostatnią klasę

```
class stacje_metra {
public:
     float km ;
     int glebokosc ;
     char nazwa[40] ;
     char przesiadki[80];
} ;
```

Jak widzimy, klasa ta nie ma składników danych chronionych słowami `private` lub `protected`. Nie ma też konstruktora.

Musisz mi teraz uwierzyć, że nie ma także klas podstawowych (bo ich nazwy musiałyby wystąpić przy samej nazwie klasy), oraz że nie ma funkcji wirtualnych (przy jakiejś funkcji składowej stałoby wówczas słowo `virtual`).

Nasza klasa jest więc agregatem (czyli inaczej mówiąc: skupiskiem danych).

Otóż:

> Agregat taki można inicjalizować za pomocą listy inicjalizatorów ograniczonej znakami { }. Poszczególne elementy tej listy oddzielone są od siebie przecinkami†).

Pokażmy na przykładzie. Najpierw inicjalizacja jednego obiektu

```
stacje_metra moja_stacja = { 14.3, -6,
                             "Yorckstrasse", "S1 Wansee" } ;
```

Widzimy tu definicję obiektu o nazwie `moja_stacja` będącego obiektem klasy `stacje_metra`. Składniki tego obiektu są inicjalizowane za pomocą listy inicjalizatorów wartościami:

– składnik `km` wartością 14.3

– składnik `glebokosc` wartością -6

– składnik `nazwa` stringiem "Yorckstrasse" (to nazwa ulicy)

---

†) Nie myl nazwy *lista inicjalizatorów {...}* z *listą inicjalizacyjną konstruktora*, stojącą przy definicji konstruktora (po dwukropku). Co prawda nazwy brzmią podobnie, jednak nie sądzę, by językowo mocniejsze odróżnienie tych nazw było godne zachodu.

– składnik `przesiadki` stringiem "S1 Wansee" (nazwa kolejki, na którą można się przesiąść).

Oczywiście nadanie tej samej treści owemu obiektowi można by wykonać takim zapisem:

```
stacje_metra moja_stacja ;          // definicja obiektu
moja_stacja.km = 14.3 ;
moja_stacja.glebokosc = -6 ;
strcpy(moja_stacja.nazwa, "Yorckstrasse");
strcpy(moja_stacja.przesiadki, "S1 Wansee");
```

Byłoby to już jednak nie inicjalizacja, ale przypisanie.

Przyznasz jednak, że ten sposób z listą inicjalizatorów jest krótszy i wygodniejszy w zapisie. Nie zawsze jednak można użyć obu form. Jeśli definiujemy sobie obiekt jako stały

```
const stacje_metra centralna { ... } ;
```

albo gdy jeden ze składników danych w klasie był typu `const`

```
// ...
const int glebokosc ;
// ...
```

to nadać wartość początkową takiemu obiektowi można tylko w momencie jego narodzin, czyli podczas definicji obiektu.

Mieliśmy jednak mówić o tablicach.

Jak inicjalizować tablicę, której elementy są agregatami

(- a więc sama tablica też jest agregatem). Oczywiście już na pewno się domyśliłeś

```
stacje_metra stacyjka [15] =
{    0, 4, "ZOO", "S3, U1, U9",          // dane dla stacyjki[0]
     1.7, 4, "Tiergarten", "",           // dane dla stacyjki[1]
     3, 3, "Bellevue", ""                // dane dla stacyjki[2]
} ;
```

Jak widzimy dane dla poszczególnych elementów tablicy są podawane kolejno.

> Na liście inicjalizatorów nie może być więcej danych niż elementów tablicy. Może być jednak mniej.

Tak jest właśnie w naszym przypadku: zdefiniowaliśmy tablicę 15 elementów, a danymi wypełniamy tylko pierwsze 3.

> Zapamiętaj: reszta wypełniana jest w takim wypadku zerami stosownego typu. Co to znaczy dla składnika `int`, `float` - wiadomo. Dla składnika, który jest stringiem oznacza NULL – czyli, że string jest pusty i nie zawiera żadnego tekstu.

Uwaga dla programistów C

Znane Ci z klasycznego C struktury `struct` są oczywiście także rozumiane w C++ jako szczególny rodzaj klasy. W języku C struktury mogły być inicjalizowane podobną listą inicjalizatorów.

Zapytasz pewnie – Jeśli mam program w języku C, gdzie korzystam ze struktur inicjalizowanych listą inicjalizatorów, to czy taki program da się bezbłędnie skompilować w C++ ?

Tak. Albowiem struktura z klasycznego C jest agregatem w rozumieniu C++

- – nie ma przecież składników `private`, `protected` - bowiem struktura jest to klasa, która przez domniemanie ma wszystkie składniki `public`,
- – nie ma ani konstruktorów, ani klas podstawowych, ani funkcji wirtualnych- po prostu dlatego, że te rzeczy w klasycznym C nie istnieją.

15.2.2 Inicjalizacja tablic nie będących agregatami

Jeśli obiekt, albo tablica obiektów nie jest agregatem, to inicjalizować ich za pomocą zwykłej listy inicjalizatorów nie można. Chociażby dlatego, że jeśli jakiś składnik jest `private` – to tym samym jest niedostępny spoza klasy. Lista inicjalizatorów nie leży przecież w zakresie ważności klasy.

Wyjściem jest wtedy posłużenie się konstruktorem. Jeśli mamy tablicę składającą się z takich obiektów, to na liście inicjalizatorów umieszczamy konstruktory dla poszczególnych elementów tablicy.

Oto przykład, w którym inicjalizujemy pojedynczy obiekt, a także tablicę obiektów. Jest tu definicja klasy, która na pewno nie jest agregatem - ma składniki prywatne, a także konstruktory.

```
#include <iostream.h>
#include <string.h>
//////////////////////////////////////////////////////////
class stacje_metra2 {
     float km ;          // na którym kilometrze trasy
     int glebokosc ;
     char nazwa[40] ;
     char przesiadki[80];
public:
     // ---------- konstruktor                                  // ❶
     stacje_metra2(float kk, int gg, char *nn,
                                        char *pp = "") ;
     // ---------- konstruktor domniemany
     stacje_metra2() ;
     // ---------- zwykła funkcja składowa
     void gdzie_jestesmy() ;

} ;
//////////////////////////////////////////////////////////
stacje_metra2::stacje_metra2(float kk, int gg,
                                  char *nn, char *pp)
               : km(kk), glebokosc(gg)                      // ❷
```

```
{
     strcpy(nazwa, nn);
     strcpy(przesiadki, pp) ;
}
/*****************************************************/
stacje_metra2::stacje_metra2()
                                        // konstruktor domniemany      ❸
{
     km = 0 ;
     glebokosc = 0 ;
     strcpy(nazwa, "Nie nazwana jeszcze" );
     przesiadki[0] = NULL ;
}
/*****************************************************/
void stacje_metra2::gdzie_jestesmy()                          // ❹
{
     cout << "Stacja : " << nazwa << endl ;

     if(przesiadki[0])   // to samo co:
                         // if(przesiadki[0] != NULL)
     {
          cout << "\tPrzesiadki : " << przesiadki << endl;
     }
}
/*****************************************************/
main()
{                                                             // ❺
stacje_metra2 ostatnia =
               stacje_metra2 (22, 0, "Wansee", "118 Bus" );

     ostatnia.gdzie_jestesmy() ;
     cout << "********************\n" ;

const int ile_stacji = 7 ;
stacje_metra2 przystanek[ile_stacji] =                        // ❻
     {
     stacje_metra2 (0, 5, "Fredrichstrasse", "Linia U6"),
     stacje_metra2 (),
     stacje_metra2 (),
     stacje_metra2 (5.7, 4, "Tiergarten"),
     stacje_metra2 (8, 4, "ZOO", "Linie U1 i U9")
     } ;

     for(int i = 0 ; i < ile_stacji ; i++)
     {
          przystanek[i].gdzie_jestesmy() ;
     }
}
```

Na ekranie po wykonaniu tego programu zobaczymy

```
Stacja : Wansee
     Przesiadki : 118 Bus
********************
Stacja : Fredrichstrasse
     Przesiadki : Linia U6
```

```
Stacja : Nie nazwana jeszcze
Stacja : Nie nazwana jeszcze
Stacja : Tiergarten
Stacja : ZOO
     Przesiadki : Linie U1 i U9
Stacja : Nie nazwana jeszcze
Stacja : Nie nazwana jeszcze
```

Komentarze

❶ Deklaracja konstruktora. Zauważ, że ostatni argument jest domniemany- jest to pusty string.

❷ To definicja tego konstruktora. W liście inicjalizacyjnej – tej za dwukropkiem – widzimy inicjalizację składników `km` oraz `glebokosc`. Ponieważ żaden z nich nie jest `const`, więc równie dobrze można było nadać im wartość początkową przez przypisanie - już w ciele konstruktora.

W ciele konstruktora widzimy akcję przypisania stringów do tablic.

❸ Klasa ma też konstruktor domniemany — czyli wywoływany bez żadnych argumentów. Tutaj, dla odmiany, dane do składników `km` oraz `glebokosc`, wpisywane są w ciele konstruktora. Zamiast nazwy stacji wpisujemy odpowiedni tekst, a zamiast informacji o przesiadkach, wpisujemy do zerowego elementu tablicy znak NULL czyli 0. Będzie to po prostu pusty string.

❹ Funkcja składowa. Mogła ona być już w poprzedniej klasie. Jej obecność lub nieobecność nie ma wpływu na fakt czy klasa jest agregatem czy nie. O ile jednak tam mogłem obejść się bez funkcji składowych, bo wszystkie składniki-dane były publiczne, o tyle tutaj tak się nie da. Pracować na składnikach prywatnych można tylko za pomocą funkcji składowych (oraz ewentualnie zaprzyjaźnionych).

❺ Definicja pojedynczego obiektu klasy `stacja_metra2`. Widzimy, że po drugiej stronie znaku '=' stoi wywołanie konstruktora z argumentami.

❻ Definicja siedmioelementowej tablicy obiektów klasy `stacja_metra2`. Zauważ listę inicjalizatorów `{...}`

Jak widać – teraz zamiast grupy 4 danych (`km`, `glebokosc`, `nazwa`, `przesiadki`) – dla poszczególnych stacji mamy wywołanie konstruktora. Na liście inicjalizatorów takie wywołania konstruktorów są oddzielone przecinkami. Są to wywołania konstruktorów dla poszczególnych elementów tablicy. Jak należy to rozumieć?

Otóż – pamiętasz chyba, jak mówiliśmy o sytuacji, kiedy konstruktor wywoływany jest jawnie. W rezultacie takiego wywołania mamy obiekt chwilowy, który służy do inicjalizacji danego obiektu tablicy. Inicjalizacja odbywa się, jak wiadomo, konstruktorem kopiującym.

Jeśli jest to zawiłe, to przeanalizujmy sytuację powoli. Na liście widzimy najpierw konstruktor wywołany dla stacji Friedrichstrasse. W jego wyniku tworzy się obiekt chwilowy zawierający dane o tej stacji. Takim obiektem inicjalizowany jest element tablicy `przystanek[0]`.

Następnie z listy inicjalizatorów brany jest kolejny konstruktor i w wyniku jego działania powstaje inny obiekt chwilowy – nim inicjalizowany jest `przystanek[1]`.

Przypominam, że inicjalizacja jakiegoś obiektu innym obiektem odbywa się:

- 1) albo przez generowany automatycznie konstruktor kopiujący, czyli metodą „składnik po składniku",
- 2) albo za pomocą konstruktora kopiującego, który dostarczymy my sami.

Tutaj – ponieważ nie dostarczyliśmy swojego – odbędzie się to „składnik po składniku". Nie ma tu żadnego ryzyka w stosunku do tej metody, bo w klasie nie ma składników będących wskaźnikami.

Zapytasz pewnie: „–W naszym przykładzie tablica ma 7 elementów, a na liście inicjalizatorów umieściliśmy wywołania tylko pięciu konstruktorów. Co z resztą?"

Dla pozostałych elementów tablicy **niejawnie** wywoływany jest konstruktor domniemany. Jest więc bardzo ważne, żeby – jeśli przewidujemy taką sytuację – wyposażyć klasę w konstruktor domniemany.

Co by było gdybyśmy nie mieli konstruktora domniemanego? Kompilator sygnalizowałby błąd. Musiałbyś wówczas zamieścić na liście inicjalizatorów {...} wywołania konstruktorów dla wszystkich elementów tablicy tak, żeby lista była kompletna.

15.2.3 Inicjalizacja tablic tworzonych w zapasie pamięci

Tablice, które tworzone są w zapasie pamięci za pomocą operatora `new`, nie mogą mieć jawnie wypisanej inicjalizacji. Takie tablice są możliwe do wykreowania tylko wtedy, gdy:

- klasa nie ma **żadnego** konstruktora,
- klasa wśród swoich konstruktorów ma konstruktor domniemany.

Właśnie konstruktor domniemany zajmuje się inicjalizacją takiej tablicy.

W naszej klasie `stacja_metra2` mieliśmy konstruktor domniemany, więc możemy zrobić w zapasie pamięci tablicę

```
stacja_metra2 * wsk ;
     wsk = new stacja_metra2[8] ;
```

Powstaje 8 elementowa tablica, a do inicjalizacji wszystkich 8 elementów został użyty konstruktor domniemany

```
stacja_metra2::stacja_metra2(void) ;
```

Co by było, gdybyśmy konstruktora domniemanego dla tej klasy nie zdefiniowali?

Byłby błąd kompilacji, bo w klasie są inne konstruktory, a wśród nich nie ma domniemanego. Gdybyśmy jednak z klasy usunęli wszystkie konstruktory wtedy błędu nie ma. Wszystkie 8 elementów tej tablicy inicjalizowane byłoby

konstruktorem domniemanym automatycznie wygenerowanym przez kompilator.

(Pamiętasz, mówiliśmy kiedyś, że w razie potrzeby konstruktor domniemany, jak i konstruktor kopiujący – mogą być generowane automatyczne).

Co wówczas zostanie wpisane do elementów tej tablicy? Oczywiście stosownego rodzaju zera.

Zapamiętaj:

Jeśli zamierzasz z obiektów danej klasy tworzyć w zapasie pamięci tablice (operatorem `new`) to ta klasa:

— albo nie może mieć żadnych konstruktorów,
— albo musi mieć konstruktor domniemany.

16 Wskaźnik do składników klasy

W rozdziale tym mówić będziemy o szczególnego rodzaju wskaźnikach. Takich, które służą do pokazywania na składniki wewnątrz danej klasy. Rozdział ten dotyczy rzeczy bardzo specyficznej, dlatego jeśli tylko uznasz, że jest za trudny, lub że Cię nudzi - przeskocz go i przejdź do następnego (o konwersjach). Wrócisz tu kiedy indziej. Podane tu informacje nie są niezbędne do zrozumienia następnych rozdziałów. Dlatego przy pierwszym czytaniu książki proponuję ten rozdział opuścić.

Pierwotne wersje języka C++ nie pozwalały na definiowanie wskaźników do składników klasy. Obecnie jest to już możliwe. Zanim jednak zaczniemy mówić o tych wskaźnikach, przypomnijmy sobie co wiemy o wskaźnikach zwykłych.

16.1 Wskaźniki zwykłe - repetytorium

Z poprzednich rozdziałów wiemy, że na różne obiekty tej samej klasy (lub tego samego typu) można pokazywać wskaźnikiem. Przypomnijmy sobie. Mamy klasę K

```
class K {
     // ...
public :
     int skladniczek ;
     // ...
} ;
```

Oto dwie definicje wskaźników. Jeden nadaje się do pokazywania na obiekty typu `int`, a drugi na obiekty klasy `K`

```
int *wskazint ;                    // do typu wbudowanego int
K *wskazobiekt ;                   // do typu zdefiniowanego K
```

Tę drugą definicję czytamy: `wskazobiekt` jest wskaźnikiem mogącym pokazywać na obiekty klasy `K`

Oto definicja kilku obiektów klasy `K`

```
K obiekt_duzy, obiekt_zielony ;
```

A tak ustawia się wskaźnik, by pokazywał na któryś z nich

```
wskazobiekt = & obiekt_zielony ;
```

Jeśli chcemy odnieść się do publicznego składnika obiektu, na który pokazuje wskaźnik, to piszemy

```
wskazobiekt -> skladniczek ;
```

Do składnika niepublicznego nie można się tak z zewnątrz klasy odnieść. Dodatkowo zwracam uwagę, że sam wskaźnik pokazuje tu na obiekt, a nie na składnik wewnątrz obiektu.

Wskaźnikami często posługujemy się przy pokazywaniu na różne elementy zgrupowane w tablicę. Oto definicja tablicy obiektów klasy `K`

```
K    tab[10] ;
```

Definicję tę czytamy: `tab` jest 10 elementową tablicą obiektów klasy `K`. Ustawienie naszego wskaźnika na element takiej tablicy może wyglądać następująco

```
wskazobiekt = & tab[6] ;
```

Czy można zwykłym wskaźnikiem pokazać na coś, co jest we wnętrzu obiektu?

Tak. Wskaźnikiem zwykłym możemy pokazać też na coś co jest we wnętrzu obiektu pod warunkiem, że będzie to składnik publiczny. Na przykład - jeśli wewnątrz klasy jest publiczny składnik typu `int`, to do pokazania na niego wystarcza zwykły wskaźnik typu `int`. Ustawiając taki wskaźnik wystarczy pamiętać, że wyrażenie

```
(obiekt_zielony.skladniczek)
```

jako całość jest typu `int` i dlatego można ustalić jego adres

```
int *wskint ;
wskint = & (obiekt_zielony.skladniczek) ;
```

wpisanie liczby 55 do tego składnika można teraz wykonać dwojako

```
obiekt_zielony.skladniczek = 55 ;
*wskint = 55 ;
```

Wskaźnik typu `int*` pokazuje na daną składową typu `int`, ale równie dobrze może za chwilę pokazać na zwykły obiekty typu `int`

```
int m ;
     wskint = &m ;
```

A co by było, gdyby składnik, na który chcemy pokazać był prywatny? Wówczas nie dałoby się ustawić wskaźnika na nim. To dlatego, że przecież wówczas wyrażenie

```
obiekt_zielony.skladniczek
```

jest nielegalne. Kompilator nie dopuści do tego, by odnosić się do składnika, który jest prywatny.

Skoro przypomnieliśmy sobie wiadomości o zwykłych wskaźnikach – przejdźmy do *meritum*.

16.2 Wskaźnik do pokazywania na składnik-daną

Zapytasz pewnie - Po co ten paragraf skoro sprawa jest już rozwiązana ? Wiemy już jak ustawić wskaźnik na publicznym składniku klasy. Rzeczywiście. Tu jednak pomówimy o nieco inteligentniejszych wskaźnikach.

Tego rodzaju wskaźnik nazywa się wskaźnikiem do składnika klasy. Konkretniej: wskaźnikiem do pokazywania na niestatyczne (czyli zwykłe) składniki klasy.

Aby zrozumieć czym szczególnym jest ten wskaźnik posłużymy się takim obrazkiem:

Mamy dwóch kolegów Piotra i Tomasza. Obaj są specjalistami od wytrzymałości metali.

Piotra bierzemy na lotnisko Tempelhof, pokazujemy mu na coś palcem (wskaźnik) i mówimy: „Widzisz ten kawałek metalu? Zatem sprawdź czy jest on w porządku". Piotr jest fachowcem, więc sprawdza to, co mu kazaliśmy. Zresztą zawsze sprawdza kawałki metali, które mu podsuwamy do sprawdzenia. To, że jest to akurat cięgło steru kierunku z samolotu, nawet go nie interesuje. Niby jest to fachowiec, ale problem w tym, że za każdym razem musimy podejść z nim do tego kawałka metalu i pokazać mu go palcem (ustawić wskaźnik na konkretny składnik obiektu).

Mamy też drugiego kolegę: Tomasza. Ten jest inteligentniejszy. Jemu mówimy tak: „Spójrz tu na ścianę. Oto wisi tu rysunek techniczny samolotu »Concorde« (czyli definicja klasy samolotów »Concorde«). A teraz uważaj: na rysunku pokazuję ci składnik tej klasy samolotów: cięgło steru kierunku. Z tego, co pokazuję, widzisz jak to cięgło steru kierunku jest umieszczone w stosunku do innych składników samolotu tej klasy. Pokazuję ci jednak nie na rzeczywisty obiekt, ale na rysunek tego cięgła." (W ten sposób zdefiniowaliśmy wskaźnik).

Następnie mówimy do kolegi: „—W tym egzemplarzu samolotu »Concorde«, który przyleciał dziś rano z Paryża (wskaźnik do obiektu) masz sprawdzić cięgło steru kierunku. Tylko nie mów mi, że nie wiesz gdzie takie cięgło jest: pokazałem ci je właśnie na rysunku technicznym."

Kolega idzie i wykonuje swoją robotę.

Jeśli innym razem mamy znowu podobne zadanie, to mamy dwa wyjścia:

- 1) Wziąć za rękę kolegę Piotra i czołgając się pośród żelastwa we wnętrzu samolotu wreszcie pokazać mu palcem i powiedzieć: „Sprawdź ten kawałek metalu."
- 2) Napotkanego w barze kolegę Tomasza podprowadzić do okna i powiedzieć: „Widzisz ten samolot, który właśnie ląduje? Sprawdź w nim proszę, to cięgło, które ci pokazałem ostatnio na planie".

Jaka jest różnica między tymi dwoma sytuacjami? Otóż taka, że by poprosić o przysługę Piotra potrzebny nam był tylko jeden palec. Mówiliśmy: Napraw TO. Czyli tylko jeden wskaźnik.

W kontaktach z Tomaszem posługiwaliśmy się tak naprawdę dwoma wskaźnikami. Raz pokazywaliśmy coś specjalnym wskaźnikiem na rysunek techniczny, a potem pokazywaliśmy palcem na samolot. (Mogliśmy palcem na samolot nie pokazywać. Zamiast tego można użyć też nazwy tego konkretnego egzemplarza samolotu).

O ile palec dobrze się nadaje do pokazania na samoloty, o tyle do pokazania drobnego szczegółu na rysunku technicznym nie stosuje się palca. Pokazujemy tu nie na rzeczywistą rzecz, a raczej na coś abstrakcyjnego (jej wizerunek). Do pokazywania na rysunku technicznym na te nieszczęsne cięgła służy więc specjalny wskaźnik. Te wskaźniki będą właśnie przedmiotem tego rozdziału.

Ten specjalny wskaźnik pokazuje nie tyle na konkretny składnik, co na jego miejsce w deklaracji klasy.

Oto jak wygląda definicja wskaźnika mogącego pokazywać we wnętrzu obiektów danej klasy `K` na obiekty np. typu `int` :

```
int K::*wsk ;
```

Definicję tę czytamy

`wsk`	
`*`	jest wskaźnikiem
`K::`	do pokazywania we wnętrzu klasy `K`
`int`	na obiekty typu `int`

Czym naprawdę różni się ten wskaźnik od zwykłego wskaźnika?

Tym, że zwykły wskaźnik pokazuje na konkretne miejsce w pamięci o jakimś absolutnym adresie. Np. na komórkę 783492614. Tam ma być dokładnie szukana zmienna `int`, albo funkcja, albo - ogólnie mówiąc - obiekt.

Natomiast wskaźnik do niestatycznego, publicznego składnika klasy nie pokazuje na żadne konkretne miejsce pamięci, ale raczej mówi nam o ile komórek dalej od początku obiektu danej klasy znajduje się zawsze żądany składnik. Mówi na przykład: „23 komórki dalej". Aby więc się dostać do tego składnika musimy złożyć dwie informacje:

- - to, gdzie w pamięci zaczyna się dany obiekt,
- - to, o ile dalej jest żądany składnik obiektu.

Za złożenie tych informacji jesteśmy odpowiedzialni my sami. Posłużenie się tym wskaźnikiem ma sens tylko wtedy, gdy jeszcze powiemy o jaki obiekt nam

chodzi - albo podamy nazwę tego obiektu, albo pokażemy na niego innym wskaźnikiem.

Mówiąc po prostu: aby odnieść się do danego składnika w obiekcie potrzebna jest składnia

obiekt.*wskaźnik

gdzie wskaźnik jest tym wskaźnikiem do pokazywania na rysunek techniczny (deklarację klasy).

Samo wyrażenie

*wskaźnik

nie oznacza nic sensownego. To tak, jakbyśmy Tomaszowi pokazali tylko część na rysunku technicznym, ale nie powiedzieli, w którym egzemplarzu samolotu ma tę część przebadać.

Oczywiście, aby w ten sposób na coś sensownego pokazać, najpierw musimy ten wskaźnik na coś sensownego ustawić. Oto klasa:

```
class concorde {
     // ...
public:
     int cieglo_steru ;
     int cieglo_klap
     // ...
} ;

int concorde::*wskaz ;
     wskaz = &concorde::cieglo_steru ;
```

Widzimy, że w klasie jest składnik typu `int` o nazwie `cieglo_steru`. Definiujemy więc wskaźnik mogący pokazywać na składniki typu `int` w tej klasie.

Następnie ustawiamy składnik `wskaz` tak, by pokazywał na cięgło_steru. Wolno nam, bo jest to składnik publiczny i niestatyczny.

Wskaźnik `wskaz` pokazuje odtąd na cięgło steru. Przypominam, że nie na żadne konkretne cięgło steru, tylko na jego umiejscowienie w klasie (na rysunku technicznym). Najlepszy dowód, że w powyższej instrukcji nie wystąpiła, nazwa żadnego konkretnego obiektu klasy `concorde`, ale tylko nazwa samego typu samolotów.

Powyższą definicję i ustawienie wskaźnika można było zrobić w tej samej linijce, ale zapis wygląda trochę zawile, więc staram się go unikać

```
int concorde::*wskaz = &concorde::cieglo_steru ;
```

A teraz załóżmy, że chcemy odnieść się do składnika konkretnego egzemplarza obiektu klasy `concorde`.

Niech ten konkretny egzemplarz samolotu nazywa się `hugo` (od Victora Hugo)

```
concorde hugo ;                    // definicja tego samolotu
```

Odniesienie się w tym konkretnym egzemplarzu do składnika pokazywanego przez nasz wskaźnik wygląda następująco

```
hugo.*wskaz
```

Wskaźnik – jak wiadomo – można przestawiać. Tu można go przestawić tak, by pokazywał we wnętrzu tej klasy na inny obiekt tego samego typu (tu: inny składnik `int` w tej klasie).

Zauważ podobieństwo do zapisu

```
obiekt.składnik
```

Teraz składnik został zastąpiony przez `*wskaz`

A teraz sytuacja, gdy na samolot o nazwie `hugo` nie mówimy po nazwisku, tylko pokazujemy na niego innym, zwykłym wskaźnikiem

```
concorde * palec ;        // ←definicja wskaźnika do
                          // pokazywania na te samoloty
palec = &hugo ;           // ←skierowanie palca na konkretny samolot
```

Pokazanie na składnik w samolocie samolotu, który właśnie pokazujemy palcem

```
palec-> *wskaz ;
```

Są tu dwa wskaźniki, ale oczywiście tylko ten drugi jest przedmiotem naszego rozdziału.

Myślę, że zdajesz sobie już sprawę z faktu, iż wskaźnik do składnika klasy jest na tyle różny od zwykłego wskaźnika, że nie nadaje się do tych celów, co zwykły wskaźnik:

- Próba ustawienia wskaźnika do składnika klasy na coś, co nie jest składnikiem klasy - spowoduje protest kompilatora.
- Także odwrotnie: jeśli wskaźnik do pokazywania na zwykłe obiekty zechcesz ustawić na składnik w definicji klasy (rysunek techniczny) – spowoduje to protest kompilatora.

Czy na wszystkie składniki klasy można pokazywać takim wskaźnikiem ?

Nie:

Jeśli składnik jest niepubliczny, to nie można uzyskać informacji o jego umiejscowieniu w klasie. (Detal, który jest ściśle tajny, nie jest rysowany na rysunku technicznym dla wszystkich).

Jeśli w klasie `concorde` jest tajne cięgło do otwierania luków bombowych, to nie da się na nie pokazać wskaźnikiem.

```
wskaz = &concorde::cieglo_bomb;      // protest kompilatora !
```

Po prostu kompilator strzeże prywatności klasy.

Nie da się też ustawić wskaźnika w klasie na coś, co nie ma swojej własnej nazwy.

Na przykład jeśli składnikiem klasy jest tablica liczb `int`, to można pokazać na tę tablicę (ona ma swoją nazwę), ale nie można pokazać tym wskaźnikiem na jej piąty element, bo on swojej **własnej** nazwy nie ma.

16.3 Wskaźnik do funkcji składowej

Składnikiem klasy, jak wiemy, może być także funkcja. Również na taką funkcję składową można pokazać wewnątrz klasy wskaźnikiem.

Oczywiście ani przez chwilę nie łudziłeś się chyba, że tym samym wskaźnikiem ! Przywykłeś już przecież przy zwykłych wskaźnikach, że wskaźnik, który służy do pokazywania na obiekty typu `float`, nie może służyć do pokazania na obiekty typu `char`. Ani tym bardziej do pokazania na funkcję wywoływaną z czterema argumentami typu `int`, a zwracającą `float`.

To samo z tymi wskaźnikami do składników klas. Jeden pokazuje tylko na składniki typu `int`, a inny typu `char`. Jeśli chcemy zaś pokazać na funkcję składową, to też musimy przygotować oddzielny wskaźnik. O tym, jak to zrobić, pomówimy teraz. Oto przykład:

```
class concorde {
public :
     int ster ;
     int podwozie ;
     // ----------- funkcje składowe
     int tankowanie(float);
     int start(float) ;
     int zaladunek(float) ;

} ;
```

Możemy sobie zdefiniować wskaźnik do pokazywania na określone funkcje składowe

```
int (concorde::*wskfun)(float) ;
```

Definicję tę czytamy zaczynając od nazwy:

`wskfun`

`*`	- jest wskaźnikiem
`concorde::`	- do pokazywania na składniki klasy `concorde`
`(float)`	- będące funkcjami wywoływanymi z arg. typu `float`
`int`	- a zwracającymi rezultat typu `int`

Zauważ, że wyrażenie `(concorde::*wskfun)` zostało ujęte w nawias. To oczywiście dlatego, że operator `()` - wywołania funkcji znajdujących się bezpośrednio za tym wyrażeniem jest o wiele mocniejszy od operatora `*` (por. tabela na str. 72).

Gdybyśmy więc zapomnieli ująć w nawias nazwę i napisali tak:

```
int concorde::*wskfun(float) ;
```

to otrzymamy coś zupełnie odmiennego. Przeczytajmy co

`wskfun`

`(float)`	- jest funkcją wywoływaną z 1 argumentem typu `float`
`*`	- która jako rezultat zwraca wskaźnik
`concorde::`	- pokazujący we wnętrzu klasy `concorde`

`int` - na obiekty typu `int`
czyli coś zupełnie innego niż zamierzaliśmy. Chcieliśmy mieć definicję wskaźnika, a zrobiliśmy deklarację funkcji. Pamiętajmy więc o nawiasach.

Sposób dla leniwych

Na stronie 211 pokazałem sposób jak, nie wysilając się zbytnio, napisać definicję wskaźnika mogącego pokazywać na wybraną funkcję. Tu chciałbym uzupełnić i powiedzieć, jak łatwo napisać definicję wskaźnika mogącego pokazać na wybrane funkcje składowe klasy.

Sposób jest także bardzo prosty. Bierzemy deklarację jednej z funkcji, na którą ma pokazywać wskaźnik, i jej nazwę zastępujemy nazwą wskaźnika z gwiazdką. Czyli czymś takim

*nazwa_wskaźnika

Zatem jeśli chcemy mieć wskaźnik mogący pokazać na funkcję

```
int concorde::zaladunek(float);
```

to nazwę `zaladunek` zamieniamy na nazwę wskaźnika z gwiazdką. np. `*www`. Ale uwaga: pozostaje teraz ujęcie w nawias - i tu jest różnica w stosunku do zwykłych funkcji. W nawias ujmujemy nazwę wskaźnika, gwiazdkę, ale także i nazwę klasy. Czyli **w sumie dokonujemy takiej zamiany**

nazwa_klasy::nazwa_funkcji ⟶ (nazwa_klasy::*wskaźnik)

W rezultacie powstaje poszukiwana definicja wskaźnika

```
int (concorde::*www)(float);
```

Gdy taką deklarację czytamy, to na widok operatora zakresu :: mówimy: „we wnętrzu klasy...".

Powyższa deklaracja (będąca też definicją) przeczytana na głos daje nam taką wypowiedź:

`www` - jest wskaźnikiem mogącym pokazywać we wnętrzu klasy `concorde` na funkcje wywoływane z jednym argumentem (typu `float`), a zwracające typ `int`

Skoro mamy już wskaźnik, to teraz można go na coś ustawić

Wskaźnik zdefiniowany tak:

```
int (K::*wskfun)(float) ;
```

nadaje się do pokazywania w tej klasie na każdą funkcję składową, która jest wywoływana z argumentem `float`, a zwraca rezultat `int`. Czyli np. funkcje składowe

```
int tankowanie(float);
int zaladunek(float);
```

Ustawienie wskaźnika odbywa się prostą instrukcją

```
wskfun = & concorde::tankowanie ;
```

Jeśli wskaźnik do funkcji ustawiamy na jakąś funkcję, to zwykle po to, by za chwilę uruchomić tę funkcję. Oczywiście funkcję składową uruchamia się na rzecz konkretnego obiektu jej klasy

```
concorde bialy ;

(bialy.*wskfun)(900.33) ;
```

Znowu zauważ nawias!

Ponieważ nasz wskaźnik pokazywał na funkcję `tankowanie`, to powyższa instrukcja odpowiada takiemu zapisowi:

```
bialy.tankowanie(900.33) ;
```

16.4 Tablica wskaźników do danych składowych klasy

Skoro w języku C++ można obiekty grupować w tablice - można też uczynić to z takimi obiektami jak wskaźniki. W tablicy oczywiście mogą być wskaźniki tylko jednego typu. (Nic dziwnego - zwykłe tablice też zawierają elementy jednego typu. Tablica `int` zawiera same elementy typu `int`).

Oto definicja tablicy wskaźników do składników klasy `K` :

```
int K::*tabwsk[10] ;
```

definicję tę czytamy:

`tabwsk`	
`[10]`	- jest 10 elementową tablicą
`*`	- wskaźników do pokazywania
`K::`	- we wnętrzu klasy `K`
`int`	- na składniki typu `int`

Ustawienie jednego ze wskaźników będących w tej tablicy tak, by pokazywał na składnik `ster` to instrukcja

```
tabwsk[5] = &K::ster ;
```

a odniesienie się do składnika w jakimś konkretnym obiekcie (o nazwie `pomaranczowy`) to wyrażenie

```
pomaranczowy.*tabwsk[5]
```

Co to oznacza? W tablicy chowaliśmy sobie różne wskaźniki do pokazywania we wnętrzu klasy `K` na składniki publiczne (niestatyczne) typu `int`. W poszczególnych elementach tej tablicy są wskaźniki. Jeden pokazuje na ten składnik `int`, a drugi na inny składnik `int`.

Powyższe wyrażenie oznacza więc jakby wypowiedź: bierzemy ten składnik `int` klasy `K`, na który pokazuje nam szósty wskaźnik z tej tablicy...

16.5 Tablica wskaźników do funkcji składowych klasy

Jest to nieco trudniejsze w zapisie, ale moim zdaniem przydaje się częściej. To jest jakby powiedzenie czegoś takiego: „Mam ponumerowane funkcje składowe. Który numer mam wykonać?"

Już zapewnie się domyśliłeś, że przyda się to przy tworzeniu menu.

Oto przykład definicji tablicy wskaźników do funkcji składowych. Oczywiście znowu nie wszystkich możliwych w tej klasie, ale o wybranej liczbie i typie argumentów oraz typie rezultatu.

```
int (K:: * tabfunskl[8]) (float) ;
```

czytamy to:

```
tabfunskl
```

`[8]`	- jest 8 elementową tablicą
`*`	- wskaźników pokazujących na
`K::`	- będące składnikami klasy `K`
`(float)`	- funkcje wywoływane z 1 argumentem typu `float`
`int`	- które w rezultacie zwracają wartość typu `int`

Trochę to wygląda skomplikowanie.

Ustawienie takich wskaźników zebranych w tablicę wykonuje się prostymi instrukcjami:

```
tabwskfun[0] = K::tankowanie ;
tabwskfun[1] = K::zaladunek ;
```

Przypominam zasadę:

> *Jeśli tylko o funkcji mówimy, a nie chcemy jej w tym momencie akurat wywoływać, to używamy tylko jej nazwy. Nawiasy (z argumentami) oznaczają wywołanie jej.*

Oto jak wywołujemy funkcję pokazywaną takim wskaźnikiem z tablicy. Oczywiście funkcję składową wywołuje się na rzecz konkretnego egzemplarza obiektu.

```
K:: egzemplarz ;                    // definicja obiektu

(egzemplarz. *tabfunskl[0]) (7.43) ;
```

Ponieważ wcześniej ten element tablicy wskaźników ustawiliśmy (na rysunku technicznym) na funkcję `tankowanie`, więc zapis powyższy odpowiada zapisowi

```
egzemplarz.tankowanie(7.43) ;
```

16.6 Wskaźniki do składników statycznych

W poprzednim paragrafie mówiliśmy, że do pokazywania na składniki w klasie potrzebny jest specjalny wskaźnik, który mówi nam, w którym miejscu w stosunku do początku obiektu znajduje się żądany składnik.

Nie wszystkie jednak składniki klasy wchodzą w skład poszczególnych egzemplarzy obiektów. Jak pamiętamy - składnik statyczny jest wspólny dla wszystkich egzemplarzy obiektów, ale nie znajduje się w żadnym z nich. Jest zdefiniowany osobno. (I to my sami musimy go zdefiniować osobno).

> W związku z tym, aby na ten składnik statyczny pokazać wskaźnikiem - używa się zwykłego wskaźnika. Tak, jakbyśmy mieli pokazywać na zwykły obiekt, nie będący składnikiem żadnej klasy.

To jest chyba zrozumiałe - taki składnik statyczny ma konkretny adres w pamięci, natomiast nie ma sensu powiedzenie o nim, że leży ileś komórek dalej od początku obiektu. On po prostu nie leży w obiekcie.

Jeśli więc chcemy pokazać na obiekt typu `int` będący składnikiem statycznym jakiejś klasy, to możemy się po prostu posłużyć wskaźnikiem

```
int *wsk ;
```

Podsumujmy dlaczego dla składników statycznych używa się zwykłych wskaźników, a dla składników niestatycznych - wskaźników specjalnych :

- - Składnik niestatyczny (zwykły) znajduje się w każdym obiekcie. Aby na niego pokazać trzeba obliczyć ile komórek w pamięci od początku obiektu znajduje się żądany składnik. To właśnie robi ten specjalny wskaźnik do pokazywania na składniki.
- - Składnik statyczny nie leży wewnątrz żadnego egzemplarza obiektu. Dlatego nie ma sensu używanie specjalnego wskaźnika, który przecież penetruje wnętrze obiektu. Pytanie o to, jak daleko od początku obiektu składnik ten się znajduje, nie ma sensu. Składnik ten jest zupełnie gdzie indziej. Tutaj powiedzmy sobie, że np. tam gdzie zmienne globalne. W *takie* miejsce pokazuje się wskaźnikami zwykłymi.

17 Konwersje

W rozdziale tym mówić będziemy o tym, jak sobie ułatwić programowanie. Zapoznamy się bowiem ze sposobami definiowania konwersji (przekształceń) obiektów jednego typu (klasy) na inny. Może nam to w przyszłości zaoszczędzić wiele pracy.

17.1 Sformułowanie problemu

Rozważmy klasę, która opisuje liczby zespolone

```
class zespol {
public :
     float rzeczyw ;
     float urojon ;
     // ----------konstruktor
     zespol(float r, float i) : rzeczyw(r), urojon(i) {}
} ;
```

Klasa jest prosta: zawiera dwa składniki-dane. (Jeden opisuje część rzeczywistą, drugi część urojoną). Do tego dochodzi konstruktor. Innych funkcji składowych dla prostoty nie wyszczególniamy.

Załóżmy teraz, że chcemy napisać funkcję, która dodaje dwie liczby zespolone. Taka funkcja może być funkcją składową tej klasy, a może być też funkcją globalną – dlatego, że w naszej klasie oba składniki są publiczne.

```
zespol dodaj(zespol a, zespol b)
{
zespol suma(0, 0) ;
     suma.rzeczyw = a.rzeczyw + b.rzeczyw ;
     suma.urojon = a.urojon + b.urojon ;
     return suma ;
}
```

Funkcja ta po prostu dodaje do siebie odpowiednie składniki. Rezultat jest wpisywany do lokalnego obiektu o nazwie `suma`. Ten obiekt stoi przy słowie `return`.

Przypominam co to znaczy.

> Obiekt `suma` jest lokalny i automatyczny, więc za chwilę przestanie istnieć. Ponieważ funkcja ma, jako rezultat, zwrócić obiekt klasy `zespol`, dlatego na ten cel zostaje przygotowany obiekt chwilowy. Na moment przed likwidacją obiektu `suma` treść tego obiektu jest kopiowana do obiektu chwilowego. Ten właśnie obiekt chwilowy udostępniany jest nam przez mechanizm `return`.

Mając tę funkcję możemy już przeprowadzić dodawanie.

```
zespol    pierwsza(1,1),              // składniki dodawane
          druga(6, -3) ,
          wynik(0,0) ;                // obiekt na wynik
wynik = dodaj( pierwsza, druga) ;// akcja dodawania
```

To było proste. Utrudnijmy to trochę. Jak wiemy, także liczba 0.5 jest liczbą zespoloną. Tyle, że taką szczególną, gdzie część urojona jest równa zero. Czy można taką liczbę też użyć w naszym dodawaniu?

```
wynik = dodaj(pierwsza, 0.5 ) ;
```

Nie, nie można. Po prostu nie zgadza się typ argumentu. Funkcja `dodaj` oczekuje przecież obiektu klasy `zespol`, a nie liczby `float`. Musimy więc zdefiniować inną funkcję na tę okoliczność. Możemy ją także nazwać `dodaj`, bo skoro argumenty będą inne, to nazwa zostanie przeładowana

```
zespol dodaj(zespol z, float f)
{
zespol wynik(0, 0) ;
     wynik.rzeczyw = z.rzeczyw + f ;
     wynik.urojon = z.urojon ;
     return wynik ;
}
```

Mając taką funkcję możemy już wykonać to kłopotliwe działanie.
Z kolei gdybyśmy chcieli także mieć możliwość zapisu

```
dodaj(66.1, druga);
```

to potrzebna jest nam funkcja

```
zespol dodaj(float, zespol) ;
```

Musimy więc trzy razy zdefiniować prawie identyczne funkcje.

Pomyślisz zapewne – „Czy kompilator nie wie, że liczbę `float` można przedstawić jako szczególnego rodzaju liczbę zespoloną?" Oczywiście, że nie wie. Trzeba mu to najpierw powiedzieć. **W tym rozdziale zajmiemy się właśnie tym, jak mu takie rzeczy mówić.**

Gdy kompilator będzie już to wiedział – zaoszczędzimy sobie pracy. Wystarczy wówczas jedna definicja

```
zespol dodaj(zespol, zespol) ;
```

która załatwi nam wszystkie trzy omawiane sytuacje.

Zamiana typu `float` na typ `zespol` to inaczej *konwersja* typu `float` na `zespol`.

Konwersje obiektu typu A na typ Z mogą być zdefiniowane przez użytkownika.

Służą do tego:

- –albo konstruktor klasy Z przyjmujący jako jedyny argument obiekt typu A
- –albo specjalna funkcja składowa klasy A zwana funkcją konwertującą (inaczej – operatorem konwersji)

> Cała wspaniałość konwersji definiowanej przez użytkownika polega na tym, że konwersje mogą być przeprowadzane niejawnie czyli samoczynnie. Po prostu wtedy, gdy zachodzi niedopasowanie typu – kompilator sprawdza czy za pomocą jakiejś konwersji nie da się danego typu zamienić na inny. Taki, który pasuje do danej sytuacji.

Oczywiście konwersje te mogą być też wywoływane jawnie – tak, jak zwykłe funkcje, ale to już przecież nic nowego. – Jeśli napisaliśmy jakąś funkcję, to ją przecież możemy jawnie wywołać.

17.2 Konstruktor jako konwerter

Konstruktor, przyjmujący jeden argument, określa konwersję od typu tego argumentu do typu klasy, do której sam należy.

Zatem jeśli w naszej klasie dla liczb zespolonych zdefiniujemy taki konstruktor

```
zespol::zespol(float r)
{
     rzeczyw = r ;
}
```

to tym samym określiliśmy jak zrobić konwersję typu `float` na typ `zespol`.

Zwracam uwagę, że ten sam efekt mogliśmy w naszej klasie `zespol` uzyskać jeszcze prościej — deklarując mianowicie drugi argument istniejącego już konstruktora jako domniemany.

```
zespol(float r, float i = 0 );
```

W obu sytuacjach efekt jest ten sam: mamy konstruktor, który można wywołać z jednym argumentem typu `float`.

Co ciekawsze – taki konstruktor będzie zawsze niejawnie wywoływany, gdy tylko funkcja oczekuje argumentu typu `zespol`, a dostanie typ `float`. To wspaniałe narzędzie sprawi, że od tej pory zamiast funkcji:

```
zespol dodaj(zespol, zespol) ;
zespol dodaj(zespol, float) ;
```

```
zespol dodaj(float, zespol) ;
zespol dodaj(float, float) ;
```

wystarczy tylko jedna

```
zespol dodaj(zespol, zespol) ;
```

Jak się to dzieje ?

Otóż teraz w wypadku, gdy kompilator zobaczy wyrażenie

```
wynik = dodaj(pierwsza, 7.5) ;
```

to zrozumie je jako

```
wynik = dodaj(pierwsza, zespol(7.5) );
```

Jak widzimy, zostało tam umieszczone wywołanie konstruktora, które zamieni liczbę `float` = 7.5 na obiekt chwilowy klasy `zespol` i ten obiekt zostanie przesłany do funkcji `dodaj`. Mówimy, że nastąpiła konwersja typu `float` do typu `zespol`.

Po wykonaniu wyrażenia, w którym wystąpiła ta funkcja, obiekt chwilowy zostanie zlikwidowany. Kiedy dokładnie – to zależy od implementacji.

Ważne jest to, że konwersja nastąpiła niejawnie. Nie musieliśmy niczego specjalnie wyszczególniać. Kompilator spodziewał się typu `zespol`, a zobaczył typ `float`. Wobec tego szukał: czy jest jakiś sposób na zamianę typu `float` na typ `zespol` ? Jest! – to dostarczony przez programistę konstruktor obiektu klasy `zespol` przyjmujący jeden argument – właśnie typu `float`. Sprawa rozwiązana.

Konstruktor taki może być też wywołany jawnie. Oto definicja obiektu klasy `zespol` przy użyciu tego konstruktora

```
zespol czworka = zespol(4.0) ;
```

Jawne wywołanie konstruktora może służyć rozwianiu wątpliwości jaki sposób konwersji kompilator ma wybrać (gdyby było kilka).

Widzimy już więc, że dzięki zastosowaniu konwersji zapobiegamy konieczności wielokrotnego przeładowania funkcji – czyli oszczędzamy sobie pracy.

Typem, z którego dokonuje się przekształcenia nie musi być koniecznie typ wbudowany. Może to być także inna klasa.

Konstruktor wówczas musi przyjrzeć się obiektowi tamtej drugiej klasy i na tej podstawie skonstruować obiekt swojej klasy. Aby to „przyjrzenie się" było możliwe, ta druga klasa powinna zadbać o to, by nasz konstruktor miał dostęp do ważnych dla niego składników. Dostęp ten może mu nadać :

- przez uczynienie interesujących składników publicznymi (co by było bardzo nierozsądne),
- przez deklarację przyjaźni z tym obcym konstruktorem, który ma z nich korzystać,

- przez publiczne funkcje składowe, które pozwolą uzyskać wszystkie interesujące informacje. Te funkcje może uruchomić każdy, więc także nasz konstruktor.

Najlepiej zilustruje to przykład. Zobaczymy w nim dwie klasy. Jedna z nich to klasa `zespol`. Zauważ, że teraz dane składowe są już prywatne.

Oto ich definicje:

```
class numer ;                                                    // ❸
//////////////////////////////////////////////////////////////////
class zespol {
private :
     float rzeczyw ;
     float urojon ;
public :
     // ----------konstruktory
     zespol(float r = 0, float i = 0) :
                    rzeczyw(r), urojon(i) {}                     // ❶
     zespol(numer) ;                                             // ❷
} ;
//////////////////////////////////////////////////////////////////
class numer {
     float n ;
     char opis[80] ;
     // deklaracja przyjaźni z takim konstruktorem
     //                                  z innej klasy
     friend zespol::zespol(numer) ;                              // ❹
     // ...
public :
     // ------- zwykły, własny konstruktor
     numer(int k , char * t = "bez opisu"): n(k)
               { strcpy(opis, t);      }
} ;
```

Przyjrzyjmy się tym klasom

❶ Deklaracja konstruktora konwertującego z typu `float` na typ własnej klasy – czyli `zespol`. Ma on argument domniemany `i`, więc można go wywołać także jako `zespol(float)` - a o to nam chodzi. Tym sposobem będziemy mieli konwersję typu `float` na typ `zespol`.
Zauważ, że teraz jesteśmy jeszcze sprytniejsi. Także pierwszy argument jest domniemany (przez domniemanie = 0). Dzięki temu w rezultacie takiej definicji:

```
zespol xxx ;
```

powstanie obiekt, w którym część urojona i część rzeczywista jest zerowa.

❷ Deklaracja konstruktora, który zamieni typ `numer` na typ swojej klasy. Tym sposobem będziemy mieli konwersję typu `numer` na typ `zespol`. Ponieważ `numer` nie jest typem wbudowanym, więc kompilator nie zna tej nazwy. To dlatego powyżej ❸ musieliśmy powiedzieć, że `numer` jest nazwą jakiejś klasy. (Deklaracja zapowiadająca).

❹ Natomiast w klasie `numer` (taka sobie nieciekawa klasa) jest deklaracja przyjaźni. Deklaracja ta mówi, że klasa `numer` zgadza się, by wyspecyfikowany

konstruktor z obcej klasy miał dostęp do jej składników prywatnych. (Pamiętasz oczywiście, że fakt iż deklaracja przyjaźni jest tu akurat w części `private` nie ma żadnego znaczenia. Deklaracja przyjaźni może być w dowolnej części definicji klasy).

A oto realizacja tego ❷ konstruktora klasy `zespol`:

```
zespol::zespol(numer ob)
{
     rzeczyw = ob.n ;
     urojon = 0 ;
}
```

Co tu jest interesującego? To, że może on odnieść się do prywatnego składnika obiektu klasy `numer` tak, jakby był on publiczny. Liczba wyjęta ze składnika o nazwie `n` jest użyta do przypisania do części rzeczywistej nowo konstruowanego obiektu.

Definicji funkcji `dodaj` (przy której wywołaniu zajdą konwersje) nie zamieszczam, bo jest identyczna jak poprzednio. Oto przykładowe użycie konwersji we fragmencie programu:

```
// ...
numer num(4, "czworka");
zespol z(10, 9) ,
          w ;                          // obiekt o treści zerowej

     w = dodaj(z, 7.5) ;        // czyli dodaj(z, zespol(float) );      ❺
     w = dodaj(z, num) ;        // czyli dodaj(z, zespol(numer) );      ❻
// ...
```

❺ Tu zostanie wywołana konwersja z typu `float` na typ `zespol`. O tym mówiliśmy powyżej.

❻ Tu natomiast nastąpi konwersja z typu `numer` na typ `zespol`. Oczywiście odbędzie się to przez niejawne wywołanie naszego konstruktora konwertującego

```
zespol::zespol(numer)
```

O tym, że tak jest w istocie, można się zawsze przekonać prostym sposobem. W takiej niejawnie wywoływanej funkcji wstawiamy instrukcję

```
cout << "To ja, twoj konstruktor konwertujacy \n"
      "z typu numer na typ zespol \n" ;
```

Teraz – ile razy konstruktor będzie pracował – zawsze się o tym dowiesz. Czasem takie zabawy bywają pouczające.

17.3 Funkcja konwertująca – operator konwersji

Wyobraźmy sobie sytuację odwrotną do poprzedniej. Mamy funkcję, która przyjmuje jako argument typ wbudowany (np. `float`)

```
void funkcja (float x );
```

a chcielibyśmy móc wywołać ją także z argumentem będącym obiektem jakiejś klasy.

```
zespol z ;
     funkcja(z) ;
```

Poprzednie rozwiązanie się nie nadaje. Polegało ono na zastosowaniu konstruktora, który stworzy obiekt żądanego (wynikowego) typu. Tym żądanym typem jest teraz `float` – czyli typ wbudowany. Nie możemy przecież napisać żadnego konstruktora w „klasie" `float`.

Co robić? Jest wyjście: musimy naszą klasę `zespol` wyposażyć w funkcję składową, która zajmie się przekształceniem obiektu tej klasy na typ `float`. Funkcja taka zwana jest funkcją konwertującą albo inaczej (choć niezbyt precyzyjnie) operatorem konwersji.

> Funkcją konwertującą jest funkcja składowa klasy `K`, która się nazywa
>
> `K::operator T()`
>
> gdzie `T` jest nazwą typu, na który chcemy dokonać konwersji. (`T` może być nawet nazwą typu wbudowanego).

Tym sposobem wyposażyliśmy kompilator w narzędzie do dokonywania konwersji z typu K na typ T.
Zróbmy to na przykładzie. Oczywiście musimy być najpierw świadomi tego, co chcemy otrzymać po konwersji. Załóżmy, że mamy klasę liczb zespolonych i wymyśliliśmy, iż konwersja na typ `int` polega na wzięciu części całkowitej składnika rzeczywistego. (Urojony zaniedbujemy). Definicja takiego operatora konwersji jest bardzo prosta:

```
class zespol {
     // ...
public :
     operator int()
     {
          return (int) rzeczyw ;  // prościej: return rzeczyw;
     }
} ;
```

To wszystko. Od tej pory możemy naszą liczbę zespoloną wysłać do wszystkich funkcji, które jako argument spodziewają się typu `int`. I to nie tylko do naszych funkcji w programie, ale nawet do funkcji bibliotecznych. Kompilator bowiem za pomocą tego operatora dokona zamiany i wyśle już tam liczbę typu `int`.

Jeśli więc jest gdzieś funkcja

```
void kukulka(int ile_razy) ;
```

to możemy teraz napisać takie jej wywołanie

```
zespol zzz(5.1, 444.5) ;      // def. przykładowego obiektu
     //-------------- wywołanie w którym nastąpi niejawna konwersja
     kukulka(zzz);
```

Bez jawnego specyfikowania jest ona wywoływana w razie potrzeby. Oczywiście możemy też, wywołać ten operator jawnie.

```
int i ;
i = (int)zzz ;
i = int(zzz) ;
```

Jak widać dopuszczalne są tu dwa zapisy. Jeden, który przypomina rzutowanie, a drugi, który przypomina wywołanie funkcji.

Wróćmy jednak do samej definicji funkcji konwertującej.

Należy tu zapamiętać parę ważnych rzeczy:

- 1) Funkcja konwertująca (operator konwersji) **musi być funkcją składową** klasy. Jest więc w niej wskaźnik `this` do obiektu, który ma poddać konwersji.
- 2) Funkcja ta **nie ma określenia typu rezultatu** zwracanego. To chyba oczywiste – funkcja ta przecież zawsze zwraca taki typ, jak się sama nazywa, więc dodatkowe określanie typu rezultatu byłoby powtarzaniem się.
- 3) Funkcja ta **ma pustą listę argumentów.** Skoro tak, to automatycznie wykluczona jest możliwość przeładowania jej.

Następne punkty są już dla wtajemniczonych:

- 4) Funkcja konwertująca **jest dziedziczona.** Czyli klasy pochodne mają ją automatycznie – chyba, że ją zasłonią definiując swoją własną wersję.
- 5) Funkcja konwertująca **może być funkcją wirtualną.**

Dotychczas pokazaliśmy, że typ, na który odbywała się konwersja, był typem wbudowanym (u nas `int`). To dobry przykład, bo wyklucza myśl o definiowaniu konstruktora konwertującego w klasie `int` - takiej klasy nie ma.

Teraz jednak wypada wspomnieć, że funkcja konwertująca może dokonać równie dobrze konwersji na typ zdefiniowany przez użytkownika (na inną inna klasę).

Oto przykład

Chcemy zdefiniować funkcję konwertującą z typu `zespol` na typ `numer` czyli w odwrotną niż poprzednio stronę. Aby to uczynić w klasie `zespol` umieszczamy deklarację funkcji konwertującej

```
operator numer() ;
```

natomiast sama definicja tej funkcji konwersji wygląda tak

```
zespol::operator numer()
{
     numer nnn(rzeczyw , "powstal z zespolonej");
     return nnn ;
}
```

Co się dzieje w ciele naszej funkcji konwertującej? Bierzemy część rzeczywistą obiektu klasy `zespol` (możemy, bo jesteśmy funkcją składową klasy `zespol`)

– tę wartość używamy do wywołania konstruktora obiektu klasy `numer`. Ten obiekt jest zwracany jako rezultat funkcji.

Zobaczmy teraz ten program całościowo

```
#include <iostream.h>
#include <string.h>
class numer ;                                                    // ❸
//////////////////////////////////////////////////////////////
class zespol {
     float rzeczyw ;                                             // ❶
     float urojon ;
public :
     // dwa konstruktory konwertujące
     zespol(float r = 0, float i = 0):rzeczyw(r),urojon(i)
     { }
     zespol(numer ob);                                           // ❷

     // dwa operatory konwersji
     operator float() { return rzeczyw ; }                       // ❹
     operator numer() ;
     void pokaz()
     {
          cout << "\tLiczba zespolona: (" << rzeczyw
               << ", " << urojon << ") \n" ;
     }
     friend zespol dodaj(zespol a, zespol b) ;
} ;
//////////////////////////////////////////////////////////////
class numer {
     float n ;
     char opis[80];                                              // ❺
     friend zespol::zespol(numer);
     friend void plakietka(numer);
public:
     numer(float k, char *t = "opis domniemany")
                                                  : n(k)         // ❻
     {
          strcpy(opis, t);
     }
     operator float() { return n ; }                             // ❼
};
//////////////////////////////////////////////////////////////
zespol::zespol(numer ob) : rzeczyw(ob.n), urojon(0)
{ /* puste ciało, bo wszystko zrobione w liście inicjalizacyjnej */ }
/*****************************************************/
zespol::operator numer()
{    // pomagamy sobie wywołując konstruktor (ale już nie konwertujący,
     // bo wywoływany z 2 argumentami
     return numer(rzeczyw, "powstal z zespolonej") ;
}
/*****************************************************/
// deklaracje funkcji globalnych
void pole_kwadratu(float bok);
void plakietka(numer nnn);
zespol dodaj(zespol a, zespol b);
```

```
/*****************************************************/
main()
{
     // definicje trzech obiektów
     float     x = 3.21 ;
     numer     nr(44, "a imie jego") ;
     zespol    z(6, -2);

     // wywołania funkcji pole_kwadratu(float);                ❽
     pole_kwadratu(x);              // niepotrzebna żadna konwersja
     // poniższe wywołania nie są dopasowane, ale mimo to możliwe, bo
     // kompilator samoczynnie zastosuje nasze konwersje
     pole_kwadratu(nr);        // operator konwersji numer-->float
     pole_kwadratu(z);         // operator konwersji zespol-->float

     //-------------------------------------------
     zespol    z2(4,5) ,            // def 2 roboczych obiektów
               wynik ;

     // wywołania funkcji dodaj(zespol, zespol)                ❾

     wynik = dodaj(z, z2) ;         // niepotrzebna żadna konwersja
     wynik.pokaz();

     // poniższe wywołania nie są dopasowane, ale mimo to możliwe, bo
     // kompilator samoczynnie zastosuje nasze konwersje

     wynik = dodaj(z, x);   // konstr. konwertujący float-->zespol
     wynik.pokaz();

     wynik = dodaj(z, nr); // konstr. konwertujący numer-->zespol
     wynik.pokaz();

     //-----------------------------------------------------

     // wywołania funkcji plakietka(numer) ;                   ❿
     plakietka(nr);

     // poniższe wywołania nie są dopasowane, ale mimo to możliwe, bo
     // kompilator samoczynnie zastosuje nasze konwersje
     plakietka(x);            // konstr. konwertujący float --> numer
     plakietka(z);            // operator konwersji zespol --> numer

}
/*****************************************************/
zespol dodaj(zespol a, zespol b)
{
     zespol chwilowy(a.rzeczyw + b.rzeczyw,
                     a.urojon + b.urojon);
     return chwilowy ;
}
/*****************************************************/
void plakietka(numer nnn)
{
     cout << "****************************" << endl ;
     cout << "***  ***\r"
```

```
            << "*** " << nnn.opis << endl ;
      cout << "*** ***\r"
            << "*** " << nnn.n << endl ;
      cout << "****************************" << endl ;
}
/*****************************************************/
void pole_kwadratu(float bok)
{
      cout << "Pole kwadratu o boku " << bok
            << " wynosi " << (bok * bok) << endl ;
}
```

W rezultacie na ekranie pojawi się

```
Pole kwadratu o boku 3.21 wynosi 10.3041
Pole kwadratu o boku 44 wynosi 1936
Pole kwadratu o boku 6 wynosi 36
     Liczba zespolona: (10, 3)
     Liczba zespolona: (9.21, -2)
     Liczba zespolona: (50, -2)
****************************
*** a imie jego          ***
***            44        ***
****************************
****************************
*** opis domniemany      ***
***            3.21      ***
****************************
****************************
*** powstal z zespolonej ***
***            6         ***
****************************
```

Ciekawsze miejsca programu

❶ Deklaracja klasy `zespol` (liczba zespolona). Spotkaliśmy się już z tą klasą. Składnikami tej klasy są dwie liczby typu `float` nazywane umownie częścią *rzeczywistą* i częścią *urojoną*.

❷ Dwa konstruktory konwertujące. Ten pierwszy potrafi zbudować obiekt klasy `zespol` z obiektu typu `float`. Drugi konstruktor, ten przyjmujący jeden argument typu `numer`, jest to konstruktor, który potrafi zamienić obiekt klasy `numer` na obiekt klasy `zespol`. Tym samym wyposażamy kompilator w narzędzia do niejawnej, samoczynnej konwersji

```
float   ———→   zespol
numer   ———→   zespol
```

W deklaracji tego drugiego konstruktora kompilator napotyka nazwę `numer`. Aby się nie zdziwił i nie protestował, że nie wie co to jest, powiedzieliśmy mu to w ❸.

❸ Deklaracja zapowiadająca. Jest to poinformowanie kompilatora,że: "jakby co, to `numer` jest nazwą jakiejś klasy".

❹ Operator konwersji (funkcja konwertująca) z typu `numer` na typ `float`.

❺ Właściwa deklaracja klasy `numer`. Jak widzimy, obiekt klasy `numer` zawiera liczbę typu `float` oraz tablicę znaków do przechowywania tekstu opisu tej liczby.
Poniżej widzimy też, że klasa deklaruje przyjaźń z konstruktorem konwertującym z klasy `zespol` - czyli udostępnia mu informację tak, by mógł z łatwością zamieniać obiekt klasy `numer` na obiekt swojej klasy.

❻ Konstruktor konwertujący obiekt typu `float` na typ `numer`. (Dzięki temu, że jest argument domniemany).

❼ Funkcja konwertująca obiekt klasy `numer` na typ `float`.

❽ Wywołania funkcji `pole_kwadratu` na rzecz obiektów trzech różnych typów. Pierwsze wywołanie jest oczywiste - funkcja wywołana jest z takim argumentem jakiego oczekuje. Ciekawsze są dwa pozostałe wywołania, bo **tu objawia się po raz pierwszy w tym przykładzie wspaniałość konwersji**. Oto funkcję, która się spodziewa argumentu typu `float` można wywołać dla argumentu typu `numer` lub typu `zespol`.
Kompilator niejawnie, samoczynnie, posługując się dostarczonymi operatorami konwersji, dokonuje konwersji i jej rezultat - już liczbę `float` wysyła do funkcji `pole_kwadratu`.

❾ Podobnie tutaj. Teraz dzięki samoczynnym konwersjom można wywołać funkcję `dodaj(zespol, zespol)` nawet dla argumentów typu `float` i `numer`. Którym narzędziem posłuży się kompilator — zaznaczyłem w komentarzach.

❿ Wywołania funkcji `plakietka(numer)`. Dzięki wspaniałości konstruktorów konwertujących i operatorów konwersji możliwe są nawet takie wywołania.

17.4 Który wariant konwersji wybrać ?

Jak widzimy, do przekształcenia jednego typu na drugi, mamy do dyspozycji dwa narzędzia:

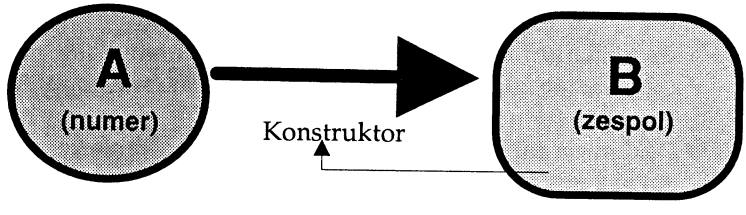
- 1) konstruktor jednoargumentowy,
- 2) funkcję konwertującą (operator konwersji).

Konstruktor zapewnia nam przekształcenie od jakiegoś obcego typu na typ swojej klasy np. `zespol(7.5)`

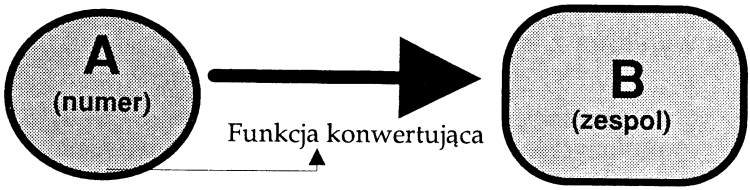
Funkcja konwertująca (operator konwersji) robi odwrotnie. Przekształca od typu swojej klasy na typ obcy.

Mówiąc ogólniej - jeśli mamy konwersję między dwoma klasami np. od klasy A do klasy B, to możemy ją zrealizować

❖ albo za pomocą odpowiedniego konstruktora w klasie B



- albo za pomocą funkcji konwertującej w klasie A



Można wybrać jeden z tych wariantów. Jeśli wybierzemy obydwa – to znaczy zdefiniujemy i konstruktor i funkcję konwertującą dla danej konwersji, to – w wypadku, gdy kompilator napotka w programie na linijkę, gdzie będzie musiał niejawnie wybrać jeden ze sposobów konwersji - wówczas stanie bezradny i zaprotestuje. Nie może być bowiem żadnej wieloznaczności. Oba sposoby konwersji są dla niego jednakowo dobre.

No, chyba że to my – jawnie wywołamy konwersję (konstruktor lub funkcję konwertującą). Wówczas kompilator nie musi niczego decydować – zrobi to, o co go prosimy.

Jednak nie polecam tego. Jak już mówiłem – wspaniałość konwersji polega na tym, że zachodzą niejawnie.

Wniosek więc taki: musimy wybrać jeden ze sposobów konwersji. Albo-albo. Wobec tego co wybrać?

Sprawa nie jest trudna. To dlatego, że te dwa warianty nieco się różnią. Oto te różnice:

Konstruktor ma pewne wady

- Nie można zdefiniować konstruktora dla typu wbudowanego.
- Nie można napisać konstruktora dla klasy, która nie jest naszą własnością – np. będącą składnikiem biblioteki. Modyfikacje takiej klasy są zwykle niepożądane.
- Nawet jeśli klasa jest naszą własnością, to konstruktor, który chcemy napisać, musi oprzeć się na informacjach z tej obcej klasy. Tamta obca klasa musi zapewnić sposoby dotarcia do tych informacji. (Robi to: albo przez publiczne dane składowe, albo przez deklarację przyjaźni). Jeśli ta obca klasa nie zapewnia nam tych informacji, to musimy ją zmodyfikować.
- Przy konstruktorze konwertującym argument musi pasować dokładnie do typu argumentu deklarowanego w konstruktorze. Nie możemy polegać na żadnych- tak zwanych konwersjach standardowych.[†)]

(dla wtajemniczonych)

- Konstruktora służącego do konwersji nie dziedziczy się (bo nie dziedziczy się przecież żadnych konstruktorów). Natomiast funkcja konwer-

†) O tym niebawem – paragraf o niedobranych małżeństwach.

tująca jest funkcją składową, więc może być dziedziczona. I to w sposób „mądry" – może być bowiem wirtualna.

Jakie wynikają z tego wskazówki

1) Aby zapewnić konwersję obiektu klasy `A` na typ wbudowany – musisz posłużyć się operatorem konwersji. Innej drogi nie ma.

2) Jeśli masz zapewnić konwersję obiektu klasy A na obiekt klasy B to:

- a) Staraj się napisać funkcję konwertującą w klasie A. Nie wymaga to **żadnych** ingerencji w klasę B.
- b) Jeśli klasa A jest Ci niedostępna, (bo na przykład jest z biblioteki) to spróbuj w klasie B napisać konstruktor konwertujący obiekt klasy A na obiekt klasy B, czyli konstruktor

  `B::B(A)`

  Uda Ci się to pod warunkiem, że interesująca Cię część informacji o obiekcie klasy A jest Ci dostępna. Jeśli nie jest, to nic nie zrobisz - nie możesz przecież modyfikować klasy A (gdybyś mógł – wybralibyśmy wariant a).

17.5 Sytuacje, w których zachodzi konwersja

Siłą tego narzędzia, jakim są konwersje definiowane przez użytkownika, jest to, że zachodzą one niejawnie. W tym paragrafie zbierzemy sytuacje, kiedy kompilator sięga po te konwersje. A zatem

Konwersje są wykonywane niejawnie, gdy:

1) Kompilator, **w wywołaniu funkcji**, widzi niezgodność argumentów aktualnych (wywołania) z argumentami formalnymi, a równocześnie jest jednoznaczna możliwość by za pomocą konwersji to niedopasowanie usunąć.

2) Gdy, **przy zwracaniu rezultatu funkcji**, obiekt stojący obok słowa `return` jest innego typu niż deklarowany dla funkcji typ rezultatu.
Jeśli funkcja ma zwrócić typ `B`, a obok słowa `return` stoi obiekt typu `A` – to jeśli tylko jest zdefiniowany jednoznaczny sposób, jak zamienić obiekt klasy `A` na obiekt klasy `B` – wówczas konwersja ta jest niejawnie przeprowadzana.

3) **W obecności operatorów**. Jeśli mamy operator dodawania (inaczej mówiąc znaczek '+') to spodziewa się on (zwykle), że obok będą stały jakieś obiekty typu `float` lub `int` – ogólnie mówiąc typu `B`

```
3.4 + 7
```

Jeśli postawimy tam obiekt klasy `A`

```
A obj ;
3.4 + obj ;
```

to jeśli tylko zdefiniowaliśmy jak z obiektu klasy `A` zrobić obiekt typu `B`, wówczas zajdzie taka niejawna konwersja z typu `A` na typ `B`.

W naszym przykładzie z liczbami zespolonymi zdefiniowaliśmy konwersję z obiektu `zespol` na `int` – więc tu właśnie by ona zaszła.

```
zespol zzz(4.4, 10.0) ;
int m ;

     m = 1 + zzz ;          // czyli m = 1 + 4 ;
```

4) Następną sytuacją, gdy konwersja zajdzie niejawnie są **wyrażenia:**

```
// obj - jest obiektem klasy A

if(obj) ...
switch(obj) ...
while(obj) ...
for(... ; obj ; ...)
```

W każdym wypadku kompilator będzie chciał określić wartość

```
(int)obiekt
```

Jeśli tego nie zdefiniujemy, to kompilator zaprotestuje przy tak zapisanych instrukcjach. Jeśli funkcję konwertującą

```
A::operator int() ;
```

zdefiniujemy, to będzie ona niejawnie użyta.
Ponieważ w naszej klasie `zespol` mamy ten operator, więc możemy stosować takie zapisy

```
zespol zzz(10, 0.3)

if(zzz)...
```

5) Następnym wypadkiem, gdy zachodzi niejawna konwersja typu, to **wyrażenia inicjalizujące**. Oto przykład:

```
zespol zzz (10.5, 3) ;
int k = zzz ; // tu niejawne wywołanie konwersji zespol na int
```

Podczas inicjalizacji obiektu typu `int` obiekt klasy `zespol` zostanie poddany konwersji na typ `int`. W rezultacie, jak wiemy, zamieniamy go na liczbę 10. Tą wartością jest inicjalizowany obiekt o nazwie `k`.

17.6 Zapis jawnego wywołania konwersji typów

Do tej pory zachwycaliśmy się tym, jak to wspaniale, że konwersje zachodzą niejawnie (czyli samoczynnie). Teraz porozmawiajmy o sytuacji, kiedy sami chcemy spowodować konwersję - czyli gdy jawnie ją wywołujemy. Konwersja

jawna za pomocą funkcji konwertującej może mieć dwie formy zapisu – formę rzutu lub formę wywołania funkcji.

Przykładowo: jeśli mamy klasę `zespol`, a jest określona konwersja z typu `zespol` na typ `inny`, to dopuszczalne są 2 formy zapisu takiej wymuszonej (jawnej) konwersji

```
zespol zzz(1, 1) ;
inny n ;
     n = inny(zzz) ;          // forma funkcji
     n = (inny)zzz ;          // forma rzutowania
```

Obie te formy robią to samo, jednak nie są całkowicie identyczne. W większości wypadków można posłużyć się oboma zapisami. Są jednak wypadki, gdy jeden z zapisów jest preferowany.

17.6.1 Advocatus zapisu przypominającego: „wywołanie funkcji"

Załóżmy, że konwersja jest dokonywana w klasie `inny` za pomocą konstruktora konwertującego o takiej deklaracji

```
inny::inny(zespol z, int i = 16) ;
```

(Jest to konstruktor konwertujący, bo skoro drugi argument jest domniemany to da się ten konstruktor wywołać z jednym tylko argumentem typu `zespol`).

Mamy więc ten kostruktor i, jak na razie, obie formy zapisu konwersji są dopuszczalne. Ale jeśli zapragniemy takiego jawnego wywołania, w którym dodatkowo jest obecny drugi argument konstruktora

```
n = inny(zzz, 333) ;
```

To oczywiście możliwa jest tylko ta forma (wywołanie funkcji).

> W drugiej formie – tej w stylu rzutowania – nie da się przesłać tego drugiego argumentu.

Zatem w tej sytuacji jest tylko jedna możliwość.

Zwracam jednak uwagę: to tylko w jawnym wywołaniu tego konstruktora mogliśmy sobie dodatkowo określić ten drugi – dotychczas domniemany - argument. W wywołaniu niejawnym - czyli konwersji „samoczynnej" - jest to niemożliwe.

17.6.2 Advocatus zapisu: „rzutowanie"

Czasem jednak preferowana jest forma rzutu, szczególnie tam, gdzie typ, na który zamieniamy, ma zbyt skomplikowaną składnię.

Załóżmy, że mamy klasę `A`, która ma funkcję konwertującą na typ `char*`. Deklaracja tej funkcji konwertującej wygląda następująco:

```
A::operator char*() ;          // !!!
```

Jawna konwersja w formie rzutowania wygląda tak:

```
A obj ;
char *napis ;

     napis = (char *)obj ;
```

Natomiast forma funkcji byłaby nielegalna

```
     napis = char*(obj) ;          // !!!
```

Łatwo się zorientować dlaczego – forma jest tak dziwna, iż kompilator raczej podejrzewa, że się pomyliliśmy.

Jest jednak wyjście: jeśli instrukcją `typedef` dla typu `char*` zdefiniujemy sobie inną nazwę

```
typedef char* tekst ;
```

to zapis funkcyjny, z użyciem tej nowej nazwy (typu `char*`), jest nadal możliwy

```
napis = tekst(obj);
```

17.7 Niecałkiem dobrane małżeństwa, czyli konwersje przy dopasowaniu

Do tej pory mówiliśmy o sytuacji idealnej:

- Mamy typ `Y` tam, gdzie potrzebny jest typ `X`. Skoro istnieje operator zamieniający obiekt klasy `Y` na obiekt klasy `X` – kompilator stosuje (niejawnie) ten operator i sprawa załatwiona.

Nie zawsze jednak jest to tak proste. Mimo to, nawet w sytuacjach niezupełnego dopasowania, może także zachodzić konwersja.

Aby mieć wyczucie, według jakich zasad się to odbywa, rozważmy kilka sytuacji:

Problem 1

Są dwie przykładowe funkcje – o przeładowanej nazwie `fun`

```
fun(int );
fun(X) ;
```

istnieje też operator zamieniający obiekt klasy `X` na typ `int`.

```
X::operator int();
```

Pytanie: jeśli kompilator napotka takie wywołanie funkcji

```
X obj ;
     fun(obj)
```

to która z powyższych wersji funkcji `fun` zostanie wywołana?

Odpowiedź brzmi: wywołana zostane funkcja

```
fun(X) ;
```

dlatego, że pasowała ona dokładnie do wywołania – bez konieczności stosowania jakiejkolwiek konwersji. Druga wersja `fun(X)` wymagałaby użycia niejawnej konwersji.

Problem 2

Mamy funkcję

```
fun(float);
```

oraz istnieje operator zamieniający obiekt klasy `X` na liczbę typu `int`.

```
X::operator int();
```

Pytanie: czy poniższe wywołanie funkcji jest legalne?

```
X obj ;
     fun(obj) ;
```

Tak, bo najpierw zostanie zastosowany operator konwersji zamieniający `obj` na liczbę typu `int`, a następnie za pomocą konwersji standardowych zostanie to zamienione na typ `float`.

Jak widzimy – nastąpiło tu jakby „kaskadowe" (2 - krotne) zastosowanie konwersji.

Ale uwaga:

> W takiej kaskadzie konwersja zdefiniowana przez użytkownika może się pojawić tylko jednokrotnie.

Innymi słowy nie jest możliwa taka sytuacja:

Mamy trzy klasy `X`, `Y`, `Z`. Zdefiniowaliśmy konwersję z typu `X` na typ `Y`, a także z typu `Y` na typ `Z` .

To wcale nie oznacza, że zdefiniowaliśmy konwersję z typu `X` na typ `Z`.
Zatem jeśli jest funkcja

```
fun(Z) ;
```

to nie można jej wywołać tak:

```
X objx;
fun(objx) ;
```

dlatego, że wymagałoby to dwukrotnego zastosowania konwersji zdefiniowanej przez użytkownika:

1) X ⟶ Y
2) Y ⟶ Z

Jak powiedzieliśmy jest to niedopuszczalne.

Zasada ta jest dla naszego dobra. Chodzi o to, by przy większej ilości zdefiniowanych konwersji po prostu nie pogubić się. Wyobraź sobie, co by było, gdyby konwersje wykonywały się niejawnie jak rząd padających kostek domina.

Czy zatem wywołanie funkcji `fun` jest dla obiektu `X` absolutnie niedopuszczalne?

Da się to zrobić specyfikując jawnie pierwszą konwersję. Wtedy niejawnie zajdzie już tylko jedna – a to jest poprawne

```
fun( (Y)objx) ;
```

Nic w tym dziwnego – wyrażenie będące argumentem funkcji jest wówczas typu `Y`. Dopiero to podlega niejawnej konwersji. W niejawnej części tego procesu występuje już tylko jedna konwersja zdefiniowana przez użytkownika.

Zapamiętaj – zasada ta dotyczy konwersji niejawnych. Jeśli natomiast robisz konwersję jawną – to możesz sobie łączyć dowolnie długie kaskady.

Problem 3

Mamy przeładowaną funkcję `fun`

```
fun(float);
fun(X) ;
```

oraz zdefiniowany sposób konwersji

`int` ⟶ `X`

Pytanie: Która z powyższych funkcji zostanie wywołana przy takim zapisie

```
fun(1) ;
```

Odpowiedź: Zadziała funkcja `fun(float)` dlatego, że wystarczyła standardowa konwersja

`int` ⟶ `float`

Konwersje zdefiniowane przez użytkownika uruchamiane zostają tylko wtedy, gdy w żaden inny sposób nie da się dopasować wywołania do funkcji.

Inaczej mówiąc:

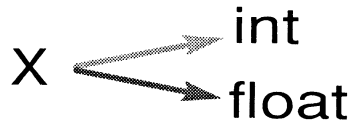
> Konwersje zdefiniowane przez użytkownika są używane niejawnie **w dodatku** do konwersji standardowych.

Problem 4

Wiemy, że konwersja zajdzie niejawnie tylko wtedy, gdy nie ma wieloznaczności. Załóżmy więc, że mamy dwie przeładowane funkcje

```
fun(int)
fun(float);
```

oraz sposób konwersji z typu `X` na typ `int`, a także sposób konwersji zamieniający typ `X` na `float`



Pytanie: Przy takim zapisie

```
X objx ;
     fun(objx) ;
```

która wersja funkcji `fun` zostanie wywołana?

Odpowiedź: **Żadna.** Jest to klasyczny przypadek dwuznaczności. Obie konwersje zamieniają typ obiektu na typ pasujący dokładnie do jednej z funkcji. Kompilator jest w rozterce, bo nie wie, którą z konwersji wybrać. Z tej rozterki wynika komunikat o błędzie.

Problem 5

Przypadek podobny, jak poprzednio, z tym, że teraz funkcje są takie

```
fun(long) ;
fun(float) ;
```

Teraz dwuznaczności nie ma. Jest bowiem jeden operator konwersji, który zamieni obiekt klasy `X` na obiekt typu dokładnie pasującego do funkcji

```
fun(float) ;
```

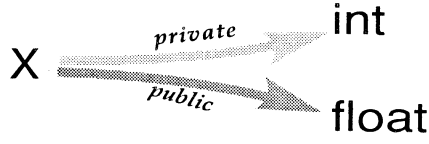
Natomiast drugi operator zamienia obiekt klasy `X` na obiekt `int`, po czym musi zostać jeszcze wykonana standardowa konwersja `int` na `long`.

> Ta druga droga jest wyraźnie dłuższa, więc nie ma wątpliwości (wieloznaczności), którą należy wybrać.

Problem 6

Klasa X ma dwa operatory konwersji

```
X::operator int() ;                    // (private)
X::operator float() ;                  // (public)
```



Pierwszy z nich mieści się w części prywatnej, a drugi w części publicznej klasy. Mamy też dwie funkcje

```
fun(int) ;
fun(float) ;
```

Czy następujące wywołanie jest legalne?

```
X objx ;
     fun(objx) ;
```

Odpowiedź: Nie, nie jest legalne. Występuje dwuznaczność. Mimo, że operator `int()` znajduje się w części prywatnej klasy. To dlatego, że

> Zawsze najpierw rozsądza się sprawę dwuznaczności, a dopiero potem sprawdza się dostęp.

Zasadę tę łatwo zrozumieć, jeśli się pamięta, że nazwa składnika `private` jest z zewnątrz widzialna tyle, że niedostępna.

Problem 7

Mamy dwie klasy `X` oraz `Y`. Konwersja z typu `X` na `Y` jest zrealizowana przez:

a) funkcję konwertującą w klasie `X`,

b) konstruktor konwertujący w klasie `Y`.

Czyli są dwa jednakowo dobre sposoby zamiany typu `X` na `Y`. Pytanie: czy to już jest błędem?

Odpowiedź: Jeszcze nie. Błąd nastąpi wtedy, gdy gdzieś potrzebna będzie niejawna konwersja z typu `X` na `Y`. (O wypadkach kiedy taka niejawna konwersja następuje – mówiliśmy niedawno).

To dopiero wtedy okaże się, że jest dwuznaczność.

Błąd nastąpi też w wypadku jawnej konwersji. Albowiem oba zapisy

```
X objx ; Y obj ;
     obj = (Y)objx ;
     obj = Y(objx) ;
```

nie precyzują, którą konkretnie drogą ma zostać zrealizowana konwersja. Jednakże zapis

```
obj = objx.operator Y() ;
```

już nie wywoła błędu, bo tu jest jasne, że chodzi o wybranie tego sposobu z funkcją konwertującą.

Problem 8

Mamy dwie klasy `X` i `Y`, a mają one wzajemnie zdefiniowane konwersje tak, że obiekt jednej klasy można zamienić na drugi.

```
X ──────────→ Y
Y ──────────→ X
```

Czy to błąd? Nie, nie jest to błąd, bo nie ma dwuznaczności
Oto dwie funkcje:

```
fun_dlaX(X) ;
fun_dlaY(Y) ;
```

Mając zdefiniowane wspomniane konwersje sprawiamy, że poniższe instrukcje są poprawne

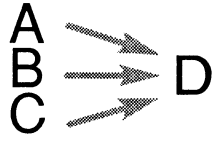
```
X objx ;          // definicje obiektów do ćwiczeń
Y objy ;

fun_dlaX(objx);                 // O.K.
fun_dlaX(objy);                 // konwersja Y ———→ X

fun_dlaY(objx);                 // konwersja X ———→ Y
fun_dlaY(objx);                 // O.K.

objx = objy ;                   // konwersja Y ———→ X
objy = objx ;                   // konwersja X ———→ Y
```

17.8 Kilka rad dotyczących konwersji

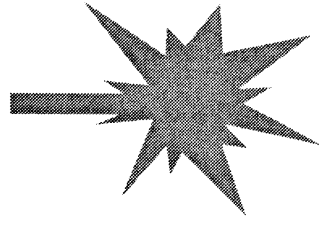
Konwersje są po to, by życie ułatwiać, a nie utrudniać. Niestety można i tutaj sprawę zagmatwać.

Co robić by konwersje nie przysporzyły kłopotu? Trzeba trzymać się kilku zasad.

Korzystając z rysunkowego zapisu, poprawny schemat konwersji można przedstawić ogólnie tak:



a niepoprawny tak:



Pierwszy schemat nazywam „dorzecze", bo przypomina zasadę, według której płyną zwykle rzeki.

Drugi schemat moża by nazwać „eksplozją" – sam chyba czujesz dlaczego.

Teraz omówmy to w szczegółach

Nie powinno się mnożyć konwersji ponad potrzebę.

Przy schemacie „eksplozja" jeśli są trzy przeładowane funkcje

```
fun(N) ;
fun(O) ;
fun(P) ;
```

to mimo tych istniejących konwersji nie możemy wywołać funkcji `fun` dla obiektu klasy M. To dlatego, że wszystkie trzy konwersje są jednakowo dobre do zrealizowania tego celu. Jednakowo dobre – czyli zachodzi wieloznaczność.

Natomiast przy schemacie „dorzecze" żadnego takiego ryzyka nie ma. Obiekt może zostać zamieniony na inny tylko w jeden sposób.

Z tego wynika taka rada:

Staraj się, by klasa miała tylko jeden operator konwersji.

Jeśli potrzebne będą Ci jeszcze inne konwersje dla tej klasy, to napisz sobie funkcję składową zajmującą się tym, ale nie dawaj jej nazwy

```
operator typ()
```

Tylko wybierz jakąś zwykłą nazwę dla tej funkcji. Zwykle nazwę *przypominającą* na jaki typ zamieniamy. Dzięki temu będzie to zwykła funkcja składowa, a nie operator konwersji. Ty sam będziesz mógł posługiwać się nią do woli, natomiast nie będzie ona brana pod uwagę w niejawnych konwersjach robionych automatycznie przez kompilator.

Operator konwersji między jedną klasą, a drugą powinien zamieniać w stronę obiektu o prostszej strukturze.

To dlatego, że operator ten nie ma sam niczego wymyślać. Znowu spójrz na nasz schemat „dorzecze": Konwersja następuje z obiektów klas `A`, `B`, `C` na obiekt klasy `D`, który jest zapewne tym, co klasy `A`, `B`, `C` mają ze sobą wspólnego.

Natomiast jeśli konwersja miałaby następować w stronę obiektu bardziej skomplikowanego – czyli na przykład z obiektu o 4 składnikach na obiekt o 444 składnikach, to kto ma wymyślić treść tych nowych 440 składników? Operator konwersji? Oczywiście mógłby, ale czy to mądre?

Nie każda konwersja ma sens

Pamiętasz naszą klasę `osoba` do spisywania danych o ludziach? Czy wpadłbyś na to, by obiekt klasy `osoba` zamieniać na obiekt klasy `zespol`? (liczba zespolona). Skoro taka zamiana nie ma specjalnego sensu, to nie należy również na siłę definiować operatora takiej konwersji.

18 Przeładowanie operatorów

Mówiąc najkrócej w rozdziale tym zajmiemy się tym, jak sprawić by znaczki takie jak +, –, itd. pracowały dla nas i robiły to, co my im każemy.

Wielokrotnie próbowałem Cię przekonać, że typy zdefiniowane przez użytkownika są tak samo władne jak typy wbudowane. Niestety tak dotąd nie było. Zauważ prostotę zapisu

```
int  a = 5,
     b = 7,
     c ;

          c = a + b ;
```

Mamy tu trzy obiekty typu wbudowanego `int`. Aby dwa z tych obiektów dodać posługujemy się po prostu znaczkiem + . Inaczej mówiąc: operatorem +.

Niestety, gdy niedawno mieliśmy do czynienia z typem zdefiniowanym przez nas, a reprezentującym liczby zespolone, to dodawanie takich dwóch obiektów zapisywaliśmy następująco

```
zespol    z(10, 5),
          x(7, 0),
          w ;

               w = dodaj(z, x) ;
```

Cóż, jeśli porównać to z zapisem `c=a+b`, to porównanie nie wypada na korzyść klas.

Zastanówmy się jednak jak to jest: przecież sam znaczek + nie dodaje dwóch liczb. Kiedy on występuje w tekście programu, kompilator wywołuje specjalną funkcję, która zajmuje się dodawaniem liczb. Nazwijmy tę funkcję **operatorem dodawania**. Jeśli kompilator po obu stronach znaczka + widzi liczby typu `int`,

to uruchamia taką funkcję. Argumentami są tu te dwie liczby typu `int` stojące po obu stronach znaku + .

Łatwo się też domyślić, że w wypadku, gdy obok znaczka + stoją liczby typu `float`, – kompilator uruchamia inną wersję tego operatora dodawania. To oczywiste, przecież w maszynie takie liczby są inaczej kodowane, więc i dodaje się je inaczej.

Zatem istnieje inna wersja funkcji „operator dodawania" – dla innych argumentów. Zaraz, zaraz- to przecież już znamy – przecież to przeładowanie funkcji: może być kilka wersji funkcji o tej samej nazwie – byle różniły się argumentami.

A teraz uwaga – będzie najważniejsze:

> Możesz sam napisać swoją funkcję „operator dodawania", która będzie wywoływana wtedy, gdy koło znaczka + pojawią się argumenty wybranego przez Ciebie typu. Po prostu po raz kolejny zostanie przeładowany operator +

Już wiem, co sobie teraz pomyślałeś: „–Tak, tak, słyszałem, że istnieją podobno ludzie, którzy rozkręcają komputer, przelutowują mu tranzystory albo piszą fragmenty systemu operacyjnego. Ja jednak nie jestem jeszcze na tym etapie!"

Mylisz się. Nie potrzeba tu żadnej specjalnej wiedzy. Jest to taka sobie zwykła funkcja. Co jest wobec tego w niej specjalnego? Nic, poza nazwą. Funkcja musi się nazywać `operator+` i jako jeden z argumentów przyjmować obiekt wybranej klasy.

```
operator+(X, Y) ;
```

Poza tym jest to najzwyklejsza w świecie funkcja, która nawet wcale nie musi dodawać – może po prostu tylko zagwizdać.

Najważniejsze jest jednak to, że gdy koło obiektu naszej klasy wystąpi znak + , to funkcja zostanie uruchomiana automatycznie

```
arg1 + arg2
```

Nie napisałem „uruchomiona niejawnie", bo w końcu jest tu w zapisie ten znak +, więc sprawa jest jawna.

Funkcję tę można oczywiście wywołać też jak zwykłą funkcję

```
operator+(arg1, arg2)
```

no ale po co? Do tego nie potrzeba nazwy `operator+` . To już przecież przerabialiśmy pisząc niedawno funkcję `dodaj`.

Pokażmy teraz przykład

Skoro narzekaliśmy na funkcję `dodaj` pracującą na liczbach zespolonych (klasa `zespol`), to pokażemy teraz jak dodawać je za pomocą operatora dodawania.

Nasz operator dodawania liczb zespolonych ma taką definicję:

```
zespol operator+(zespol a, zespol b)
{
zespol suma ;
     suma.rzeczyw = a.rzeczyw + b.rzeczyw ;
     suma.urojon = a.urojon + b.urojon ;
```

```
        return suma ;
}
```

Dzięki tak zdefiniowanemu operatorowi dodawania, możemy teraz wykonywać działania

```
zespol a(1, 0 ) ,
      b(6.3, 7.8) ,
      c ;

      c = a + b ;     // automatycznie pracuje nasz operator
```

Teraz pozwól, że przypomnę jak wyglądała funkcja `dodaj` z poprzedniego rozdziału

```
zespol dodaj(zespol a, zespol b)
{
zespol suma ;
     suma.rzeczyw = a.rzeczyw + b.rzeczyw ;
     suma.urojon = a.urojon + b.urojon ;
     return suma ;
}
```

Co widzimy? – Tak! Te dwie funkcje są identyczne! Jedyne, co je odróżnia, to nazwa. Teraz już mi chyba wierzysz, że aby przeładować operator+ nie trzeba rozkręcać komputera, ani wgryzać się w jego system operacyjny.

Jak już powiedziałem, `operator+()` wcale nie musi służyć do dodawania. Równie dobrze mógłby zagwizdać głośniczkiem komputera. Tyle, że ten gwizd rozlegnie się wtedy, gdy w programie obok znaku + znajdą się argumenty klasy `zespol`.

Zatem wiemy już, że można przeładowywać operator+. Kiedy to zrobić? Oczywiście w sytuacji, gdy wobec obiektu danej klasy bardziej naturalne wydaje się użycie znaku + niż wywołanie funkcji. Jest to przesłanka, by rozważyć możliwość przeładowania. Najczęściej tak się dzieje, wobec obiektów, których klasy mają jakieś powinowactwo z matematyką. Ale niekoniecznie – można przecież zastosować zapis

```
ekran + okienko
```

po to, by na ekranie pojawiło się to okienko.

Przeładowanie operatora nie jest obowiązkowe. Można sobie dobrze radzić nie używając go. Jednakże dobrze obmyślony system przeładowania operatorów dla jakiejś klasy sprawia, że posługiwanie się nią wydaje się tak naturalne, jakbyśmy mieli do czynienia z typem wbudowanym. Program jest czytelniejszy i krótszy.

18.1 Przeładowanie operatorów – definicja i trochę teorii

W poprzednim paragrafie zdobyliśmy ogólne pojęcie o przeładowaniu. Teraz podejdźmy do tego precyzyjniej.

Projektując klasę możemy zadbać o to, by pewne operacje na obiektach tej klasy wykonywane były na zasadzie przeładowania operatorów.

Przeładowania operatora dokonuje się definiując swoją własną funkcję która:

- nazywa się `operator@` – gdzie @ oznacza symbol operatora, o którego przeładowanie nam chodzi (np. `+`, `-`, `*`, `&`, itd.)
- jako co najmniej jeden z argumentów, przyjmuje obiekt danej klasy. (Musi być obiekt, nie może być wskaźnik do obiektu).

W sumie więc definicja takiego operatora wygląda tak

```
typ_zwracany operator@( argumenty)
{
        // ciało funkcji
}
```

Do przeładowywania mamy do dyspozycji bardzo wiele operatorów — wybieramy z nich te, które rzeczywiście będą nam potrzebne. Nie ma sensu przeładowywać wszystkich.

Oto lista operatorów, które mogą być przeładowane:

```
+    -    *    /    %    ^    &    |    ~
!    =    <    >    +=   -=   *=   /=   %=
^=   &=   |=   <<   >>   >>=  <<=  ==   !=
<=   >=   &&   ||   ++   --   ,    ->*  ->
new  delete  ()   []
```

Operatory `&` , `*` , `-` , `+` mogą być przeładowywane zarówno w swojej wersji jedno- jak i dwuargumentowej.

Dwa ostatnie operatory to: `()` – wywołanie funkcji, oraz `[]` - odniesienie się do elementu tablicy.

Natomiast nie mogą być przeładowane operatory

```
.    .*    ::    ?:
```

To dlatego, że przeładowanie polega na nadaniu symbolowi (np. `+`) specjalnego znaczenia, które ma on w momencie, gdy stoi on obok obiektu jakiejś klasy. Tymczasem wyżej wymienione operatory już mają bardzo ważne znaczenie wobec obiektów każdej klasy

`.` – tak się odnosimy do składnika klasy
`.*` – tak wybieramy składnik wskaźnikiem
`::` – tak określamy zakres

Nie możemy tych bardzo ważnych znaczeń niszczyć.

Ostatniego operatora `?:` nie przeładowujemy, bo – jak twierdzą Bjarne S. i Margaret E. – po prostu szkoda sobie tym zawracać głowy.

Domyślam się co sobie pomyślałeś patrząc na listę operatorów :

„– No dobrze, możliwe, że przeładuję kiedyś operator `+` , `-` , `*` , `/` ale bez przesady! Nie będę przecież przeładowywał operatora `()` lub `[ ]`."
Identycznie pomyślałem sobie, gdy uczyłem się języka C++. Tymczasem właśnie pierwsze moje przeładowanie zmuszony byłem zrobić w stosunku do operatora [ ]

Problem był taki. Miałem do czynienia z dużą tablicą.

```
int tablica[4096][4096] ;
```

Taka tablica nie mieści się przeważnie w pamięci operacyjnej. Na komputerze, w którym typ `int` ma rozmiar 2 bajtów wymaga ona 32 megabajty pamięci.

Wobec tego tablicę trzeba trzymać zapisaną na dysku w postaci jednego lub wielu plików. Jeśli potrzebujemy informacji z dowolnego elementu tej tablicy, to trzeba go odczytać z dysku. To właśnie zrobiłem. Napisałem funkcję `operator[]`, która sięgała na dysk, szukała tam żądanego elementu, po czym sprowadzała mi go do pamięci.

Nie pokażę tutaj realizacji tego przykładu, bo o operacjach z dyskiem rozmawiać będziemy dopiero w rozdziale o operacjach wejścia/wyjścia. Tam zilustruję to przykładem (str. 665). Tutaj ważna jest tylko idea: oto przeładowanie operatora [ ] pozwoliło mi używać zapisu

```
duza_tablica t ;

    t[7][4] = t[33][4] + 5 ;
```

Wierz mi, to naprawdę rewelacyjne. W tych klamrach udało mi się ukryć wszystkie operacje dyskowe otwierania pliku, czytania go, zapisywania do niego. To bardzo ułatwiło sprawę.

Wróćmy jednak do naszej listy operatorów. Oto kilka uwag:

Przeładować można te operatory i tylko te. Nie można wymyślać swoich.

Przeładowanie może nadać operatorowi dowolne znaczenie, nie ma też ograniczeń co do wartości zwracanej przez operator (wyjątkiem są operatory `new` i `delete`).

Nie można jednak zmienić priorytetu wykonywania tych operatorów.
Wyrażenie

```
a + b * c
```

zawsze będzie wykonywane jako

```
a + (b * c)
```

a nie jako

```
(a + b) * c
```

i to niezależnie od tego, czy operator * zajmuje się mnożeniem, czy gwizdaniem.

Nie można też zmienić „argumentowości" operatorów, czyli tego, czy operator jest jedno-, czy dwuargumentowy.

Przykładowo: operator / (dzielenie) musi być zawsze dwuargumentowy, niezależnie od tego, czym się zajmuje. Obok niego muszą stać zawsze dwa argumenty

```
obiektA /obiektB
```

natomiast nie są poprawne takie wyrażenia

```
/ obiektA
obiektA /
```

Podobnie operator ! (negacja) jest zawsze jednoargumentowy. Nie można więc napisać wyrażenia

```
obiektA ! obiektB
```

dopuszczalne jest tylko wyrażenie typu

```
! obiektA
```

Nie można też zmienić łączności operatorów – czyli tego, czy operator łączy się z argumentem z lewej, czy z prawej strony.

Dzięki temu, że operator = (przypisanie) jest prawostronnie łączny, wyrażenie

```
a = b = c = d ;
```

odpowiada wyrażeniu

```
a = (b = (c = d))) ;
```

i niezależnie jak ten operator przeładujemy taka prawostronna łączność nadal będzie zachowana.

Jeśli funkcja operatorowa jest zdefiniowana jako zwykła (globalna) funkcja, to przyjmuje tyle argumentów na ilu pracuje operator. Skoro dzielenie pracuje na 2 argumentach

```
klasaA arg1 ;
klasaB arg2

     arg1 / arg2
```

to funkcja `operator/` ma mieć dwa argumenty:

- pierwszy argument ma typ – taki, jak obiekt stojący po lewej stronie operatora /
- drugi argument – typu takiego, jak obiekt stojący po prawej stronie

```
operator/(kasaA, klasaB) ;          // deklaracja funkcji: operator/
```

Przynajmniej jeden z tych argumentów musi być typu zdefiniowanego przez użytkownika. Nie ma znaczenia który.

Oczywiście jest jasne, że argumenty nie mogą być domniemane:
Kiedy używasz zapisu

```
arg1 / arg2
```

to musisz po obu stronach symbolu / te oba argumenty napisać - nie możesz kazać kompilatorowi się ich domyślać.

Ten sam operator można przeładować wielokrotnie, z tym, że na okoliczność pracy z innym zestawem argumentów. To też już znamy: można przecież tworzyć bardzo wiele wersji funkcji przeładowanej – pod warunkiem, że będą się one różniły kolejnością lub typem argumentów.

W definicji operatora funkcji jest zastrzeżenie o tym, że przynajmniej jeden z argumentów operatora musi być typu zdefiniowanego przez użytkownika. Zapytasz pewnie: „Czy oznacza to, że nie można napisać funkcji operator+, która będzie przyjmowała oba argumenty będące typami wbudowanymi?"

```
operator+(int, float) ;
```

Nie, nie można tak przeładować. Przeładowanie służy tylko do usprawnienia pracy z klasami. Nie można zdefiniować funkcji operatorowej pracującej na samych typach wbudowanych.

Jeśli chcesz zapamiętać dlaczego, to pomyśl: skoro operacja

```
2 + 3
```

jest wykonywana przez komputer, to znaczy, że gdzieś już jest operator

```
int operator+(int, int)
```

Nie możesz więc tak przeładowywać, bo ten zestaw argumentów jest już zajęty[†)].

Przeładowywać więc tylko dla typów wbudowanych nie można. Z drugiej strony nie ma czego żałować. Nie zamierzasz chyba sam pisać operatorów na dodawanie liczb całkowitych!

18.2 Moje zabawki

Przedstawię teraz klasę, z którą w mojej praktyce przychodzi mi pracować najczęściej.

†) Trochę to moje tłumaczenie jest naciągane, przepraszam.

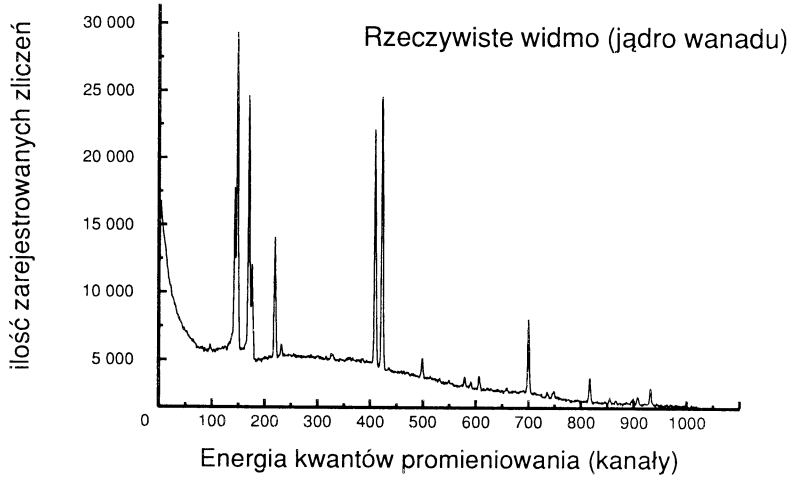
W laboratoryjnych zastosowaniach komputerów bardzo często mamy taką sytuację, że komputer steruje pomiarami. Rezultatem takiego pomiaru jest zwykle tablica liczb.

W elementach tej tablicy liczb mogą być zanotowane dane o ilości wyprodukowanych w jednostce czasu nowych komórek tkanki, może być to tablica, w której są zapisane temperatury poszczególnych dni w roku.

W moim wypadku jest to najczęściej tablica, w której poszczególnych elementach zapisane są energie kwantów promieniowania. (Te energie mierzymy w kilo elektronowoltach keV, ale to jest teraz nieistotne).

Zasada pomiaru jest taka: jeśli układ pomiarowy zarejestrował kwant promieniowania o energii 511, to w tablicy element nr 511 zwiększa się o jeden. Czyli jeśli do tej pory była tam wartość 6, to teraz będzie 7.

Jeśli w naszym pomiarze zarejestrowaliśmy 1000 takich kwantów, to w rezultacie w tablicy będą zapisane same zera, a tylko w elemencie o numerze 511 zapisana będzie liczba 1000. Te elementy tablicy zwyczajowo nazywa się kanałami.



Jeśli więc zarejestrowaliśmy 100 kwantów o energii 511 oraz 200 kwantów o energii 350, to zera będą we wszystkich kanałach z wyjątkiem kanału 511, gdzie będzie 100 i kanału 350 – gdzie będzie liczba 200

```
kanal[511]          200
kanal[350]          100
```

Ten typ danych pomiarowych nazywa się widmem. Zauważyłeś co powiedziałem: ... ten **typ** .... Nic więc dziwnego, że definiuję go jako klasę

```
class widmo
{
public:
     int kanal[1024] ;
     //...
};
```

Klasa ta, oprócz tego, że zawiera tablicę, zawiera też funkcje składowe. Niektórymi z tych funkcji mogą być przeładowane operatory. To dlatego klasą tą posłużę się w przykładach do tego rozdziału.

18.3 Funkcja operatorowa jako funkcja składowa

Funkcja operatorowa może być zwykłą, globalną funkcją, a może być także funkcją składową klasy.

Załóżmy, że w wypadku klasy `widmo` zachodzi konieczność dodawania do wszystkich elementów tablicy `widmo` jakiejś liczby `int`. Oto definicja takiej funkcji jako funkcji globalnej:

```
widmo operator+(widmo dane, int liczba)
{
widmo rezultat ;
     for(int i = 0 ; i < 128 ; i++)
          rezultat.kanal[i] = dane.kanal[i] + liczba ;
     return rezultat ;
}
```

Działanie funkcji jest proste – w jej ciele definiujemy obiekt klasy `widmo`, który posłuży nam do przechowania rezultatu operacji. Następnie do poszczególnych kanałów widma `dane` dodajemy `liczbe`. Wynikowe widmo `rezultat` zwracane jest jako rezultat działania funkcji.

Teraz zobaczmy, jak wygląda realizacja tego samego operatora dodawania, jeśli zostanie on zdefiniowany jako funkcja składowa klasy `widmo`.

```
widmo widmo::operator+(int liczba)
{
widmo rezultat ;
     for(int i = 0 ; i < 128 ; i++)
          rezultat.kanal[i] = kanal[i] + liczba ;
     return rezultat ;
}
```

Jak widzimy, teraz funkcja przyjmuje tylko drugi argument. Co się stało z pierwszym ? Skoro funkcja jest teraz funkcją składową klasy `widmo`, to znaczy, że jest wywoływana na rzecz jakiegoś obiektu klasy `widmo` - dostaje zatem wskaźnik do tego obiektu. Jest to wskaźnik `this`. Pierwszy argument przesyłany jest więc do funkcji dzięki wskaźnikowi `this`.

Najlepiej to zauważyć w zapisie jawnego wywołania tej funkcji. Jeśli mamy obiekt klasy widmo

```
widmo kobalt ;
```

to wyrażenie

```
kobalt + 5
```

jest równoważne jawnemu wywołaniu operatora dodawania

```
kobalt.operator+(5)
```

Skoro przy przekazywaniu argumentów bierze udział wskaźnik `this`, stąd jasne jest, że nie może być to funkcja składowa typu `static`, bo taka wskaźnika `this` nie ma. Jak pamiętamy – tego typu funkcje pracują nie dla konkretnych egzemplarzy obiektów, lecz dla klasy „w ogólności". Ponieważ nam chodzi o dodanie liczby do konkretnego egzemplarza obiektu klasy `widmo`, zatem o funkcji składowej typu `static` nie może być mowy.

> Dana funkcja operatorowa może być więc albo funkcją globalną, albo niestatyczną funkcją składową klasy, dla której pracuje.

Oczywiście: albo-albo! Nie możemy zdefiniować obu form tej funkcji. Obie formy bowiem pracują na tych samych argumentach

```
(widmo, int)
```

chociaż w drugim wypadku wydaje się to ukryte.

Ogólnie widać, że:

> Jeśli operator definiujemy jako funkcję składową, to ma ona zawsze o jeden argument mniej niż ta sama funkcja napisana w postaci funkcji globalnej. Ten „brakujący" argument – w wypadku funkcji składowej – spowodowany jest oczywiście istnieniem wskaźnika `this`.

Czas chyba na przykład działającego programu.

```
#include <iostream.h>

const int rozmiar = 1024 ;                                  // ❷
/////////////////////////////////////////////////////////////
class widmo
{
public:
     int kanal[rozmiar] ;                                   // ❶
     //———————————————— konstruktor
     widmo(int wart = 0) ;
     //———————————————— przeładowany operator
     widmo operator+(int ) ;
};
/*********************************************************/
widmo::widmo(int wart)                                      // ❸
{
     for(int i = 0 ; i < rozmiar ; i++)
          kanal[i] = wart ;
}
/*********************************************************/
widmo widmo::operator+(int liczba)
{
widmo rezultat ;
     for(int i = 0 ; i < rozmiar ; i++)
          rezultat.kanal[i] = kanal[i] + liczba ;           // ❹
     return rezultat ;
}
/*********************************************************/
main()
{
```

```
widmo kobalt(5) ;                                          // ❺
widmo nowe ;                                               // ❻
      nowe = kobalt + 100 ;                                // ❼

      cout << "Przykladowo patrzymy na na kanal 44. \n"
           "Widmo 'kobalt' ma tam : "
           << kobalt.kanal[44]
           << "\na w widmie 'nowe' jest tam : "
           << nowe.kanal[44] << endl ;

      nowe = nowe + 700 ;                                  // ❽

      cout <<"A teraz w kanale 44 obiektu 'nowe' jest : "
           << nowe.kanal[44] << endl ;
}
```

Po wykonaniu go na ekranie zobaczymy

```
Przykladowo patrzymy na na kanal 44.
Widmo 'kobalt' ma tam : 5
a w widmie 'nowe' jest tam : 105
A teraz w kanale 44 obiektu 'nowe' jest : 805
```

Komentarz

❶ Przypominam, że rozmiar definiowanej tablicy musi być stałą znaną kompilatorowi w trakcie kompilacji. To dlatego w definicji ❷ jest obowiązkowe słowo `const`.

❸ Definicja konstruktora. Ciało tego konstruktora to oczywiście absurd, bo dane będące treścią widma pochodzą zwykle z układu pomiarowego. Tutaj symulujemy je wypełniając całe widmo tą samą wartością. Zauważ w deklaracji tego konstruktora, że argument `wart` może być domniemany.

❹ Definicja operatora+ będącego funkcją składową klasy `widmo`. Zauważ, że zapis `kanal[i]` odpowiada zapisowi `this->kanal[i]`

❺ Definicja obiektu klasy `widmo`. Ten obiekt nazywa się `kobalt` i postanawiamy wypełnić go wartością 5.

❻ Definicja innego obiektu tej samej klasy. Zauważ, że teraz zadziała argument domniemany konstruktora. Obiekt `nowe` zostanie wypełniony wartością 0.

❼ **To jest najpiękniejsze miejsce tego programu.** Zapis ten sprawia, że rusza do pracy nasz operator dodawania. Piękna jest tu oczywiście prostota zapisu. Tę samą linijkę można by zapisać również tak

```
nowe = kobalt.operator+(100) ;
```

❽ Ta linijka jest jakby powtórzeniem powyższego. Inny sposób zapisania tego to

```
nowe = nowe.operator+(700) ;
```

Mam nadzieję, że przykład ten przekonał Cię, jak łatwo przeładowuje się operator.

18.4 Funkcja operatorowa nie musi być przyjacielem klasy

Wiemy już, że mamy dwa sposoby definicji tego samego operatora. Może być on

- funkcją składową klasy,
- zwykłą funkcją globalną.

Funkcja zwykła – jest naprawdę zwykła ; tylko w nazwie występuje słowo `operator`.

Tu chciałbym zwrócić uwagę, że ta zwykła funkcja nie musi mieć też specjalnych uprawnień w stosunku do klasy. Podkreślam to dlatego, że w wielu anglojęzycznych publikacjach uznaje się za oczywistość, że funkcja musi być zaprzyjaźniona z klasą, na której pracuje[†)].

Nie jest tak na szczęście. Gdyby tak było – nie moglibyśmy dopisać operatorów do klas, które ktoś inny dawniej napisał (i są na przykład w bibliotece). Skoro ten ktoś pisząc klasę nie umieścił tam deklaracji przyjaźni z naszą funkcją – to przepadło.

Jak mówię – nie jest tak, na szczęście. Można zdefiniować operatory dla klas już wcześniej napisanych. (Dlaczego to takie ważne – wrócimy do tego przy omawianiu przeładowania operatora <<).

Czy przyjaźń jest zatem niepotrzebna ?

Jeśli funkcja operatorowa – w skrócie mówić będziemy: operator – ma pracować tylko na publicznych składnikach klasy, to może być ona zwykłą funkcją składową, bez żadnych specjalnych uprawnień.

Tak jest z naszą klasą `widmo`. Celowo tablicę `kanal` uczyniliśmy tam publiczną.

Jeśli jednak chcemy, by operator mógł pracować także na niepublicznych składnikach klasy – wówczas klasa musi dać operatorowi *votum* zaufania – musi zadeklarować tę funkcję operatorową jako zaprzyjaźnioną.

Tak też dzieje się w większości wypadków. Operatory są

- – albo funkcjami *składowymi* klasy ; mają wtedy dostęp do składników prywatnych swojej klasy,
- – albo są funkcjami *zaprzyjaźnionymi* z klasą; przyjaźń ta daje im dostęp do składników prywatnych.

Chcę jednak, abyś wiedział, że przyjaźń nie jest obowiązkowa. To przecież oczywiste – jeśli operator ma np. zagwizdać na cześć obiektu danej klasy, to do tego nie jest mu potrzebny dostęp do jej prywatnych składników.

†) Tak pisze nawet sam Bjarne Stroustrup w swojej pierwszej książce.

18.5 Operatory predefiniowane

Jeśli wobec obiektu danej klasy użyjemy operatora, którego nie zdefiniowaliśmy, wówczas kompilator zasygnalizuje błąd. Po prostu dlatego, że nie wie, jak ma wtedy postąpić. To tak, jakbyśmy wywoływali funkcję, której nie zdefiniowaliśmy.

Jest jednak kilka operatorów, których znaczenie jest tak oczywiste, że zostają one automatycznie generowane dla każdej klasy

Te operatory to:

= przypisanie [podstawienie] do obiektu danej klasy,

& (jednoargumentowy) pobranie adresu obiektu danej klasy,

, (przecinek) – znaczenie identyczne jak dla typów wbudowanych,

`new`, `delete` – kreacja i likwidacja obiektów w zapasie pamięci.

Operatorem =, w jego predefiniowanej wersji, już się posługiwaliśmy w naszych przykładach. Instrukcja

```
nowy = kobalt + 5 ;
```

była możliwa dzięki operatorowi+ , ale także dzięki operatorowi = (przypisanie). Przypisanie odbywa się metodą „składnik po składniku". O tym, że nie zawsze nam to musi odpowiadać, porozmawiamy niebawem.

Jeśli chcemy, by operatory wykonały dla nas inną akcję niż predefiniowana, wówczas musimy zdefiniować swoją wersję tego operatora, a wówczas kompilator uzna tamtą predefiniowaną wersję za niebyłą. Natomiast nie ma eleganckiego sposobu, by zrezygnować z predefiniowanego znaczenia tych operatorów.

Słowem – nie można ich „rozdefiniować" tak, by nie znaczyły nic i by kompilator widząc taki operator zastosowany wobec obiektu danej klasy zaprotestował mówiąc, że nie wie jak się zachować.

Jedynym sposobem jest napisanie swoich wersji, które albo będą robiły coś sensownego, albo nie będą robiły nic (puste ciało funkcji operatorowej).

Oczywiście te automatycznie generowane operatory rzadko nam przeszkadzają. Najczęściej skwapliwie się godzimy na nie – czasem tylko zmieniając znaczenie operatora przypisania = . Ale o tym w swoim czasie.

18.6 Argumentowość operatorów

Operatory działają albo na jednym argumencie (jak np. negacja), albo na dwóch argumentach (jak np. dzielenie). Poza jednym wyjątkiem nie ma operatorów, które pracują na więcej niż dwóch argumentach. To znaczy: operator mnożenia * ma mieć dwa argumenty. Jeśli spróbowalibyśmy go zdefiniować jako funkcję z większą ilością argumentów, to kompilator zaprotestuje. Czyli niemożliwa jest np. taka deklaracja

```
widmo operator*(widmo,int,float,char,int,char *);    // !!
```

Jednym jedynym wyjątkiem, kiedy do operatora można przesłać większą liczbę argumentów jest `operator()` – [dwa nawiasy okrągłe]. Nic w tym dziwnego: operator ten nazywa się operatorem: „wywołanie funkcji".

O ile ze znakiem + kojarzą się dwa (i tylko dwa) argumenty stojące po jego obu stronach, o tyle ze znakiem `()` kojarzy się większa liczba argumentów, które wysyłane są do funkcji. Argumenty te oddzielone są od siebie przecinkami, ale wszystkie są argumentami wywołania funkcji. Dokładniej: „... są argumentami operatora wywołania funkcji". Temu ciekawemu operatorowi poświęcimy specjalny paragraf.

Teraz przyjrzyjmy się tym podstawowym grupom argumentów.

18.7 Operatory jednoargumentowe

Operatory te występują zwykle jako przedrostek (prefix), czyli stoją przed obiektem danej klasy. Oto przykłady – dla oswojenia pokazuję też ich odpowiedniki zastosowane dla obiektu typu `int`:

```
widmo wid ;
int i ;

            - wid              - i
            ! wid              ! i
            ++wid              ++i
            ~ wid              ~ i
            & wid              & i
```

Mogą być też jednoargumentowe operatory przyrostkowe (końcówka) (postfix). Stoją one za obiektem.

```
            wid ++             i++
            wid --             i--
```

Jeśli mamy daną klasę K, to operatory jednoargumentowe typu przedrostkowego pracujące na obiektach klasy K można zdefiniować jako

- nieskładową (zwykłą) funkcję wywoływaną z jednym argumentem (obiektem tej klasy K)

```
operator@(K) ;
```

albo

- jako funkcję składową klasy K wywoływaną bez żadnych argumentów

```
K::operator@(void) ;
```

Oto przykład:

Dotyczy on operatora jednoargumentowego '–' . Operator ten w stosunku do typu `int` oznacza liczbę przeciwną do danej. (Nie jest to odejmowanie – bo to jest dwuargumentowe). Przypomnijmy sobie

```
int a = 2 , b = -15

cout << (-a) ;
cout << (-b) ;
cout << a ;
```

Na skutek działania tego operatora w pierwszym wypadku na ekranie zostanie wypisana liczba -2, a w drugim wypadku liczba +15, czyli liczby przeciwne do oryginalnych `a` i `b`. W trzecim wypadku na ekran zostanie wypisana będąca w obiekcie `a` liczba 2 – na dowód, że treść samego obiektu nie uległa zmianie.

Pytanie: Co taki operator znaczy wobec naszej klasy `widmo`?

Jeszcze nic, wszystko zależy od nas – będzie znaczył to, co sobie postanowimy. Proponuję, żeby operator ten w rezultacie zastosowania wobec obiektu klasy `widmo` dawał taki obiekt klasy `widmo`, w którym we wszystkich kanałach będą liczby przeciwne.

Dobrze jest też zachować taką konsekwentną logikę: skoro w wypadku obiektów typu `int` operator nie zmieniał treści samego obiektu (po operacji zmienna `a` miała nadal wartość 2), to ten sam operator zastosowany wobec widma niech także niczego nie zmienia w samym widmie. Niech tylko zwróci jako wartość inne widmo, które będzie miało wartości we wszystkich kanałach zamienione na liczby przeciwne.

Jest to dobra rada na przyszłość:

> Staraj się by operator, który jest nieszkodliwy dla obiektów typu wbudowanego – był także nieszkodliwy dla obiektów Twojej klasy.
> Siłą rzeczy przenosi się przyzwyczajenia z typów wbudowanych na klasy. Łatwiej wtedy zresztą „czytać" takie wyrażenia.

Napiszmy teraz definicję tego operatora. Mamy dwie możliwości: może być on funkcją składową klasy `widmo` lub może być funkcją spoza tej klasy.

Zróbmy go jako funkcję nieskładową

```
widmo operator-(widmo zrodlo)
{
widmo rezultat ;
     for(int i = 0 ; i < rozmiar ; i++)
          rezultat.kanal[i] = - zrodlo.kanal[i] ;
     return rezultat ;
}
```

Przyjrzyjmy się tej funkcji. Przyjmuje ona jako argument obiekt klasy `widmo`. Przesłanie następuje przez wartość. W rezultacie w obrębie funkcji powstaje kopia o nazwie `zrodlo`. Widzimy też, że w funkcji definiowany jest obiekt lokalny o nazwie `rezultat`.

Następne linijki to przepisywanie jednego widma do drugiego. Przy okazji zmienia się znak wyrażeniu `(zrodlo.kanal[i])`.

Wyrażenie to ma – jak wiemy – typ `int`, zatem stojący przed nim operator – (minus) jest zwykłym operatorem liczby przeciwnej dla typów wbudowanych. („Nasz" pracuje tylko wtedy, gdy znak „–" stoi przed obiektem klasy `widmo`).

Ostatnia instrukcja `return` powoduje przesłanie (przez wartość) widma będącego rezultatem. Sam obiekt o nazwie `rezultat` (ponieważ jest automatyczny) przestaje istnieć.

Funkcję można by napisać o wiele sprytniej, jednak stracilibyśmy na czytelności zapisu, co wydaje się tu istotniejsze.

A oto jak używamy tego operatora w programie (zakładam, że mamy tu obiekty z naszego ostatniego przykładu):

```
// ...
nowy = (-kobalt) ;
```

W rezultacie w widmie `nowy` znajdzie się widmo, którego wszystkie kanały są liczbami przeciwnymi do treści kanałów widma `kobalt`.

Oto ten sam operator w wersji jako funkcja składowa klasy: `widmo`

```
widmo widmo::operator-()
{
widmo rezultat ;
     for(int i = 0 ; i < rozmiar ; i++)
          rezultat.kanal[i] = - kanal[i] ;
     return rezultat ;
}
```

Porównajmy oba warianty funkcji operatorowej

Oczywiście po pierwsze różnica jest w liczbie argumentów. W drugiej wersji nie ma przysłania żadnych argumentów. Funkcja – jako składowa – wywoływana jest na rzecz obiektu swej klasy `widmo`, zatem zapis

```
- kobalt ;
```

oznacza to samo co

```
kobalt.operator-() ;
```

Wskaźnik do obiektu `kobalt` został przesłany do wnętrza funkcji i nazywa się tam `this`. To, że ta funkcja składowa jest funkcją operatorową, to nic szczególnego. Jeśli jest tylko niestatyczna, to ma wskaźnik `this`. To tłumaczy dlaczego w miejscu, gdzie przedtem był zapis

```
rezultat.kanal[i] =     - zrodlo.kanal[i] ;
```

jest teraz

```
rezultat.kanal[i] =     - kanal[i] ;
```

Naprawdę jest tam przecież

```
rezultat.kanal[i] =     - (this->kanal[i]) ;
```

Pokazaliśmy przeładowanie operatora jednoargumentowego „–". Wszystkie inne jednoargumentowe operatory przedrostkowe (stojące przed nazwą obiektu) przeładowuje się podobnie.

Są jednak jeszcze jednoargumentowe operatory stojące **za** nazwą obiektu (że tak powiem: operatory 'końcówkowe', postfix). Są to operatory: postinkrementacji i postdekrementacji. Z nimi sprawa jest szczególna. Omówimy je w stosownym miejscu (str. 480).

18.8 Operatory dwuargumentowe

Operatory dwuargumentowe możemy także przeładowywać na dwa sposoby

- albo jako funkcję składową niestatyczną wywoływaną z jednym argumentem

  `x.operator@(y)`

- albo jako funkcję nieskładową (czyli zwykłą), wywoływaną z dwoma argumentami.

  `operator@(x, y)`

Taka funkcja operatorowa zostaje automatycznie wywołana, gdy obok znaczka danego operatora znajdą się dwa argumenty określonego przez nas typu

`x @ y`

Znowu jest: albo-albo. Nie można dla tego samego zestawu argumentów definiować i tak, i tak.

18.8.1 Przykład na przeładowanie operatora dwuargumentowego

Załóżmy, że chcemy przeładować operator mnożenia `*`. Dla odmiany weźmy klasę reprezentującą trójwymiarowy wektor. Trzeba się zastanowić co chcemy, by ten `operator*` z obiektem klasy `wektorek` dla nas robił. Niech, dajmy na to, służy do mnożenia wektora przez liczbę rzeczywistą. Jeśli jakiś wektor mnożymy przez 2, to wszystkie jego współrzędne mają zostać podwojone.

Oto realizacja tego operatora jako zwykłej, globalnej funkcji:

```
#include <iostream.h>
//////////////////////////////////////////////////////////
class wektorek {
public :
     float x, y, z ;
     //———— konstruktor
     wektorek(float xp = 0, float yp = 0, float zp = 0 )
                    : x(xp), y(yp), z(zp)              //❶
     { /* ciało puste */ } ;

     // ... inne funkcje składowe
} ;
```

```
/////////////////////////////////////////////////////////////
wektorek operator*(wektorek kopia, float liczba )    // ❷
{
wektorek rezultat ;

     rezultat.x = kopia.x * liczba ;
     rezultat.y = kopia.y * liczba ;
     rezultat.z = kopia.z * liczba ;
     return rezultat ;
}
/*********************************************************/
void pokaz(wektorek www) ;                    // deklaracja
/*********************************************************/
main()
{
wektorek  a(1,1,1) ,
          b(-15, -100, +1) ,
          c ;

          c = a * 6.66 ;                                  // ❸
          pokaz(c) ;

          c = b * -1.0 ;
          pokaz(c) ;
}
/*********************************************************/
void pokaz(wektorek www)
{
      cout << " x = " << www.x
           << " y = " << www.y
           << " z = " << www.z << endl ;
}
```

Po wykonaniu tego programu na ekranie pojawi się

```
x = 6.66 y = 6.66 z = 6.66
x = 15 y = 100 z = -1
```

Komentarz

❶ Dla skrócenia zapisu konstruktor nadaje składnikom wartości początkowe w liście inicjalizacyjnej.

❷ Definicja `operatora*` , gdy jest on funkcją globalną. Jak widać wywoływany jest on z dwoma argumentami.

❸ Przykładowe użycie tego operatora. Operator ten (jak można zobaczyć z jego definicji) wywoływany jest wtedy, gdy obok znaczka * po lewej stoi obiekt klasy `wektorek`, a po prawej obiekt typu `float`.

A oto realizacja tej samej funkcji operatorowej jako funkcji składowej klasy `wektorek`:

```
wektorek wektorek::operator*(float liczba )
{
```

```
wektorek rezultat ;

     rezultat.x = x * liczba ;
     rezultat.y = y * liczba ;
     rezultat.z = z * liczba ;

     return rezultat ;
}
```

Podstawową różnicą jest to, że teraz ta funkcja operatorowa wywoływana jest z jednym argumentem typu `float`. Jak wiadomo – informacja o obiekcie klasy `wektorek` przychodzi dzięki wskaźnikowi `this`.

Użycie w programie operatora w tej wersji jest identyczne jak poprzednio. Zapis jest ten sam.

18.8.2 Przemienność

Zauważ, że dzięki przeładowaniu, możemy teraz tworzyć wyrażenia

```
wektorekA = wektorekB * 11.1 ;
```

natomiast odwrócenie kolejności czynników iloczynu nie wchodzi w grę, czyli zapis

```
wektorekA = 11.1 * wektorekB ;
```

zostanie przez kompilator odrzucony jako błędny. Dlaczego?

Dlatego, że przecież zdefiniowaliśmy operator pracujący na argumentach

```
(wektorek, float)
```

ale nie zdefiniowaliśmy dla argumentów

```
(float, wektorek)
```

To oczywiście nie jest to samo. Jeśli chcemy, by stosowanie zapisu

```
6.26 * wektorekB
```

było możliwe, to musimy dodatkowo zdefiniować operator na taką okoliczność.

Oto jak wygląda realizacja tego operatora jako funkcji globalnej:

```
wektorek operator*(float liczba, wektorek kopia)
{
wektorek rezultat ;

     rezultat.x = kopia.x * liczba ;
     rezultat.y = kopia.y * liczba ;
     rezultat.z = kopia.z * liczba ;

     return rezultat ;
}
```

Argumenty formalne są oczywiście w odwrotnej kolejności natomiast ciało funkcji operatorowej jest takie samo. To zrozumiałe – chodzi nam przecież o to,

by ten operator robił to samo co tamten. Bardziej fachowo: chcemy by nasze mnożenie było przemienne.

A teraz zagadka: Jak wyglądałby ten operator jako funkcja składowa klasy?

Właśnie, to jest problem. Zauważyłeś już może, że jeśli funkcja operatorowa była funkcją składową klasy, to należała do tej klasy, do której należał jej pierwszy argument.

Przedtem pierwszym argumentem był `wektorek`, drugim `float` - więc operator był funkcją składową klasy `wektorek`.

Teraz pierwszym argumentem jest `float`, a `wektorek` jest drugim. Operator nie może być funkcją składową klasy `float` – bo takiej klasy nie ma – `float` to typ wbudowany.

Jeśli jeszcze nie jest to dla Ciebie jasne, to zauważ, jak wywoływalibyśmy jawnie funkcję operatorową w obu sytuacjach.

```
wektorek krotki(1,1,1), wynik ;
     wynik = krotki.operator*(3.16) ;
```

to jest w porządku, ale zapis

```
wynik = 3.16.operator*(krotki);
```

nie ma sensu. Nie można wywołać funkcji składowej `operator*()` na rzecz liczby 3.16

Widzisz więc, że wybór jednej z dwóch możliwości realizacji funkcji operatorowej może być istotny.

Funkcja operatorowa, która jest **funkcją składową klasy** - wymaga, aby obiekt stojący po lewej stronie (znaczka) operatora był obiektem jej klasy. Operator, który jest **zwykłą funkcją globalną** – nie ma tego ograniczenia.

Nie chciałbym jednak byś z tego wyciągał zbyt pochopne wnioski – iż wobec tego od dziś wszystkie przeładowania operatorów robisz za pomocą funkcji nieskładowych.

I jedna i druga forma ma swoje wady i zalety. O tym jednak, którą kiedy wybrać, porozmawiamy wtedy, gdy już poznamy wszystkie aspekty przeładowywania operatorów.

18.9 Przykład zupełnie niematematyczny

Z tego, co dotychczas mówiliśmy, możesz odnosić wrażenie, że przeładowanie operatorów, to coś związane z obliczeniami matematycznymi. Myśląc tak, czytelnicy, którzy używają komputera raczej do sterowania, niż do obliczeń, mogą w tym miejscu stracić zainteresowanie tym tematem, jako mało przydatnym. Dodatkowo czytelnicy, którzy z pewnych bardzo ważnych powodów życiowych mają wstręt do matematyki - też mogliby się w tym miejscu zniechęcić.

Dla Was to, moi drodzy, przygotowałem ten paragraf. Zobaczymy tu zastosowanie przeładowania operatorów do pracy z ekranem i pojawiającymi się na nim okienkami. Czyli coś, co nie ma nic wspólnego z matematyką.

Najpierw jednak wprowadzenie

Jak wiesz, dawno minęły czasy, gdy praca komputerem wyglądała tak, jak rozmowa z kimś przez dalekopis. Obecnie mamy do czynienia z monitorem ekranowym, na którym może być równocześnie kilka działających programów. Każdy z nich ma dla siebie pewną część ekranu - zwaną oknem. To tak, jakby program miał swój własny monitor, narysowany na ekranie. Na ekranie możemy mieć kilka takich okien - a jeśli są one duże, to mogą na siebie zachodzić zasłaniając się nieco.

Praca z tak oprogramowanym komputerem polega na tym, że najpierw uruchamiamy wybrane programy. Każdy z nich pojawia się na ekranie w przydzielonym mu oknie. Tak, jak przyniesione i położone na biurku rysunki. Oczywiście ostatnio położony rysunek może zasłaniać poprzednie.

Gdy chcemy zwrócić się do jednego z programów, to wskazujemy komputerowi, że chcemy rozmawiać z danym oknem. (Możemy to zrobić za pomocą myszki lub wciskając kombinację klawiszy). Wybrane odpowiednie okienko - możliwe, że fragmentami przysłonięte przez inne okna - komputer wyjmuje nam na sam wierzch. Tak, jak rysunek na biurku, częściowo przysłaniany przez dwa inne, wyjmujemy na wierzch i od tej pory to on przysłania inne. Możemy wówczas rozmawiać z danym programem. Jeśli przyjdzie czas wydania komend innemu programowi, to postępujemy podobnie.

Możemy też zdecydować się uruchomić jeszcze jakiś nowy program i wówczas komputer dołączy dodatkowe okno do istniejących na naszym ekranie.

Możemy też jakiś z działających programów zakończyć, wówczas jego okno zniknie. Program też może zakończyć się sam i wówczas okno też zniknie (choćby nawet było gdzieś pod spodem).

Jeśli chcielibyśmy napisać taki program obsługujący okienka na ekranie, to można operacje, o których mówiłem, zrealizować za pomocą przeładowania operatorów.

> Oczywiście, tak czy owak, musimy mieć w naszym programie jakieś funkcje dodające okienka, usuwające je, czy sprowadzające na pierwszy plan. Jednak zamiast wywoływać je w zwykły sposób - możemy posłużyć się symbolami. Będzie to możliwe jeśli tylko funkcje te będą zrealizowane jako przeładowane operatory.

Łatwo się domyślić, że w programie wystąpią takie obiekty, jak ekran czy okienka. To do pracy z nimi przeładujemy operatory. To znaczy, że to one będą argumentami operatorów.

Oto moje propozycje przeładowania:

```
ekran += okno1
```

❖ do ekranu, na którym coś już może jest, dodaj okno1

```
ekran -= okno2
```

❖ usuń z ekranu okno2

```
ekran != okno3
```

❖ wyjmij na pierwszy plan (na sam wierzch) okno3

Powyższe operatory zaspokajają nasze wszystkie potrzeby. Jeśli jednak chcielibyśmy, aby możliwe było takie składanie operacji

```
ekran = okno1 + okno2 + okno3
```

czyli

❖ na pustym ekranie umieść trzy okna 1,2,3

co można zapisać inaczej

```
ekran = (okno1 + okno2) + okno3
```

to musimy mieć po pierwsze operator:

```
okno1 + okno2
```

❖ stwórz chwilowy pusty ekran i na nim umieść okna 1 i 2

a po durgie operator:

```
ekran = ekran_chwilowy + okno3
```

❖ do istniejącej treści ekranu chwilowego dodaj okno3.

Jak widać dwa razy występuje operator +, ale za pierwszym razem argumentami są

(okno,okno)

a za drugim razem

(ekran,okno)

W naszym programie okna imitować będziemy kolorowymi kwadratami z opisem. I tu należą Ci się przeprosiny. W programie w miejscach, gdzie odbywa się rysowanie okien na ekranie, zobaczysz funkcje, których nie rozumiesz. Wynika to z faktu, że chodzi tu o pracę z urządzeniem zewnętrznym jakim jest ekran - a te zagadnienia omawiać możemy dopiero na końcu książki.

Nie przejmuj się jeśli, przy pierwszym czytaniu książki, tego rysowania na ekranie nie rozumiesz. W rozdziale tym zajmujemy się przecież operatorami, a nie operacjami z ekranem. Zrozumienie funkcji rysujących nie jest konieczne.

(Te funkcje biblioteczne są dostępne jeśli pracujesz z komputerem IBM PC i masz kompilator Borland C++.)

```
#include <constream.h>
#include <string.h>
#include <dos.h>          // dla funkcji sleep. sleep(1) oznacza:
                          //                    poczekaj 1 sekundę

constream monitor ; // klasa do pracy z kolorowym monitorem        ❶

class okno ;          // deklaracja zapowiadająca
//////////////////////////////////////////////////////////////////
```

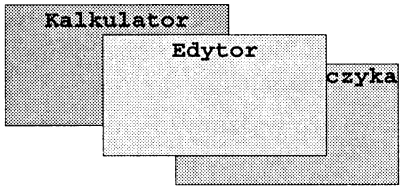
```
class kl_ekran {                                                    // ❷
     okno * tab_okn[20] ;                    // tablica wskaźników
     int pierwsze_wolne ;
public:
     // konstruktor
     kl_ekran()
     {
          pierwsze_wolne = 0 ;
     }
     void odswiez();        // narysowanie obecnego stanu ekranu
     void mazanie();        // mazanie treści całego ekranu
     // ----------- przeładowane operatory
     void operator +=(okno & ref_ok) ; // dodawanie okienka

     // dodawanie okienek ekran = ekran + okno
     kl_ekran & operator +(okno & ref_ok);

     void operator -=(okno & ref_ok) ;        // usuwanie okienka
     void operator != (okno & ref_ok) ;       // wyjmowanie na
                                              // sam wierzch
};
//////////////////////////////////////////////////////////////
class okno {                                                        // ❸
     int x, y, wys, sz ;        // geometria okna
     char kolor ;               // kolor okienka
     char tytul[20] ;           // opis okienka
public:
     okno(int xx, int yy, int ss, int ww,
                                  char kk, char * tt)
          : x(xx), y(yy), wys(ww), sz(ss), kolor(kk)
     {
          strcpy(tytul, tt);
     }
     // dodanie okienka w operacji:      ekran_chwilowy = okno + okno
     kl_ekran operator +(okno & m2) ;
     void narysuj_sie() ;
};
//////////////////////////////////////////////////////////////
// definicje funkcji składowych klasy okno
/*****************************************************/
void okno::narysuj_sie()
{
     monitor.window(x,y, x+sz-1, y+wys-1); // obszar pracy
     monitor << setattr((kolor<<4) | WHITE); // ustal kolor tła
     monitor.clrscr() ;                     // zamaluj kolorem tła

     // napisanie tytułu okna na środku pierwszej linijki
     monitor   << setxy( (sz-strlen(tytul)) / 2, 1)
               << tytul ;
}
/*****************************************************/
kl_ekran okno::operator +(okno & m2)                                // ❹
{
     // dodanie okienek do chwilowego ekranu roboczego
     kl_ekran roboczy ;
     roboczy += *this ;   // dodajemy pierwsze (ekran += okno)
```

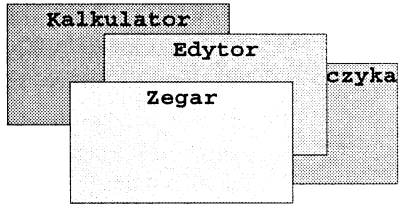
```
        roboczy += m2 ;          // dodajemy drugie
        return roboczy ;         // zwracamy ekran (przez wartość)
}
/*****************************************************/
// definicje funkcji składowych klasy kl_ekran
/*****************************************************/
void kl_ekran::operator +=(okno & ref_ok)
{                                                  // dodawanie okna ❺
        tab_okn[pierwsze_wolne++] = &ref_ok ;
        odswiez();
}
/*****************************************************/
void kl_ekran::operator -=(okno & ref_okna)
{                                                  // usuwanie okna ❻
        // odszukanie w tablicy wskaźnika do tego okna
        for(int i= 0 ; i < pierwsze_wolne ; i++)
        {
                if(tab_okn[i] == &ref_okna) break ;
        }

        // sprawdzamy jak zakończyło się poszukiwanie
        if(i == pierwsze_wolne)
        {
                // tzn. takiego okna nie ma obecnie na ekranie
                // więc po prostu nic nie robimy
        }
        else
        {       // okno odszukane, to je usuwamy z tablicy
                for(int k = i ; k < pierwsze_wolne ; k++)
                {       // przesuwanie pozostałych
                        tab_okn[k] = tab_okn[k+1] ;
                }
                pierwsze_wolne -- ;
        }
        mazanie();          // najpierw musimy zmazać cały ekran
        odswiez();
}
/*****************************************************/
/* wydobycie na sam wierzch zadanego okienka */
void kl_ekran::operator != (okno & ref_ok)          // ❼
{
        // polega na postawieniu go na samym koncu tablicy
        // w tym celu najprosciej usunac je z listy i natychmiast dodać
        *this -= ref_ok ;          // czyli ekran -= okno
        *this += ref_ok ;          // czyli ekran += okno
        odswiez();
}
/*****************************************************/
void kl_ekran::odswiez()                            // ❽
{
        for(int i = 0 ; i < pierwsze_wolne ; i++)
        {
                tab_okn[i]-> narysuj_sie();
        }
}
/*****************************************************/
```

```
void kl_ekran::mazanie()
{                          // cały ekran wypełniam czernią
     monitor << setattr( (BLACK<<4) | WHITE) ; // tło czarne
     monitor.window(1,1,80,25);          // na takim obszarze
     monitor.clrscr() ;                  // wykonać!
}
/*****************************************************/
kl_ekran & kl_ekran::operator +(okno & ref_ok)        // ❾
{          // dodanie okna do ekranu
     *this += ref_ok ;                   // czyli ekran += okno
     return *this ;                      // czyli return ekran
}
/*****************************************************/
main()
{
     monitor << setbk(BLACK);
     monitor.clrscr() ;

     kl_ekran ekran ;                    // definicja obiektu ekran

     // definicja kilku okienek
     okno gra(15,5,15,3,BROWN, "Gra w Chinczyka");
     okno kalkulator(2,3,14,3, RED, "Kalkulator");
     okno edytor(7,4,18,3, GREEN, "Edytor");

     // umieszczenie okien na ekranie

     ekran = gra + kalkulator + edytor ;                 // ❿
                         sleep(1) ;

     // wymyślamy jeszcze jedno okno
     okno zegar(4,6,18,3, BLUE, "Zegar");

     ekran += zegar ;    // dodanie go do obecnego ekranu        ❶❶
                         sleep(1) ;
     ekran != kalkulator ;//wyjęcie kalkulatora na sam wierzch   ❶❷
                         sleep(1) ;
     ekran != gra ;      // teraz wyjęcie gry
                         sleep(1) ;
     ekran -= kalkulator ;    // usunięcie jednego z okien       ❶❸
                         sleep(1) ;
}
/*****************************************************/
```

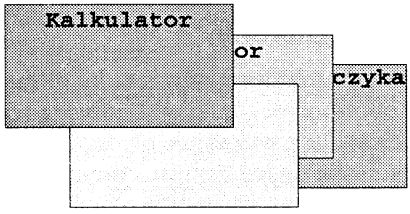
Ekran w trakcie pracy będzie się zmieniał, więc zamieszczam kilka (czarno-białych) rysunków z różnych faz

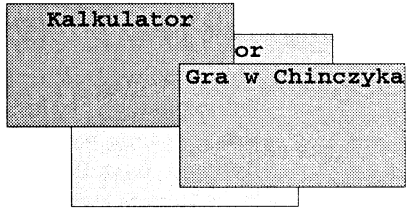
Po wykonaniu linijki ❿ i po wykonaniu linijki ❶❶



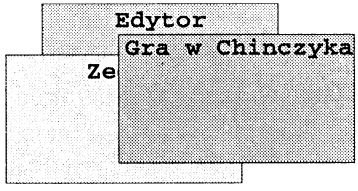


Po wykonaniu linijki ❶❷ i po wykonaniu 2 następnych





Po wykonaniu linijki ❶❸ - (patrz rysunek niżej)



Przyjrzyjmy się ciekawszym miejscom tego programu

❶ Abym mógł pisać kolorowe teksty w wybranych miejscach ekranu posłużę się klasą biblioteczną `constream`. (Oczywiście zauważyłeś komendę `#include` włączającą deklarację tej części biblioteki). Nie musisz teraz rozumieć działania tej klasy. Jak się okaże - obiekt tej klasy (nazywający się `monitor`) występować będzie w podobnych sytuacjach jak `cout`. Z tą różnicą, że jest on mądrzejszy, bo zna się na kolorach i pozycjonowaniu na ekranie.

❸ Deklaracja klasy `okno`. Jak widzimy, składnikami są liczby określające położenie lewego górnego rogu tego okna na ekranie oraz jego szerokość i wysokość. Okno ma jakiś kolor i tekst, który go opisze. Spójrz na konstruktor. Umieszcza on argumenty w odpowiednich składnikach i nie robi nic więcej - w szczególności nie rysuje okna na ekranie. Okno się narysuje na ekranie dopiero wtedy, gdy dostanie polecenie. Składnikiem tej klasy jest, jak widzimy, funkcja składowa `narysuj_sie`. Jest to publiczna funkcja składowa, bo wywoła ją ktoś inny. Ciało tej funkcji zawiera wywołania funkcji, których zrozumienie nie jest teraz konieczne. Ich działanie opisałem komentarzami.

❷ Deklaracja klasy `kl_ekran` reprezentująca ekran monitora. Ta klasa, to po prostu jakby lista, na którą wpisują się okienka chcące się pokazać na ekranie. Widzimy, że składnikiem jest 20 elementowa tablica wskaźników do okien. Dodatkowo jest składnik mówiący ile okien jest aktualnie na liście. Ekran, aby się narysować, po prostu przebiega tę tablicę i wydaje polecenia napotkanym

oknom, by się narysowały. W klasie widzimy funkcję składową `mazanie` - która zamalowuje ekran na czarno. Ciekawsza jest funkcja `odswiez` - która odpowiada za odświeżenie wizerunku ekranu. To ona rysuje wszystko na ekranie w sytuacji, gdy zajdzie jakaś zmiana.

❽ Patrzymy więc do definicji tej funkcji i co widzimy? Funkcja sama niczego nie rysuje. Jej praca polega na tym, że bierze tablicę wskaźników do okien, patrzy na co pokazuje pierwszy wskaźnik i zleca: „Okno, które pokazuję pierwszym wskaźnikiem proszę się teraz narysować na ekranie!". Potem to samo z wszystkimi wskaźnikami zapisanymi aktualnie w tej tablicy.

Zajmijmy się teraz przeładowanymi operatorami

❺ Aby możliwe było dodanie okienka do bieżącego wizerunku ekranu (wszystko jedno pustego czy już nie) mamy operator `+=`. Tutaj widzimy jego definicję. Jego praca polega na tym, że na końcu tablicy wskaźników dopisuje wskaźnik do nowego okna. Przy okazji inkrementuje się też licznik. Potem następuje wywołanie funkcji rysującej na ekranie jeszcze raz wszystkie okienka.

Jak widać jest to bardzo prosta funkcja, a mimo to obsługuje zapis

```
ekran += okno1
```

Jej wykorzystanie widzimy w programie w miejscu ❶❶.

❻ Nieco bardziej skomplikowana jest realizacja operatora `-=` zdejmującego dane okienko z ekranu. Zasada jednak jest dość czytelna. Najpierw należy odszukać w tablicy wskaźników adres wybranego okienka i ten adres stamtąd usunąć.

Nie jest to trudne. Do operatora jako argument przysyłane jest okienko (przez referencję). Pozwala nam to dowiedzieć się o adres tego okienka w pamięci. Ten adres porównujemy z kolejnymi adresami zapisami w tablicy. Jeśli nie znajdziemy, to znaczy, że tego okienka nie ma teraz na ekranie wcale. Możemy wówczas nic nie robić.

Jeśli natomiast w tablicy jest już ten adres, to usuwamy go. Usuwanie polega po prostu na tym, że wszystkie adresy na pozycjach dalszych przepisują się o jedną pozycję tablicy w dół. Tak, jakby stojący w kolejce ludzie nagle zadeptali tego, kto stał jako piąty. Kolejka się zmniejszyła, a po piątym elemencie zostaje tylko mokra plama.

Jeśli byśmy teraz zawołali funkcję odświeżającą ekran, to narysuje ona już o to jedno okno mniej. Dlatego, że to okno - jako nieobecne na liście - nie dostanie już więcej rozkazu, by się narysowało. Z drugiej jednak strony wizerunek tego okna na ekranie już od dawna jest. Trzeba więc najpierw wymazać całą treść ekranu i dopiero potem odświeżyć rysując bieżącą sytuację.

Jak prosto wykonuje się to w programie widzimy chociażby w miejscu ❶❸

❼ Wyjęcie na sam wierzch dowolnego okna polega na tym, by przy odświeżaniu zostało ono narysowane jako ostatnie. Aby tak się stało wskaźnik do tego okna powinien być jako ostatni w naszej tablicy. Jak zrobić takie przemieszczenie? Bardzo prosto - wystarczy usunąć okno z listy i natychmiast dodać. Ponieważ dodawanie okien do ekranu odbywa się zawsze przez dopisanie do końca tablicy, więc tym prostym sposobem załatwiamy cały problem.

Ponieważ operator zrealizowany jest jako funkcja składowa klasy `kl_ekran` więc ekran dostaje się do wnętrza tej funkcji przez wskaźnik `this`. Obiekt ekranowy to oczywiście `*this` a instrukcja usuwania z niego okienka to

```
*this -= ref_ok ;
```

To, że zamiast nazwą usuwanego okienka posługujemy się przezwiskiem (referencją), nie ma żadnego znaczenia - i tak chodzi o właściwy oryginalny obiekt. Dodawanie okienka do listy odbywa się według podobnej składni.

Mając tak zdefiniowany operator można bardzo elegancko pisać instrukcje żonglujące okienkami. Taką operację wyjmowania okienka na pierwszy plan wykonujemy w naszym programie w miejscu ❶❷ i po raz drugi linijkach następnych.

W zasadzie na tym wyczerpuje się lista naszych potrzeb. Mamy co chcieliśmy – możemy okna dodawać, usuwać i sprowadzać na pierwszy plan. Zdecydowałem się jednak na dalsze przeładowania by pokazać jeszcze kilka aspektów.

❿ Zobacz na ekranie tę instrukcję. Chodzi tu o to, by na ekranie znalazły się trzy okna

```
ekran = okno1 + okno2 + okno3 ;
```

Podobny zapis zmiennych przy działaniach matematycznych nie pozostawia złudzeń, że dotychczasowa treść zmiennej `ekran` jest niszczona, a wpisuje się do niej suma wyszczególnionych trzech składników. Zróbmy więc tak i u nas: niezależnie co było na ekranie do tej pory - odtąd mają być te trzy okna.

Problem jest jednak w tym jak rozwiązać takie wielokrotne sumowanie. Nie jest to trudne - inaczej można to zapisać jako:

```
ekran = (okno1 + okno2) + okno3 ;
```

a to proponuję rozdzielić na dwie poniższe instrukcje

```
ekran = okno1 + okno2 ;        // operator + dla arg: (okno, okno)
ekran = ekran + okno3 ;        // operator + dla arg: (ekran, okno)
```

Po prawej stronie zaznaczyłem jakie dodawanie tu występuje. Chodzi tu więc by zrealizować przeładowania właśnie takich operatorów.

Jest to prostsze niż się wydaje, bo prawie całą pracę mamy już zrobioną. Weźmy pierwszy z tych operatorów.

❹ Oto jego realizacja. Jak widzisz wewnątrz tej funkcji operatorowej definiujemy sobie roboczy obiekt klasy `kl_ekran`. (Nie ma żadnej kolizji z istniejącym już takim obiektem klasy ekran). Do tego ekranu znanym już operatorem `+=` dodajemy najpierw jedno okno, potem drugie.
Pierwsze okno przesłane jest to tej funkcji operatorowej (składowej klasy `okno`) za pomocą wskaźnika `this`, więc samo okno to po prostu `*this`. Umieszczenie go na ekranie to

```
roboczy += *this ;
```

Drugie okno jest przysłane przez przezwisko (referencję), więc po prostu tej referencji używamy, a oznacza ona oryginalny obiekt. Stąd wziął się zapis

```
roboczy += m2 ;          // m2 to przezwisko
```

Ponieważ rezultatem działania tego operatora ma być jakiś ekran, więc ten właśnie roboczy obiekt klasy `kl_ekran` zwracamy przez wartość.

❾ Przyjrzyjmy się teraz realizacji tego drugiego operatora. Ma on dodawać ekran i okienko, a w rezultacie dostać mamy nową postać ekranu - wzbogaconą o to okienko. Łatwo zauważyć, że chodzi o coś takiego

```
ekran = ekran + okienko
```

a to przecież (tylko logicznie!) to samo co

```
ekran += okienko
```

Skoro ten drugi wariant już potrafimy zrealizować, to oczywiście to wykorzystujemy i w rezultacie w funkcji widzimy

```
*this += ref_ok ;          // czyli ekran += przezwisko_okna
```

funkcja zwraca jako rezultat ten ekran - a ekran to przecież `*this`, bo jest to funkcja składowa klasy `kl_ekran`, czyli argument `ekran` dostał się tam przez składnik `this`.

Mając te dwa operatory możemy stosować już ten zapis z wielokrotnym dodawaniem okienek w jednej instrukcji.

Po co były nam te wszystkie przeładowania operatorów? Po to by funkcja `main` mogła wyglądać tak prosto i obrazowo, jak to widzisz w tekście. Nic więcej, ale czasem to wzgląd ogromnie ważny - szczególnie w wypadku, gdy piszemy taką klasę dla innych użytkowników. Ci użytkownicy łatwiej zapamiętają używanie operatorów `+=`, `-=`, `!=`, `+`, niż nazwy jakichś funkcji.

Wiem, że ten rozdział o przeładowaniu operatorów w pewnym momencie może Ci się wydać trudny. Chciałbym wobec tego powiedzieć coś na pocieszenie, dla tych którzy nie zrozumieją go od razu.

> Przeładowanie operatorów jest niczym więcej jak tylko efektownym sposobem ułatwiającym notację wyrażeń, w których występują obiekty danych klas. Przeładowanie operatorów nie daje niczego takiego, co nie było możliwe do tej pory. To tylko notacja się upraszcza.

Nie przejmuj się więc jeśli sprawią Ci jakieś trudności najbliższe paragrafy. Z drugiej strony jednak często w bibliotekach, którymi będziesz się posługiwał napotkasz operacje wykonywane właśnie za pomocą przeładowanych operatorów. Dobrze jest wówczas rozumieć ten mechanizm. Dlatego czytaj dalej ten rozdział.

18.10 Cztery operatory, które muszą być niestatycznymi funkcjami składowymi

Powiedzieliśmy, że w wypadku funkcji operatorowej mamy wybór: albo realizujemy ją jako funkcję składową danej klasy, albo jako zwykłą funkcję globalną.

Jednakże w wypadku czterech operatorów

```
=    []    ()    ->
```

wyboru nie ma. Operatory te **muszą** być niestatycznymi funkcjami składowymi klasy. Omówmy sobie kolejno te operatory.

18.11 Operator przypisania =

Dwuargumentowy operator przypisania

```
klasa & klasa::operator=(klasa &)
```

służy do przypisania jednemu obiektowi klasy `klasa` treści drugiego obiektu tej klasy. Często mówi się na to: „podstawienie".

Jeśli nie zdefiniujemy sobie tego operatora – kompilator automatycznie wygeneruje swoją wersję tego operatora – polegającą na tym, że przypisanie odbędzie się metodą „składnik po składniku"[†]. W rezultacie takiego przypisania będziemy mieli dwa obiekty o bliźniaczo identycznej treści.

> Najczęściej rzeczywiście o to chodzi, ale czasem może nam to nie odpowiadać. Jeśli w klasie składnikami są jakieś wskaźniki, lub jeśli klasa używa operatora `new` do rezerwacji miejsca w zapasie pamięci, wówczas mogą wyniknąć kłopoty.

Mówiliśmy już o analogicznej sytuacji przy okazji omawiania konstruktora kopiującego, więc jeśli nie pamiętasz - radzę zajrzeć do tego rozdziału (str. 361).

Skoro więc mamy akurat do czynienia z klasą, dla której przypisanie „składnik po składniku" nie jest dobre, to co robić?

Oczywiście: napisać swoją wersję operatora przypisania.

Jeśli pamiętasz jeszcze naszą klasę `wektorek`, to łatwo zauważysz, że dla niej przypisanie „składnik po składniku" jest tym, o co chodzi. Po prostu chcemy tu obiekt absolutnie identyczny. Nie musimy więc pisać tu swojej wersji operatora =.

Załóżmy jednak, że mamy do czynienia z klasą `wizytówka` – którą posługiwaliśmy się niedawno. O ile pamiętasz, na pomieszczenie informacji o nazwisku i imieniu rezerwowaliśmy operatorem `new` tablice w zapasie pamięci. Przedtem robiliśmy to dość rozrzutnie – i na nazwisko i na imię rezerwowaliśmy 80 znaków. Jeśli zamierzamy zbierać dane o tysiącach ludzi – musimy zrobić to sprytniej. Tym razem więc rezerwować będziemy tylko tyle pamięci, ile jest potrzebne do pomieszczenia konkretnego nazwiska oraz imienia. To, jak długie jest nazwisko – łatwo ustalić funkcją biblioteczną `strlen` [string length – ang. długość stringu] [††]. Funkcja ta zastosowana na przykład tak

†) Trzeba by raczej powiedzieć: kompilator będzie *chciał wygenerować* swoją wersję. Nie zawsze jest to możliwe.

††) (Czytaj: „string lengf")

```
strlen("Mickiewicz")
```

daje w rezultacie liczbę 10 – tyle bowiem liter ma to nazwisko. Nie muszę chyba przypominać, że rezerwując tablicę do przechowywania tego nazwiska należy przewidzieć dodatkowy element na kończący string znak NULL

```
char *wsk ;
     wsk = new char[10 + 1] ;
```

To wszystko na razie nie ma nic wspólnego z operatorem przypisania. Załóżmy jednak, że chcemy wykonać przypisanie dwóch obiektów klasy wizytówka. Jeden z nich ma krótkie nazwisko, a drugi długie.

```
wizytowka krotkie("Jan", "Kot") ;
wizytowka dlugie ("Wincenty",
                  "Konstantynopolitanczykiewicz");
```

A teraz dokonujemy przypisania

```
krotkie = dlugie ;
```

Czyli chcemy, by nazwisko z obiektu `dlugie` zostało przepisane do obiektu `krotkie`.

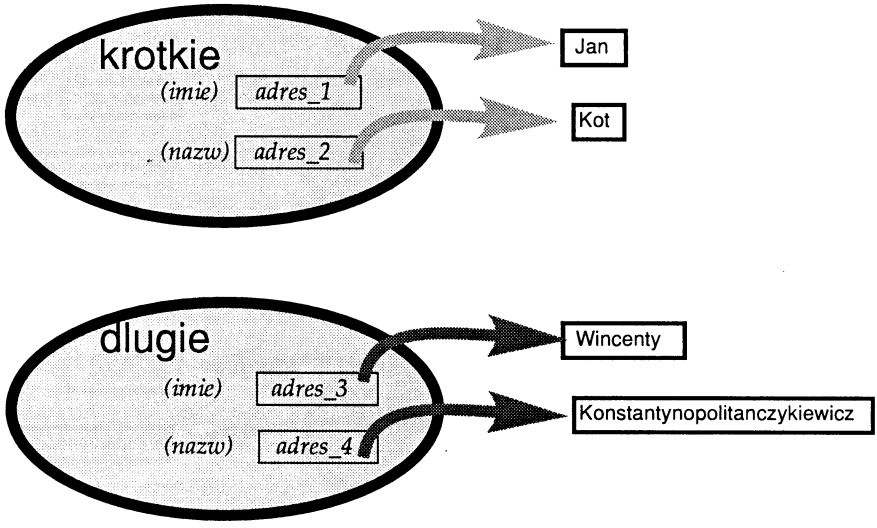
Jak to zrobić?

Przede wszystkim – ponieważ składnikami klasy są *wskaźniki* do tablic – nie możemy polegać na operatorze przypisania generowanym automatycznie. To dało by nam przypisanie „składnik po składniku" - czyli absolutną identyczność - a na to nie możemy się zgodzić.

Oto przyczyna: po przypisaniu wskaźniki obiektu `krotkie` (identycznie jak wskaźniki w obiekcie `dlugie`) pokazywałyby na tablice obiektu `dlugie`. To oznacza, że w wypadku, gdyby ten obiekt chciał coś zapisać do tablic – to pisałby nie po swoich, ale po tych, na których od niedawna pasożytuje – czyli tych należących do obiektu `dlugie`.

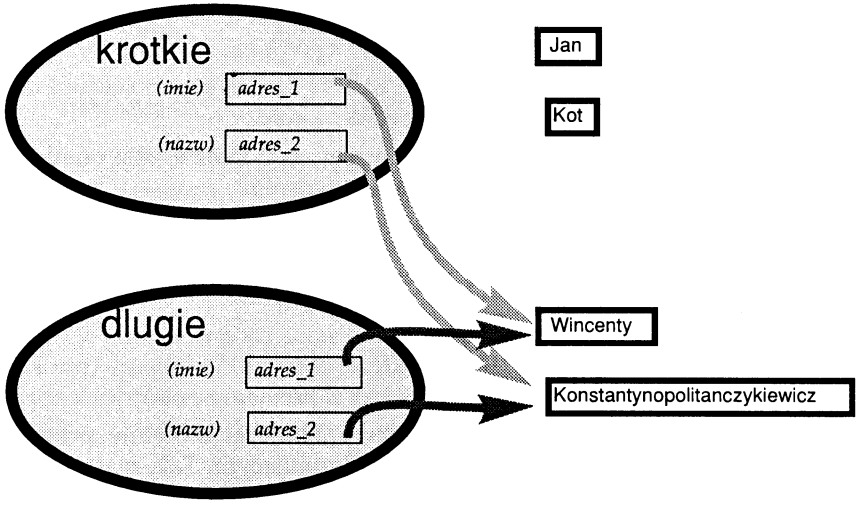
Co robić? – oczywiście zdefiniować operator przypisania, który nie przepisze „na ślepo" treści wskaźników, ale zachowa się poprawnie.

Zwróć uwagę na zamieszczone na tych stronach rysunki. Pierwszy pokazuje jak obiekty te wyglądają przed przypisaniem.



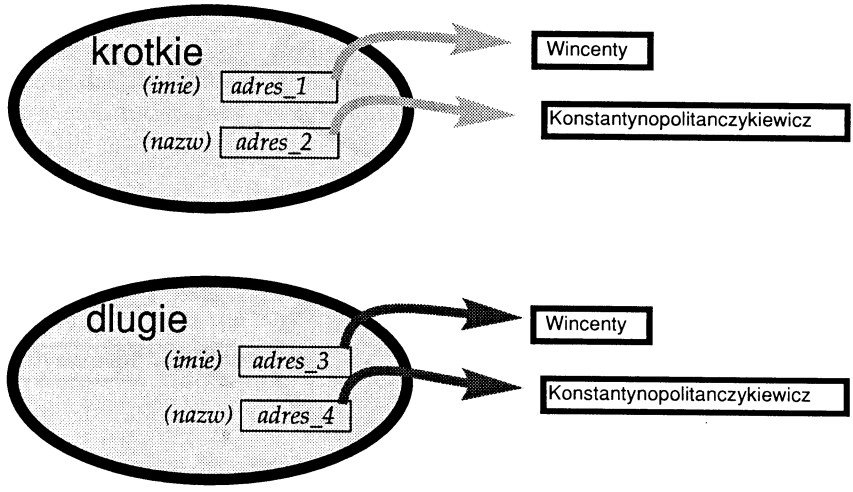
Następny rysunek pokazuje sytuację po przypisaniu systemem „składnik po składniku" – (tak nie chcemy),

Po przypisaniu metodą "składnik po składniku"



a kolejny rysunek – sytuację po przypisaniu naszym własnym systemem - to będziemy chcieli zrealizować.

Tak chcemy zrobić naszym operatorem



Ustalmy sobie co zatem trzeba zrobić przy przepisywaniu treści obiektów

Oczywiście dobre obyczaje nakazują, by niczego nie ruszać w obiekcie, który został nam dany tylko na wzór. Oto jak mamy postąpić z obiektem, któremu przypisujemy – czyli tym stojącym po lewej stronie znaku '='. (W naszym wypadku obiekt ten nazywa się `krótkie`.

- 1) Należy zlikwidować obie tablice tego obiektu. I tak są za krótkie, by pomieścić nowe nazwisko i imię.
- 2) Obliczyć jak długie mają być tablice, aby pomieścić nową treść, po czym takie tablice zarezerwować.
- 3) Patrząc na obiekt wzorcowy – przepisać z niego informacje do tych nowych tablic.

Jaki jest wniosek:

> W operatorze przypisania można wyraźnie rozróżnić dwie części:
> — tę, w której obiekt likwidujemy (punkt 1)
> — i tę, w której kreujemy obiekt jeszcze raz (punkty 2 i 3).

Zapewne zauważyłeś, że sprawy, o których mówimy, przypominają to, o czym mówiliśmy przy konstruktorze kopiującym. Tak, masz rację. To dlatego, że w obu wypadkach chodzi o ten sam znaczek '='.

Nie jest to jednak ta sama sytuacja. Jeśli znaczek = występuje w linijce definicji obiektu (inicjalizacja), to do pracy rusza konstruktor kopiujący.

W każdej innej sytuacji rusza do pracy operator przypisania. Radzę tę zasadę zapamiętać. Jeśli zapomnisz, to może się zdarzyć, że pracując nad tym fragmentem programu, gdzie znaczek = ma wykonać dla Ciebie jakieś usługi, robił będziesz omyłkowo modyfikacje w operatorze przypisania, podczas gdy wtedy pracuje właśnie konstruktor kopiujący. Lub odwrotnie – zmieniał będziesz konstruktor kopiujący, podczas gdy tam pracuje właśnie operator przypisania.

Zapamiętaj:

> Jedyną sytuacją, gdy na widok znaczka '=' rusza do pracy konstruktor kopiujący jest wystąpienie tego znaku w linijce definicji obiektu.

Znaczek ten wtedy oznacza inicjalizację, a nie przypisanie.

> Inicjalizacją zajmuje się konstruktor kopiujący,
> a przypisaniem - operator przypisania.

18.11.1 Przykład na przeładowanie operatora przypisania

Te wszystkie sprawy ilustruje nasz przykład.

```
#include <iostream.h>
#include <string.h>

//////////////////////////////////////////////////////////
class wizytowka {
     char *nazw ;
     char *imie ;
public :

     wizytowka(char *n, char *im) ;          // konstruktor
     wizytowka(const wizytowka &wzor) ;      // konstr. kopiujacy
     ~wizytowka();                           // destruktor
     void pisz(char *) ;
                                        // operator przypisania
     wizytowka & operator=(const wizytowka &wzor) ;
} ;
//////////////////////////////////////////////////////////
wizytowka::wizytowka(char*n, char *im)
{
     nazw = new char[strlen(n) + 1] ;                // ❶
     imie = new char[strlen(im) + 1] ;
     strcpy(nazw, n);                                // ❷
     strcpy(imie, im);
     cout << "Pracuje konstruktor zwykly" << endl ;
}
/*******************************************************/
wizytowka::wizytowka(const wizytowka &wzor)          // ❸
{
     nazw = new char[strlen(wzor.nazw) + 1] ;
     imie = new char[strlen(wzor.imie) + 1] ;
     strcpy(nazw, wzor.nazw);
     strcpy(imie, wzor.imie);
     cout << "Pracuje konstruktor kopiujacy " << endl ;
}
/*******************************************************/
wizytowka::~wizytowka()                              // ❹
{
     delete nazw ;
     delete imie ;
}
/*******************************************************/
void wizytowka::pisz(char *txt)
```

```
{
     cout << " " << txt
          << ": Mamy goscia, jest to "
          << imie << " " << nazw << endl ;
}
/*****************************************************/
wizytowka & wizytowka::operator=(const wizytowka &wzor)
{                                                        // ❺
     //—— część "destruktorowa" ————————
     delete nazw ;
     delete imie ;
     //—— część "konstruktorowa (konst. kopiujący)"—
     nazw = new char[strlen(wzor.nazw) + 1] ;
     imie = new char[strlen(wzor.imie) + 1] ;
     strcpy(imie, wzor.imie);
     strcpy(nazw, wzor.nazw);
     cout << "Pracuje operator= (przypisania)\n" ;
     return *this ;
}
/*****************************************************/
main()
{
     cout << "Definicje 'veneziano', i 'salzburger' \n" ;

wizytowka veneziano("Vivaldi", "Antonio"),              // ❻
          salzburger("Mozart", "Wolfgang A.");

     cout << "Definicja 'nowy' : \n" ;

wizytowka nowy = veneziano ;            // konstruktor kopiujący ❼
     cout << "Oto tresc w obiektach\n" ;

     veneziano.pisz("ven1");                            // ❽
     salzburger.pisz("sal1");
     nowy.pisz("now1");

     cout << "Zabawy z przypisywaniem ——\n" ;

     nowy = salzburger ;                                // ❾

     nowy.pisz("now2");
     nowy = veneziano ;
     nowy.pisz("now3");

     nowy = salzburger = veneziano ;                    // ❿

     nowy.pisz("now4");
     salzburger.pisz("sal4");
     veneziano.pisz("ven4");
}
```

Na ekranie po wykonaniu tego programu zobaczymy

```
Definicje 'veneziano', i 'salzburger'
Pracuje konstruktor zwykly                              ❻
Pracuje konstruktor zwykly
```

```
Definicja 'nowy' :
Pracuje konstruktor kopiujacy                    ❼
Oto tresc w obiektach
 ven1: Mamy goscia, jest to Antonio Vivaldi
 sal1: Mamy goscia, jest to Wolfgang A. Mozart
 now1: Mamy goscia, jest to Antonio Vivaldi
Zabawy z przypisywaniem ——
Pracuje operator= (przypisania)
 now2: Mamy goscia, jest to Wolfgang A. Mozart
Pracuje operator= (przypisania)
 now3: Mamy goscia, jest to Antonio Vivaldi
Pracuje operator= (przypisania)
Pracuje operator= (przypisania)
    now4: Mamy goscia, jest to Antonio Vivaldi
 sal4: Mamy goscia, jest to Antonio Vivaldi
 ven4: Mamy goscia, jest to Antonio Vivaldi
```

Komentarz

❶ W definicji konstruktora widzimy, że rezerwujemy tylko tyle pamięci, ile jest potrzebne do pomieszczenia konkretnych danych: imienia `im` oraz nazwiska `n`. Aby pomieścić obowiązkowy znak końca stringu NULL, dodajemy do tablic jeszcze po jednym elemencie. Stąd: +1.

❷ Do zarezerwowanego już obszaru pamięci kopiuje się stringi nazwiska i imienia.

❸ Naszą klasę wyposażamy dodatkowo w konstruktor kopiujący. Ten konstruktor – jak pamiętamy – musi jako jedyny argument przyjmować referencję obiektu swojej klasy. Ponieważ obiekt dostajemy jako wzór, więc składamy obietnicę, że wzorcowego obiektu nie będziemy zmieniać. Tą obietnicą jest słowo `const` przy argumencie formalnym `wzor`.

Treść (ciało) tego konstruktora bardzo przypomina treść konstruktora, który omówiliśmy powyżej. Jedyna różnica to to, że teraz nazwisko i imię nie zostają przysłane jako argumenty, ale odczytujemy je z wnętrza obiektu wzorcowego. Wolno nam – mimo, że wskaźniki `nazw` i `imie` są prywatne – bo to przecież obiekt tej samej klasy.

❹ Ponieważ klasa rezerwuje obszary w zapasie pamięci, dlatego wyposażamy ją w destruktor, który te rezerwacje odwołuje.

❺ Oto, będący przedmiotem tego rozdziału, operator przypisania. Po pierwsze jest zrealizowany jako funkcja składowa klasy `wizytówka`. Jest to, jak wiemy, obowiązkowe: operator przypisania musi być funkcją składową.

Druga bardzo ważna rzecz: (a jest ona ważna, bo daje receptę na pisanie takich operatorów) – Pamiętasz, przed przykładem mówiłem, że operator przypisania składa się z dwóch części: tej w której likwidujemy stary obiekt i tej, w której kreujemy go jeszcze raz od nowa tak, by spełnił nasze nowe oczekiwania. To właśnie widzimy w naszym operatorze.

Zapamiętajmy:

Operator przypisania składa się zwykle z dwóch części. Najpierw następuje część „destruktorowa", po czym następuje część „konstruktorowa" - przypominająca konstruktor kopiujący.

Nasz przykładowy operator przypisania jest idealną ilustracją tej zasady. Na zakończenie jest jednak coś wyjątkowego.

O ile ani destruktor, ani konstruktor nie mogły specyfikować typu rezultatu zwracanego, o tyle operator przypisania zwraca referencję obiektu swojej klasy. W deklaracji tego operatora widzimy przecież

```
wizytowka & operator=(const wizytowka &wzor) ;
```

Chodzi oczywiście o ten zapis `wizytowka &`, który stoi po lewej stronie słowa `operator=(...)`.

Zatem zwraca referencję obiektu klasy `wizytówka`. Którego to konkretnie obiektu tej klasy?

Tego już z deklaracji nie wyczytamy, spójrzmy do ciała funkcji i zobaczmy co stoi koło słowa `return` – tam jest odpowiedź.

```
return *this ;
```

`this` jest, jak wiemy, wskaźnikiem pokazującym na obiekt klasy `wizytówka`, na ten konkretny, na którego rzecz wywołano operator. Czyli ten, który stoi po lewej stronie znaku przypisania. To na niego pokazuje wskaźnik `this`. Wyrażenie `*this` określa już nie wskaźnik, ale obiekt, na który on pokazuje. Nasza funkcja operatorowa zwraca więc referencję tego obiektu.

Zapytasz pewnie: „Czy musi to zwracać? Przecież całe przypisanie już zostało zrobione i nowy obiekt ma się dobrze. Po co zatem zwracać dodatkowo tę referencję?"

Masz rację, wszystko jest już zrobione. Można by tego wcale nie zwracać. Swoją pieczeń już upiekliśmy. Czy pamiętasz jednak porzekadło o pieczeniu dwóch pieczeni na jednym ogniu? To właśnie tu robimy. O co chodzi dokładnie, wytłumaczymy za chwilę. Teraz przyjrzyjmy się funkcji `main`.

❻ W funkcji `main` widzimy definicję dwóch obiektów klasy wizytówka. Jest to najzwyklejsza definicja, zatem do pracy przystąpi zwykły konstruktor ❶. Po dowód spójrz na ekran, konstruktory są w naszym przykładzie „gadatliwe", więc łatwo zobaczyć, który kiedy pracuje.

❼ Definicja obiektu na wzór innego obiektu. Nawet się nie zastanawiaj: jest to linijka definicji i w niej występuje znak `'='`. Oznacza to, że działa tu konstruktor kopiujący ❸.

❽ Po tych wszystkich definicjach wypisujemy zawartość poszczególnych obiektów.

❾ Wszystkie powyższe operacje – to było tylko przypomnienie. **W tej linijce widzimy wspaniałość operatora przypisania.** Przypominam, że poniższe linijki sobie odpowiadają:

```
nowy = salzburger ;
nowy.operator=(salzburger) ;
```

W rezultacie tych zapisów, w obiekcie `nowy` od tej pory jest zapisane to samo nazwisko i imię, co w obiekcie `salzburger`. Na dowód wypisujemy to na ekranie.

Szczególnie z drugiej formy zapisu widzimy wyraźnie, że `operator =` został wywołany na rzecz obiektu `nowy` (na niego więc pokazuje wewnątrz tego operatora wskaźnik `this`).

Pamiętasz, że mówiłem iż funkcja `operator=` zwraca jako rezultat referencję do obiektu pokazywanego przez wskaźnik `this`.

Pytanie: Co tutaj robimy z tą referencją?

Odpowiedź: Nic. Nawet się nią nie zainteresowaliśmy.

W następnych linijkach widzimy dalsze przypisywania w podobnym stylu.

⑩ To jest bardzo ciekawa konstrukcja. Pamiętasz zapewne, że wypadku typów wbudowanych dopuszczalny jest taki zapis

```
int a,b ;
     a = b = 7 ;
```

W rezultacie obie te zmienne mają tę samą wartość – liczbę 7. To samo można zapisać tak

```
a = (b = 7) ;
```

Zapis ten jest możliwy dlatego, że najpierw wykonywane jest przypisanie w nawiasie. Pamiętasz zapewne, że wyrażenie przypisania, samo w sobie, ma także wartość. Jest nią właśnie wartość będąca przedmiotem przypisania. W naszym wypadku wyrażenie `(b=7)` jako całość ma także wartość 7. Czyli drugim etapem pracy nad tą instrukcją jest

```
a = (7) ;
```

Wygodny zapis prawda? Skoro więc można było tak z typem wbudowanym... Znasz moją obsesję. Teraz będę Cię przekonywał, że Twój własny typ nie będzie wcale gorszy. Otóż jeśli chcemy by instrukcja

```
nowy = salzburger = veneziano ;
```

była możliwa, to trzeba by wyrażenie

```
(salzburger = veneziano)
```

czyli

```
salzburger.operator=(veneziano)
```

miało samo w sobie wartość będącą przedmiotem przypisania.

Jak to zrobić? To proste! – ta funkcja operatorowa musi zwracać jakiś rezultat. Taki mianowicie, który nadaje się do przypisania następnemu obiektowi.

```
nowy = (salzburger.operator=(veneziano) ) ;
```

Skoro po prawej stronie znaku '=' ma u nas stać obiekt klasy `wizytowka` (ewentualnie referencja takiego obiektu) więc musimy to właśnie uczynić rezultatem funkcji `operator=()`. Dla oszczędności czasu decydujemy się zwracać

referencję. Ten zwrot obiektu, lub jego referencji, jest właśnie tajemnicą instrukcji

```
return *this ;
```

w operatorze przypisania.

Jak powiedziałem – nie jest to konieczne – bo proste (jednokrotne) przypisania możliwe są i bez tego. Z drugiej strony, jeśli tak małym kosztem możemy upiec tę drugą pieczeń, to czemu z tego rezygnować?

Jeśli jednak na początek wydaje Ci się to zawiłe, wówczas swój operator przypisania zdefiniuj jako

```
void wizytowka::operator=(const wizytowka &wzor) ;
```

- czyli nie zwracający niczego `(void)`. Oczywiście w ciele operatora usuwasz wówczas instrukcję

```
return *this;
```

Program wówczas będzie działał tak samo dobrze. To tylko linijka ⑩ będzie niemożliwa. Możesz ją jednak zastąpić dwoma innymi

```
salzburger = veneziano ;
nowy = salzburger ;
```

Jak zabezpieczyć się przed przypisaniem `a = a`

Może się zdarzyć, że ktoś, posługując się operatorem przypisania, napisze taką instrukcję

```
salzburger = salzburger ;
```

Jest to przypisanie, które nie ma specjalnego sensu, ale gramatycznie takie wywołanie jest poprawne. Może jednak sprawić kłopoty - bo przecież tablica z nazwiskiem zostanie na chwilę skasowana po to, by do niej wpisać tekst z nowej - która jest przecież tą samą, czyli właśnie skasowaną.

Jak się zabezpieczyć przed tym? Wystarczy jeśli nasz operator przypisania zacznie się od sprawdzenia czy aby nie przypisujemy tego samego temu samemu. Jeśli tak, to niech nic nie robi, jedynie zwróci referencję do obiektu z lewej strony

```
wizytowka & wizytowka::operator=(const wizytowka &wzor)
{
     if (this == &wzor)return *this
     // pozostałe instrukcje, jak poprzednio
     // ...
}
```

Na koniec jeszcze jedna uwaga. Przeładowany operator przypisania jest – jak wiemy – zawsze funkcją składową jakiejś klasy. Oznacza to, że po jego lewej stronie **musi** stać obiekt jakiejś klasy – bo to on zostanie przesłany za pomocą

wskaźnika `this`. To na jego rzecz wywoływany jest ten operator. Wynika z tego ważna konsekwencja: niemożliwy jest zapis instrukcji w stylu

```
6 = obiekt ;
```

Tylko mi nie mów, że Ci żal.

18.11.2 Jak to opowiedzieć potocznie?

Niniejszy paragraf przeznaczony jest dla najmłodszych czytelników. Jeśli jesteś starszy, to zdecydowanie opuść go, bo wyda Ci się infantylny.

Bardzo dużo powiedziałem na temat operatora przypisania. Tak dużo i trochę formalnie, że boję się czy najmłodszych czytelników nie znudziło to. Jeśli takie właśnie ogarnęło Cię, drogi czytelniku, uczucie, to zatrzymajmy się na chwilę i spójrzmy z dystansem na tę sprawę. Otóż dowiedzieliśmy się właśnie, że kompilator w stosunku do obiektów każdej naszej klasy - stara się nam zrobić prezent. Chce mianowicie podarować nam operator przypisania pozwalający na przypisywanie obiektów naszej klasy.

Przypisywanie - mówiąc inaczej - polega na tym, że możemy o dwóch obiektach naszej klasy zadecydować: „Niech ten obiekt będzie od tej pory taki, jak tamten". W rezultacie tak wypowiedzianego życzenia mamy teraz dwa absolutnie identyczne obiekty.

Najczęściej jest to właśnie to, o co nam chodzi - czyli bardzo cieszymy się z tego operatora, bo możemy w stosunku do obiektów naszej klasy używać znaczka = bez najmniejszego wysiłku definiowania go. Bez żadnej pracy możemy łatwo sprawić, że dowolne obiekty danej klasy stają się absolutnie identyczne.

Jeśli jesteś purystą językowym, to pewnie się teraz zdenerwowałeś - co to znaczy „absolutnie identyczne"? Po co to słowo „absolutnie"? Identyczne coś albo jest, albo nie jest i nie potrzeba dodawać żadnych przymiotników. Masz rację. Dodając słowo „absolutnie" chciałem byś odczuł, że chodzi o identyczność aż do bólu!

Na czym ma polegać ów „ból"? Na pewnych skutkach ubocznych, które mogą nas zaskoczyć w wypadku pracy z niektórymi klasami. Oczywiście mówiliśmy już o tym, więc teraz pokażę Ci jak takie skutki zaskoczyły by Cię w życiu nie-codziennym.

Historia jeszcze gorsza niż króla Midasa

Wyobraź sobie, że pewnego dnia wzdychasz: „Ech, chciałbym być taki jak... ". Tu, w miejsce kropek wstaw sobie nazwisko Twojego aktualnego idola muzyki rozrywkowej. Westchnąłeś tak sobie, a złe moce podsłuchały i spełniły natychmiast Twoje życzenie. Życzenie, które miało Cię uszczęśliwić, ale naprawdę...

Patrzysz do lustra i oczom nie wierzysz - zamiast swoich piegów widzisz szlachetnie piękną twarz, identyczną jak u Twojego idola. Taka jest teraz naprawdę Twoja twarz. Krzyknąłeś ze zdumienia - i usłyszałeś głos, który jest identyczny jak jego głos. Nie możesz uwierzyć, zaczynasz śpiewać - śpiewasz głosem identycznym jak on. I tak samo świetnie. „To fantastyczne" – myślisz

sobie – „jakiś kompilator wygenerował mi operator przypisania dla klasy człowiek. Mogę teraz po obu stronach znaku równości napisać

```
ja = moj_idol ;
```

– wreszcie koniec z bezbarwną egzystencją. Świat leży u moich stóp!"

Nie przewidziałeś jednak efektów ubocznych. Idziesz do drugiego pokoju, a tam siedzi twój idol. Wpadasz w bezgraniczny zachwyt, a on bezceremonialnie pyta co robisz w jego domu. Ty jednak twierdzisz, że to Twój dom - zresztą jedyny jaki masz.

Wywiązuje się sprzeczka w trakcie, której on wyciąga z kieszeni... (nie, nie rewolwer - skończ z oglądaniem tych głupich kaset) - więc on wyciąga z kieszeni akt własności tego domu, aby Ci udowodnić, że to jego dom. Ty także wyciągasz z kieszeni akt własności *wskazujący*, że jesteś właścicielem tego samego domu.

Postanawiacie jechać samochodem do notariusza. On wyciąga z kieszeni dowód rejestracyjny *wskazujący*, że samochód stojący w garażu, to jego samochód; Ty natomiast (choć nigdy dotąd nie miałeś garażu) masz taki sam dowód *wskazujący*, że jesteś właścicielem tego samochodu (i garażu też).

Czyje papiery są prawdziwe? Jedne i drugie, albowiem ten, kto nas w te tarapaty wpędził, postarał się, aby dokumenty były absolutnie identyczne.

Jadąc na policję myślimy, że historyjka ta przypomina nam starogrecki mit o królu Midasie, który gorzko pożałował swego życzenia, by wszystko, czego dotknie, zamieniało się w złoto.

Zastanówmy się, kiedy popełniony został błąd, który w rezultacie wprowadził nas w tarapaty

Powiedzieliśmy, że chcemy być identyczni jak nasz idol i staliśmy się tacy. Mamy taką samą twarz - ale jest to nasza twarz, on ma swoją. Wielokrotnie podziwialiśmy w telewizji jego harmonijny muskularny tors - i oto mamy takie same wspaniałe muskuły i obaj wyglądamy jak greccy bogowie. Z tym, że każdy z nas ma swój tors, swoje dłonie - i żadnego konfliktu nie ma.

Dlaczego zatem każdy z nas nie ma swojego (choć identycznego) domu i swojego (choć identycznego) samochodu?

Dlatego, że ich nie nosi się przy sobie, więc one nie są składnikiem obiektu klasy człowiek. (Myślę o człowieku w sensie nie tylko anatomicznym, ale także prawnym). Takim składnikiem jest jednak dowód rejestracyjny - czyli wskaźnik, że dany samochód jest jego własnością.

Wszystkie takie wskaźniki zostały skopiowane - stąd mamy też identyczne dokumenty własności domu. To w tym miejscu zaczęły się kłopoty: zły duch zamiast skopiować Ci samochód skopiował dowód rejestracyjny. Bo tylko ten dowód człowiek może nosić przy sobie.[†)]

†) Przypomina się łacińska maksyma: *Omnia mea mecum porto!*

Podobne kłopoty powstały przy prawie własności jego domu, którym odtąd musicie się dzielić. Nie będę ciągnął dalej tej historii, bo wolę nie myśleć o kłopotach związanych ze skopiowaniem aktu zawarcia małżeństwa z panią X.

Widzisz więc, że są sytuacje, kiedy otrzymany od kompilatora operator przypisania może nam przysporzyć kłopotów. Tak się może zdarzyć, gdy składnikami obiektu danej klasy są wskaźniki. Szczególnie, gdy pokazują one na coś, co ma być wyłączną własnością danego obiektu. Gdy przewidujemy, że będziemy przypisywać obiekty takiej klasy, to lepiej otrzymanego w prezencie operatora przypisania nie przyjąć i zdefiniować swoją wersję. O tym, jak to się robi, traktował poprzedni paragraf.

18.11.3 Kiedy operator przypisania nie jest generowany automatycznie

Operator przypisania może jako argument przyjmować obiekt danej klasy przysłany przez wartość lub przez referencję. Wiemy już, że jeżeli takiego operatora nie zdefiniujemy, wówczas kompilator postara się o wygenerowanie swojego – przypisującego na zasadzie „składnik po składniku".

Słowa „postara się" dobrze oddają tu sytuację. Mianowicie czasem to generowanie może się okazać niemożliwe.

- Jeżeli klasa ma składnik `const`, to operator nie będzie wygenerowany. Jest to oczywiste, bo skoro w klasie jest jakiś składnik ustanowiony jako `const` to znaczy, że dopuszczamy jego inicjalizację, ale potem nie wolno już go zmieniać, czyli nic do niego przypisywać.
- Podobnie w wypadku obecności składnika będącego referencją. Jak wiemy, referencję (przezwisko) tylko się inicjalizuje, a potem już przepadło – nie można rozmyślić się i przerzucić przezwisko na inny obiekt.
- Jeśli klasa (np. `radio`) ma składnik będący obiektem innej klasy (np. `tranzystor`) i w tej innej klasie (`tranzystor`) operator przypisania określony jest jako `private` – to wówczas nie będzie generowany operator przypisania dla klasy go zawierającej (klasa `radio`).

  To zrozumiałe – jeśli ktoś określa `operator=` jako `private`, to znaczy, że nie chce, aby przypisywanie do obiektu odbywało się spoza tej klasy. Klasa zawierająca nie może go użyć.

Dla wtajemniczonych:

- analogicznie jest w przypadku, gdy klasa ma klasę podstawową, w której `operator=` jest typu `private`.

Dla wtajemniczonych: przypomnienie o dziedziczeniu operatorów

Jeżeli operator jest funkcją składową klasy, to może być dziedziczony. Dotyczy to wszystkich operatorów oprócz operatora przypisania '='. (Patrz nast. rozdział str.505)

18.12 Operator [ ]

Operator [ ] – odwołania się do elementu tablicy jest operatorem dwuargumentowym. Jeśli chcemy przeładować ten operator, to funkcja operatorowa musi być niestatyczną (czyli zwykłą) funkcją składową klasy.

Jeśli mamy obiekt klasy `K` i dokonamy przeładowania tego operatora dla tej klasy, to wówczas wyrażenie

```
K obiekt ;

     obiekt[argument]
```

odpowiada wyrażeniu

```
     obiekt.operator[](argument)
```

Przypomnę po raz kolejny, że także i tutaj treść operatora nie musi mieć nic wspólnego ze znaczeniem, jakie ma on dla typów wbudowanych. Oczywiście lepiej jednak jeśli służy nam do podobnych celów.

Godny zwrócenia uwagi jest fakt, że argument nie musi być wcale typu `int`

Było to nie do pomyślenia dla typów wbudowanych. Pisaliśmy przecież wyrażenia

```
     float tablica[30] ;

          tablica[7]
          tablica[0]
```

nie można jednak było napisać

```
     tablica[0.5]
```

bo element 0.5 po prostu nie istnieje. Jest: albo 0 albo 1.

W przypadku przeładowania operatora[ ] argument wcale nie musi oznaczać numeru elementu tablicy. `Obiekt` wcale nie musi być tablicą.

Przykładowo wyobraź sobie klasę `miejsce`, która oznacza miejsce na ekranie, a operator [ ] zastosowany wobec obiektu tej klasy może oznaczać wypisanie w tym miejscu jakiegoś tekstu w ramce. Tylko dlatego, że symbol [ ] przypomina ramkę. Co jest wówczas argumentem ? Ani liczba `int`, ani nawet `float` tylko string. Tekst, który chcemy wypisać

```
     void miejsce::operator[](char* napis)
     {
          // ...
     }
```

Hipotetyczne użycie takiego operatora.

```
     miejsce info(6, 4);

          info["Wszystko w normie" ] ;
```

co odpowiada zapisowi

```
info.operator[]("Wszystko w normie") ;
```

Zastosowania mogą być wręcz nieprawdopodobne, ale zwykle o przeładowaniu tego operatora myślimy, gdy chcemy, by oddał nam podobne usługi, jak w przypadku typów wbudowanych.

Jednak ten operator [ ] w stosunku do typów wbudowanych ma pewną szczególną cechę:

Może on stać po obu stronach wyrażenia przypisania. Zauważ to w stosunku do tablicy `int`

```
int tablica[10] ;
int x ;

     x = tablica[5] ;              // po prawej stronie
     tablica[7] = x ;              // po lewej
     tablica[0] = tablica[4] ;     // po obu
```

Nic w tym nadzwyczajnego, znamy to już od dawna. Jak jednak zrobić, aby po naszym przeładowaniu tego operatora, mógł on również stać po obu stronach przypisania? Nie jest to takie trywialne.

Dla przykładu zróbmy taki eksperyment: udajemy, że nic nie wiemy o istnieniu tablic typu `int`. Zdefiniujemy sobie klasę, w której w środku co prawda będzie taka tablica, ale z zewnątrz niewidoczna

```
class tab_calkow {
     int a[100] ;
public :
     // deklaracja funkcji operatorowej
     // ...
} ;
```

To wszystko. Oto definicja obiektu naszej klasy:

```
tab_calkow t ;
```

Problem: jak powinien wyglądać operator `[]`, aby możliwa była instrukcja

```
int x = t[5] ;
```

polegająca na uzyskaniu wartości z szóstego (!) elementu tablicy?

Oto przykładowa definicja takiej funkcji operatorowej:

```
int tab_calkow::operator[](unsigned int ktory)
{
     return a[ktory] ;
}
```

Po prostu odczytuje się element o żądanym numerze i zwraca się jego wartość instrukcją `return`.

> *Podkreślam jednak, że operator `[]`, który widzisz wewnątrz funkcji, to już nie żadne przeładowanie, gdyż stoi tu koło składnika `int`. W tym wypadku więc zadziała standardowa wersja tego operatora.*

Zatem operator ten jako rezultat zwróci liczbę `int` schowaną w żądanym elemencie wewnętrznej tablicy. Inaczej mówiąc rezultatem wyrażenia

```
( t[5] )
```

czyli

```
( t.operator[](5) )
```

jest liczba typu `int` (Np. równa 22).
W instrukcji

```
x = t[5] ;
```

Chodziło nam właśnie o to. Inaczej mówiąc po obliczeniu wartości wyrażenia instrukcja ta zamienia się na

```
x = (22) ;
```

No dobrze, teraz dalsza część problemu: zastanówmy się co by było w wypadku instrukcji

```
t[5] = 8 ;
```

Czy w wyniku tego do elementu tablicy zostanie rzeczywiście wpisana liczba 8? Oczywiście, że nie! Lewa strona przypisania ma przecież *wartość* 22. Czy możliwy jest zapis

```
(22) = 8 ;                    // bezsens !
```

A teraz bardziej formalnie:

> Po lewej stronie przypisania mogą się znaleźć tylko tak zwane l-wartości (l- jak lewa strona) (patrz str. 155).

Zatem po lewej stronie może być tylko wyrażenie oznaczające obiekt. Może być to sam obiekt, może być to obiekt pod przezwiskiem (referencja), a może to być obiekt pokazywany przez wskaźnik.

Tylko do tego można wstawić (przypisać) w naszym wypadku liczbę 8.

To jest już prawie rozwiązanie naszego problemu. Co ma zatem zwrócić jako rezultat funkcja `operator[]` ? Referencję do tego, co ma być poddane podstawieniu.

Co jest *tym* w naszym wypadku? Obiekt klasy `tab_calkow`? Nie!

Co prawda to na rzecz tego obiektu wywołuje się operator `[]`, ale liczbę wstawia się już do jednego konkretnego elementu tablicy, która jest w środku. **Zatem zwracamy referencję do wybranego elementu tablicy.**

Teraz zobaczmy realizację tego operatora. To aż śmieszne, że tyle było tłumaczenia, a sprawę załatwia jeden jedyny znaczek `&` w pierwszej linijce definicji funkcji. Dzięki niemu operator zwraca element nie przez wartość, tylko przez referencję

```
#include <iostream.h>
#include <string.h>
//////////////////////////////////////////////////////////
class tab_calkow {
     int a[100] ;
```

```
public :
     // deklaracja funkcji operatorowej
     int & operator[](unsigned int ktory)          // tutaj cała
     {                                             // tajemnica
          return a[ktory] ;
     }
} ;
//////////////////////////////////////////////////////////
main()
{
tab_calkow t ;

     for(int i = 0 ; i < 100 ; i++)
          t[i] = 100 + i ;          // załadowanie tablicy

     // pracujemy tak jak na zwykłej tablicy !
     t[1] = t[2] + 50 + t[3] ;

     cout << "Mamy kolejno "
          << t[0] << ", " << t[1] << ", "
          << t[2] << ", " << t[3] << " itd..." ;
}
```

Na ekranie zobaczymy

```
Mamy kolejno 100, 255, 102, 103 itd...
```

Komentarz

To raczej podsumowanie:

> Chcąc napisać funkcję operatorową [ ] tak, by operator mógł stać po obu stronach znaku przypisania – musimy zdeklarować, że funkcja zwraca referencję do tego, czemu mamy przypisywać.

Nie jest to jednak obowiązek. Funkcja `operator[]` wcale nie musi nam służyć do pracy z tablicami.

Zastosowanie przeładowania operatora [ ]

Kiedy przeładowanie tego operatora może się nam przydać? Wspominałem już nieco o tym, że był to jeden z pierwszych operatorów, który musiałem przeładowywać.

Sprawa polegała na tym, że miałem posługiwać się olbrzymią tablicą. Taka tablica nie mieściła mi się w pamięci, więc trzymałem ją na dysku.

Ile razy potrzebowałem skorzystać z określonego fragmentu tablicy – uruchamiałem funkcję, która czytała żądany fragment do pamięci. Następnie mogłem dotrzeć do żądanego elementu i wykonać na nim jakąś operację: odczytać z niego wartość lub coś tam przypisać. Jeśli potrzebowałem innego elementu — stary odsyłałem specjalną funkcją na dysk i wczytywałem nowy.

Nie muszę chyba dodawać, że wykonanie instrukcji

```
tab[10][5] = tab[511][5] + tab[77][5] ;
```

było wręcz karkołomne. Tymczasem przeładowanie operatora [ ] doskonale rozwiązuje tu sprawę. Nie chodzi tu tyle o prędkość, co o łatwość zapisu.

Ponieważ nie mówiliśmy jeszcze o operacjach z dyskiem, dlatego z przykładem realizacji tego przeładowania musimy zaczekać (str. 665).

Innym, prostszym przykładem może być chociażby przeładowanie, które sprawdza czy nie odnosimy się do takiego elementu tablicy, który nie istnieje (bo tablica jest krótsza). W naszej klasie `tab_calkow` tak zrealizowany operator miałby następującą definicję

```
int & operator[](unsigned int ktory)
{
     if(i << 99) return a[ktory] ;
     else blad() ;
}
```

gdzie `blad` jest nazwą funkcji, która ostrzega nas, że oto właśnie chcieliśmy popełnić samobójstwo.

Podsumujmy

W paragrafie tym zobaczyliśmy, jak prosto przeładowuje się operator [ ]. Oczywiście można użyć go do dowolnego celu. Jeśli jednak chcielibyśmy by, w stosunku do obiektu naszej klasy, miał podobne działanie, jak w stosunku do typów wbudowanych (czyli służył do pracy z obiektami ustawianymi w tablicę), to trzeba pamiętać o jednej „harcerskiej" zasadzie. Po prostu ten nasz przeładowany operator powinien jako rezultat zwracać referencję do obiektu będącego pojedynczym, wybranym elementem tablicy.

Krótko mówiąc jego deklaracja powinna wyglądać tak:

```
nasza_klas & nasza_klasa::operator[ ](int nr_obiektu);
```

To wszystko.

18.13 Operator ( )

Jest tą trzeci z grupy operatorów, które muszą być niestatycznymi (czyli zwykłymi) funkcjami składowymi klasy.

O ile przy przeładowaniu operatorów = i [ ] mówiłem co zrobić, by operatory te sensownie i bezpiecznie imitowały działanie swych odpowiedników wobec typów wbudowanych - o tyle tutaj ten problem nie istnieje. Tym operatorem robimy naprawdę co chcemy, i jego użycie nie ma nic wspólnego z imitowaniem wywoływania funkcji. Dlatego przeładowanie tego operatora jest najłatwiejsze z omawianej czwórki. Pewnie pomyślałeś: „No to po co poświęcać mu cały paragraf?!" Odpowiem tak:

> Wyjątkowość tego operatora polega na tym, że jako jedyny może przyjąć więcej niż dwa argumenty. Wszystkie inne operatory są albo jedno-, albo dwuargumentowe. Ten przyjmuje więcej, czyli dzięki niemu do funkcji operatorowej może zostać przesłana większa liczba argumentów.

Jeśli `obiekt` jest obiektem klasy, dla której ten operator został zdefiniowany, to operatorem tym można przykładowo posłużyć się w taki sposób

`obiekt(`*arg1, arg,2 arg3, arg4, arg5*`)`

lub wywołując go jawnie na rzecz tego obiektu.

`obiekt.operator()(`*arg1, arg,2 arg3, arg4, arg5*`)`

Zauważ, że do funkcji operatorowej przesyłamy więcej niż 2 argumenty (tutaj 5, plus jeden ukryty – wskaźnik do obiektu, na rzecz którego został ten operator wywołany).

Oczywiście mogliśmy zdefiniować funkcję przyjmującą inną liczbę argumentów. Funkcje takie mogą istnieć równocześnie, gdyż przeładowują się nawzajem. Spełnione są przecież warunki przeładowania

- funkcje mają ten sam zakres ważności nazwy (zakres klasy),
- różnią się rodzajem, liczbą lub kolejnością argumentów.

Kiedy takim przeładowaniem można się posłużyć?

Oczywiście wtedy, gdy chcielibyśmy skorzystać z możliwości przesłania do operatora większej liczby argumentów. Jest to najważniejsza i najczęstsza przesłanka do stosowania tego operatora.

Wiele argumentów może być nam potrzebnych na przykład wtedy, gdy klasa opisuje tablicę wielowymiarową. Zwykle, aby odnieść się do tablicy, przeładowuje się operator `[]`. Jednak ten operator może nam „obsłużyć" tylko jeden wymiar tablicy – jest bowiem tylko dwuargumentowy. Jeśli natomiast tablica jest 3 wymiarowa to wszystkie trzy jej indeksy można wysłać do operatora umieszczając je w nawiasie. Np.

```
tab(1,1,6) ;
```

Innym powodem, dla którego wybór może paść właśnie na ten operator, jest jego skojarzenie z wykonywaniem jakiejś czynności.

Klasa bowiem może opisywać nie tylko jakiś obiekt złożony z jakichś liczb, ale może też opisywać proces. Jeśli obiektem takiej klasy jest proces zbierania danych, to zapis

```
zbieranieDanych() ;
```

od razu kojarzy się z wykonywaniem jakiejś czynności. Jest to argumentacja czysto skojarzeniowa, ale właściwie czemu nie?

O przeładowaniu takiego operatora pomyślimy także wtedy, gdy w klasie jest tylko jedna jedyna funkcja składowa, którą co chwilę wywołujemy na rzecz obiektu

Jeśli mamy klasę `sygnalizator` opisującą różne urządzenia alarmowe

```
sygnalizator syrena ;                    //def obiektu
```

I co chwilę wywołujemy funkcję

```
syrena.ryczec() ;
```

Ponieważ robi się to ciągle, można sobie oszczędzić nazwy tej funkcji i pisać po prostu

```
syrena() ;                    // czyli: syrena.operator()() ;
```

Oszczędzamy pisania, a zapis i tak jest jasny. Funkcja nie musi być nawet tą jedną, jedyną. Mogą być inne, ale wyraźnie mniej ważne. Podstawową funkcją syreny jest jednak: ryczeć. Możliwe są dla niej jakieś inne funkcje – np. regulacja wysokości tonu. Tę funkcję wywołuje się jednak rzadziej.

Inny przykład z życia codziennego. Jeśli w teatrze inspicjent woła: „Kurtyna!" to nie chodzi przecież o nazwanie czegoś, tylko o wywołanie na tym **wiadomej** akcji.

Wszystko załatwia obecność w klasie takiej funkcji:

```
void kurtyna::operator()(int kierunek)
{
     if(kierunek)
          w_gore();
     else
          w_dol();
}
```

Ponieważ nie przewidujemy, żeby ten operator miał się znaleźć kiedykolwiek po lewej stronie znaku przypisania, więc operator zwraca typ `void`.

Nie musi jednak być tak zawsze. Jeśli tylko chcesz, by operator ten mógł stać po lewej stronie znaku przypisania to powinieneś postarać się, by zwracał referencję do obiektu, któremu przypisanie ma zostać wykonane. O tym już mówiliśmy w poprzednim paragrafie.

18.14 Operator ->

Jeśli nie zrozumiesz tego paragrafu od razu, nie przejmuj się. Przeładowywanie tego operatora to już naprawdę hiszpańska szkoła jazdy. Nie, żeby to było takie trudne, po prostu rzadko się to robi.

Operator -> odniesienia się do składnika klasy musi być zdefiniowany jako niestatyczna (czyli zwykła) funkcja składowa klasy.

Operator ten jest jednoargumentowy, a działa on na argumencie stojącym po jego lewej stronie. Jest to trochę zaskakujące. Operatora tego nie interesuje zupełnie co stoi po prawej stronie. Później zobaczymy dlaczego.

```
obiekt->
```

Argumentem jest tu obiekt, a nie – jak do tego jesteśmy przyzwyczajeni - wskaźnik do obiektu.

Jeśli dla danej klasy zdefiniowany został ten operator, to jego wystąpienie wobec obiektu tej klasy

```
obiekt->składnik
```

jest równoważne wyrażeniu

```
(obiekt.operator->() ) -> składnik
```

Zauważ, że teraz w tym wyrażeniu mamy dwa symbole ->. Tylko ten lewy jest tutaj przeładowaniem. Ten prawy to zwykły operator. Argumentem tego 'naszego' przeładowanego operatora jest obiekt (albo jego referencja). Natomiast ten drugi, zwykły operator -> , po staremu wymaga po swojej lewej stronie adresu.

Zapytasz pewnie:

Dlaczego nasz przeładowany operator jest jednoargumentowy ?

Zobacz sam. Na argumencie stojącym ze swojej lewej strony wykonuje on jakąś akcję po czym zwraca rezultat.

```
(obiekt.operator->( ) ) -> składnik
( rezultat ) -> składnik
```

To wszystko. Składnik go nie interesuje. Teraz na ten rezultat oraz na składnik patrzy już *zwykły* operator -> , on jest już dwuargumentowy.

Jeśli chcemy, by ten przeładowany operator rzeczywiście wykonywał dla nas operację odniesienia się do składnika (a wcale nie musi) – wówczas funkcja operatorowa -> powinna zwrócić albo wskaźnik do obiektu, albo referencję (która jest przecież jakby ukrytym adresem). Po wykonaniu naszej funkcji operatorowej -> musimy więc mieć

```
(wskaźnik) -> składnik
```

bo to może stać po lewej stronie zwykłego, dwuargumentowego operatora ->

Co naprawdę może dla nas zrobić taki operator?

To samo co zwykły operator -> **ale**:

- Zwykły operator nie może być zastosowany do **obiektu** danej klasy. Stosuje się go tylko wobec wskaźnika do obiektu tej klasy. To jest ogromna różnica.
- Operator ten pozwala nam wykonać jakąś dodatkową akcję przy okazji sięgania do składnika klasy. Tę dodatkową akcję określamy w ciele funkcji operatorowej.

„Zręczny wskaźnik"

Standardowym przykładem cytowanym w takim przypadku jest tak zwany **zręczny wskaźnik.** Mimo tej nazwy nie chodzi tu o wskaźnik, tylko o klasę,

która służy do produkcji takich wskaźników. Zachowują się one jak zwykłe wskaźniki, ale mają jeszcze pewną dodatkową inteligencję.

Zwykłym wskaźnikiem sięgamy do wnętrza klasy w ten sposób:

```
zwykły_wskaźnik -> składnik
```

wskaźnikiem zręcznym sięgamy tak:

```
zręczny_wskaźnik -> składnik
```

czyli identycznie. No to gdzie tu spryt?

> Jest, jest! Nie zapominaj, że pod operatorem -> kryje się w drugim wypadku funkcja, która może zrobić parę sprytnych rzeczy. Może to być liczenie ile razy dokonywane jest odnoszenie się do składników klasy tym właśnie sposobem, może to być wyświetlanie na ekranie informacji z ostrzeżeniem itd.

Powiedziałem, że operator ten **można** zdefiniować – z poprzednich paragrafów wiesz, że nie jest on automatycznie generowany.

Pewnie myślisz teraz: „Wobec tego jeśli nie zdefiniujemy operatora -> , to jakim prawem możliwa jest operacja

```
zwykły_wskaźnik_do_klasy -> składnik
```

skoro ten operator nie został (jeszcze) zdefiniowany?"

Prosta sprawa: po lewej stronie operatora -> stoi nie obiekt danej klasy, ale wskaźnik do takiego obiektu. Wskaźnik to nie to samo co obiekt.

Wskaźnik (do czegoś), to typ wbudowany. Zatem w tym wypadku działa wersja operatora -> dla typów wbudowanych. Ta już istnieje! Za to nie mamy przeładowania na okoliczność wystąpienia tego symbolu obok obiektu danej klasy.

Dopóki nie zdefiniujemy funkcji operatorowej -> dla danej klasy `K`, po lewej stronie takiego znaku -> nie może się znaleźć **obiekt** tej klasy `K` (ani jego przezwisko czyli referencja)

```
obiekt -> składnik          // kompilator wówczas zaprotestuje !
```

Zręcznych wskaźników może być wiele rodzajów, pokażemy więc jakieś przykładowe zastosowanie

Oto program, w którym mamy do czynienia z dwoma klasami obiektów.

Klasa `wekt` reprezentuje wektor trójwymiarowy. Podobną klasą już bawiliśmy się wielokrotnie.

Klasa `spryciarz`, która reprezentuje wskaźnik do obiektów klasy `wekt`. Właściwie klasa ta powinna nazywać się nie `spryciarz` ale `szpicel`. Klasa ta bowiem notuje sobie w pamiętniku do jakich celów i ile razy użyliśmy tego wskaźnika. To dzięki przeładowaniu operatora -> . Ilekroć posłużymy się tym operatorem wobec **obiektu klasy spryciarz**, za każdym razem zostanie to wpisane do akt.

```
#include <iostream.h>
//////////////////////////////////////////////////////////
class wekt {
public :
     float x, y, z ;
     //——— konstruktor ————————
     wekt(float a, float b, float c)
                         : x(a), y(b), z(c)          //❶
     { }
     //——— kilka zwykłych funkcji składowych
     void podwojenie()
     {
          x *= 2 ; y *= 2;   z *= 2 ;
     }
     //————————
     void pokaz()
     {
          cout << "x= "<< x << ", y= "<< y
               <<", z= "<< z << endl ;
     }
} ;
//////////////////////////////////////////////////////////
class spryciarz {
     wekt *wsk ;                                      // ❷
     wekt * (pamietnik[10]) ;                         // ❸
     int uzycie ;
public :
     //———————— operator przypisania————————————————
     void operator=(wekt* w)                          // ❹
     {
          wsk = w ;
     }
     //——— konstruktor (także domniemany !)
     spryciarz(wekt * adr= NULL) ;                    // ❺
     //————————————— przeładowanie operatora ->
     wekt * operator->() ;

     void statystyka(void);
} ;
/*********************************************************/
spryciarz::spryciarz(wekt * adr) :    wsk(adr),
                                      uzycie(0)       // ❻
{
     for(int i = 0 ; i < 10 ; i++)pamietnik[i] = NULL ;
}
/*********************************************************/
wekt * spryciarz::operator->()                        // ❼
{
     //———————— akcja sprytna : wpisujemy do akt !
     pamietnik[uzycie] = wsk ;
     uzycie = (++uzycie) % 10 ;                       // ❽
     //———————— zwykła akcja————————————
     return wsk ;
}
/*********************************************************/
void spryciarz::statystyka(void)
```

```
{
     cout << "Ostatnie 10 wypadkow uzycia odbylo sie "
               "dla obiektow \no adresach :\n" ;
     for(int i=0 ; i < 10 ; i++){
          cout << pamietnik[ ( (uzycie) + i) % 10 ] // ❾
               << ((i==4)? "\n" : ", ") ;
     }
     cout << endl ;
}
/*****************************************************/
main()
{
float m ;
wekt www(1, 1, 1);

wekt *zwykly_wsk ;                                    // ❿
spryciarz zreczny_wsk ;                               // ⓫

     zwykly_wsk = &www ;
     zreczny_wsk = &www ;                             // ⓬

     cout << "Operacja za pomoca zwyklego wskaznika \n" ;
     m = zwykly_wsk -> x ;
     cout << "m = " << m << endl ;
     cout << "Operacja za pomoca zrecznego wskaznika \n";
     m = zreczny_wsk -> x ;                           // ⓭
     cout << "m = " << m << endl ;

wekt      w2(2, 2, 2),
          w3(3, 3, 3),
          w4(44, 10, 1) ;

     zreczny_wsk = &w2 ;
     zreczny_wsk->podwojenie() ;                      // ⓮
     zreczny_wsk->pokaz();

     zreczny_wsk = &w3 ;
     zreczny_wsk->podwojenie() ;
     zreczny_wsk->pokaz();

     zreczny_wsk = &w4 ;
     zreczny_wsk->podwojenie() ;
     zreczny_wsk->pokaz();

     zreczny_wsk.statystyka() ;                       // ⓯
     zreczny_wsk = &www ;
     zreczny_wsk->pokaz();

     zreczny_wsk.statystyka() ;
}
```

Na ekranie po wykonaniu tego programu pojawi się

```
Operacja za pomoca zwyklego wskaznika
m = 1
Operacja za pomoca zrecznego wskaznika
```

```
m = 1
x= 4, y= 4, z= 4
x= 6, y= 6, z= 6
x= 88, y= 20, z= 2
Ostatnie 10 wypadkow uzycia odbylo sie dla obiektow
o adresach :
0x0, 0x0, 0x0, 0x3e110fec, 0x3e110fb2
0x3e110fb2, 0x3e110fa6, 0x3e110fa6, 0x3e110f9a,
0x3e110f9a,
x= 1, y= 1, z= 1
Ostatnie 10 wypadkow uzycia odbylo sie dla obiektow
o adresach :
0x0, 0x0, 0x3e110fec, 0x3e110fb2, 0x3e110fb2
0x3e110fa6, 0x3e110fa6, 0x3e110f9a, 0x3e110f9a,
0x3e110fec,
```

Komentarz

❶ Klasa `wekt` nie jest niczym szczególnym. Jej konstruktor, jak widzimy, inicjalizuje składniki w liście inicjalizacyjnej. Tak wydawało mi się krócej.
Inne funkcje składowe to `podwojenie` – (podwajające każdą ze współrzędnych) oraz funkcja `pokaz` (wypisująca na ekran).

❷ Klasa `spryciarz`. Ponieważ jest ona klasą obiektów będących wskaźnikami do obiektów klasy `wekt`, dlatego gdzieś składnikiem tej klasy powinien być właśnie taki składnik. Mamy go dokładnie w miejscu ❷. Reszta składników, to już jakby tylko dla nadania temu wskaźnikowi „inteligencji".

❸ Tablica wskaźników do obiektów klasy `wekt`. To tutaj będziemy sobie notować adresy, na które pokazywał zręczny wskaźnik w chwili, gdy użyto operatora `->`.
Widzimy też składnik `uzycie`. Ponieważ pamiętnik nie jest nieskończony – możemy w nim zapisać tylko 10 ostatnich wypadków, dlatego posłużymy się składnikiem `uzycie`. Dzięki niemu najstarsze dane w pamiętniku będziemy kasować i wpisywać w to miejsce najnowsze.

❹ Definicja operatora przypisania dla tej klasy. Wydaje się oczywiste, że wskaźnik często ustawia się tak, by pokazywał na coraz to inne obiekty. Tak, jak w wypadku typów wbudowanych

```
int a, b;
int *wski ;
          wski = &a ;
          wski = &b ;
```

Skoro występuje tu znak `'='` , a nie jest to definicja obiektu, znaczy to, że tu zadziałać ma operator przypisania (a nie konstruktor kopiujący).

Realizacja tego operatora jest prymitywna – przysłany adres obiektu klasy `wekt` wpisujemy do składnika `wsk`. Przeładowanie tego operatora nie jest przedmiotem tego rozdziału. Jednak, gdy go mamy, operacje zmiany wskaźnika będą się odbywały w zapisie bardzo naturalnym. Widać to choćby w ❶❷.

❺ Klasa `spryciarz` ma konstruktor. Ponieważ w deklaracji widzimy, że można go także wywołać bez żadnych argumentów, dlatego jest to także konstruktor domniemany.

❻ Realizacja tego konstruktora. Co się da inicjalizuję listą inicjalizacyjną – dla skrócenia zapisu. W ciele funkcji jest wpisanie adresów zerowych do elementów `pamietnika`.

❼ Oto funkcja będąca przedmiotem tego paragrafu: przeładowanie operatora `'->'` Operator – wedle zasady jest funkcją składową klasy `spryciarz`. Jest on jednoargumentowy, a ten jedyny argument przychodzi do niego za pośrednictwem wskaźnika `this`. Dlatego w liście argumentów formalnych nie ma nic.

Ponieważ jesteśmy zdecydowani, że operator ten ma nam oddawać usługi związane z **pokazywaniem** (a nie gwizdaniem itd.) dlatego powinniśmy zwrócić jako jego rezultat adres obiektu, na który nasz `spryciarz` właśnie pokazuje. Ten adres mamy wpisany w składniku `wsk`. To właśnie zwracamy jako rezultat działania operatora.

❽ Zanim jednak to zrobimy – zapisujemy sobie bieżącą wartość składnika `wsk` do tablicy pamiętnik. W miejscu ❽ widać chwyt jaki zastosowałem, by indeks `uzycie` – mimo, że za każdym razem inkrementowany – po osiągnięciu wartości 9 zmienił się na 0. Dzięki temu starsze zapisy w pamiętniku będą kasowane.

❾ W funkcji `statystyka` wypisujemy na ekran adresy schowane w tablicy pamiętnik. Wypisywanie nie odbywa się od elementu 0 do elementu 9, lecz od najstarszego do najnowszego. Jeśli więc przed chwilą w elemencie nr 7 zanotowaliśmy jakiś adres, to wypisywanie odbędzie się według porządku: 8, 9, 0, 1, 2, 3, 4, 5, 6, 7.

❿ Definiujemy zwykły wskaźnik do pokazywania na obiekty klasy `wekt`.

❶❶ Definiujemy obiekt „zręczny wskaźnik". (Zauważ, że w definicji nie ma gwiazdki, bo nasz „wskaźnik" tak naprawdę nie jest wskaźnikiem, a obiektem).

❶❷ Ustawienie zręcznego tak, by pokazywał na jakiś obiekt, odbywa się podobnie, jak w wypadku zwykłego wskaźnika. Tu oczywiście działa nasz przeładowany operator **przypisania**. Opłaciło się tutaj, a potem jeszcze wiele razy poniżej.

❶❸ Posługiwanie się zręcznym wskaźnikiem także jest identyczne, jak w przypadku wskaźnika zwykłego (por. 3 linijki wyżej). Jedyna różnica to to, że w linijce ❶❸ po cichu zostało zapisane, że skorzystaliśmy z tego wskaźnika. Adres obiektu `www` poszedł do `pamietnika`.

❶❹ Widzimy tu całą serię sytuacji, gdy posługujemy się operatorem `->'`
Za każdym razem jest to notowane w pamiętniku.

❶❺ Od czasu do czasu możemy zażądać raportu. Funkcja `statystyka` pokazuje nam informację, które w pamiętniku złożył przeładowany operator '->'

Jak widać operator „->" przydaje się, gdy piszemy klasę, której obiekty mają pełnić rolę podobną wskaźnikom.

Na zakończenie wypada dodać, że jednoargumentowy operator '->' nie ma nic wspólnego z operatorem '->*'. Szczególnie ten drugi wcale nie musi być funkcją składową.

W ten sposób zakończyliśmy omawianie czterech wyjątkowych operatorów = [ ] ( ) ->

Wyjątkowych przez to, że jeśli chcemy je przeładować, to musimy je zdefiniować jako niestatyczne funkcje składowe klasy.

Są jednak inne operatory odróżniające się od pozostałych tym, że jest ściśle określone, co mają zwracać jako rezultat swojego wykonania.

Te operatory to new i delete. Nimi zajmiemy się w następnych paragrafach.

Przypominam, że w wypadku innych operatorów nie obowiązują żadne restrykcje jeśli chodzi o typ rezultatu. Funkcje mogą nawet nic nie zwracać. Wszystko, co do tej pory napisałem, to były rady co „opłaca się" zwracać po to, by mieć taki lub inny efekt.

18.15 Operator new

Operator new służący do obsługiwania rezerwacji obszarów w zapasie pamięci (free store) – ma swoją wersję globalną i tą wersją do tej pory posługiwaliśmy się. Tej także wersji można użyć dla obiektów dowolnych klas.

Jeśli jednak z jakichś powodów ta wersja nam nie odpowiada – możemy na użytek obiektów wybranej klasy ów operator przeładować. Wówczas to obok wersji globalnej istnieje wersja lokalna zdefiniowana na użytek danej klasy. Kompilator przy tworzeniu obiektów dowolnej klasy najpierw sprawdza, czy na tę okazję nie zdefiniowaliśmy operatora new. Jeśli nie, to wtedy uruchamiana jest globalna wersja tego operatora.

Gdy chcemy zdefiniować funkcję operator new dla jakiejś klasy to

musimy pamiętać, że jest parę zastrzeżeń:

- operator new jest funkcją składową **statyczną**. Oznacza to, że jest wywoływany nie na rzecz konkretnego obiektu (bo ten jeszcze nie istnieje), ale na rzecz klasy. Jeśli zapomnimy o przydomku static, kompilator zrobi to i tak w ten sposób. Bez upominania nas o błędzie.
- nowa funkcja operator new() **musi zwracać typ void***, a pierwszy argument ma być typu size_t.

Argument służy do określenia ile pamięci zarezerwować. (Co to jest typ size_t – to zależy od implementacji. Definicja tego typu jest w pliku nagłówkowym stddef.h)

O przesłanie tego argumentu nie musimy się martwić – kompilator sam przesyła tam wartość wynikającą z rozmiaru obiektu danej klasy.

Przypominam, że skoro funkcja jest statyczna, to nie ma ukrytego argumentu this. Zatem pierwszy argument, to naprawdę pierwszy argument z listy argumentów.

Jakie są konsekwencje tego, że operator ten jest funkcją statyczną?

To mianowicie, że operator ten to nie funkcja charakterystyczna dla istniejących konkretnych obiektów tej klasy. Jest on raczej **zdolnością klasy** do tworzenia nowych obiektów.

Oto przykładowa definicja takiego operatora dla klasy `wekt`, którą ostatnio się zajmowaliśmy. Zauważ, że nie ma słowa `static`. Kompilator i tak zrobi tę funkcję funkcją statyczną.

```
void * wekt::operator new(size_t rozmiar)
{
     cout << "Kreuje obiekt !\n" ;
     return (new char[rozmiar] );
}
```

Funkcja ta po prostu używa globalnego `new`, by zarezerwować odpowiedni obszar. Wskaźnik do tego obszaru zwraca.

Dzięki temu możliwa byłaby teraz taka definicja obiektu klasy `wekt` w zapasie pamięci

```
wekt *wsk ;
     wsk = new wekt ;
```

Napisałem to ostatnie zdanie w trybie przypuszczającym, dlatego że konieczna jest jeszcze jedna rzecz. Może jeszcze pamiętasz, że jeśli klasa ma mieć obiekty tworzone w zapasie pamięci operatorem `new`, to powinna mieć konstruktor domniemany, czyli taki, który można wywołać bez żadnych argumentów. W naszej klasie takiego konstruktora nie było. To nic – zaraz będzie. Wystarczy, że deklarację konstruktora

```
wekt::wekt(float a, float b, float c) ;
```

zamienisz na

```
wekt(float a = 0, float b = 0, float c = 0) ;
```

Od tej pory konstruktor ten można wywołać nawet bez żadnych argumentów.

Co do treści (ciała) samego `operatora new()` to, jak widzisz, nie ma tam nic odkrywczego. W rezultacie bowiem dostaliśmy przeładowany operator `new`, który robi to samo, co globalny operator `new`.

Kiedy zatem przeładowanie tego operatora może się naprawdę przydać ?

Myślę, że np. wtedy, gdy w programie zmuszeni jesteśmy posługiwać się tym operatorem wielokrotnie i czujemy, że można by jakoś to rezerwowanie usprawnić. Na przykład zrobić to hurtem co ileś nowych obiektów. Można też przeładowany `operator new()` zastosować do monitorowania zapasu pamięci. Możemy prowadzić statystykę, jaki procent zapasu pamięci zajmują obiekty tej właśnie klasy.

Skoro wersja globalna tego operatora istnieje także obok wersji przeładowanej, to jeśli mamy klasę `wekt`, dla której zdefiniowany jest `operator new` wówczas posłużenie się globalną wersją operatora `new` wygląda tak:

```
wekt *wsk1, wsk2 ;

     wsk1 = new wekt ;          // wersja dla klasy wekt
     wsk2 = ::new wekt ;        // wersja globalna
```

Widzimy dwa warianty wywołania. Pierwsze uruchamia operator `new` przeładowany na użytek klasy `wekt`. Operator ten zasłania globalny (tradycyjny) operator `new`.

Jeśli z jakichś powodów chcemy, by rezerwacja obiektu tej klasy odbyła się globalną wersją operatora `new`, to przed operatorem stawiamy znak `::` – co oznacza, że chodzi o wersję globalną. (Jest to, jak pamiętamy, powszechna praktyka docierania do zasłoniętych globalnych nazw).

18.16 Operator `delete`

Operator `delete`, służący do oddawania obszarów pamięci rezerwowanych operatorem `new`, ma także swoją wersję globalną, która może być użyta w stosunku do typów wbudowanych, a także w stosunku do klas nie mających swojej, przeładowanej wersji tego operatora

Jeśli chcemy dla danej klasy zdefiniować operator `delete`, to musimy pamiętać o tym, że:

- Funkcja operator `delete` jest statyczną funkcją składową klasy. Nawet jeśli zapomnimy napisać przy niej tego słowa `static`, to kompilator i tak tę funkcję uczyni typu `static`.
- Funkcja operatorowa `delete` musi mieć pierwszy argument typu `void*` (czyli wskaźnik do czegoś nieokreślonego).
- Funkcja zwracać ma typ `void` (czyli nic).

Oto realizacja przeładowanej wersji operatora `delete` dla naszej klasy `wekt`:

```
void wekt::operator delete(void * wsk)
{
     cout << "Kasuje obiekt !\n" ;
     delete wsk;
}
```

W ciele operatora widzimy jeszcze raz operator `delete`. Teraz stoi przy nim nie wskaźnik do obiektu klasy `wekt`, ale wskaźnik do `void`, zatem teraz ruszy do pracy globalna wersja tego operatora.

Oczywiście i tutaj jest możliwe wywołanie globalnej wersji operatora `delete` dla kasowania obiektu klasy `wekt`. Znowu posługujemy się tu operatorem zakresu '`::`'. Załóżmy, że kasujemy te obiekty, które wykreowaliśmy w poprzednim paragrafie.

```
delete wsk1 ;          // wersja przeładowana
::delete wsk2 ;        // zasłonięta wersja globalna
```

Zapytasz pewnie:

> Czy obowiązuje nas konsekwencja – to znaczy jeśli obiekt kreowaliśmy przeładowanym operatorem `new` dla danej klasy – powinniśmy także zlikwidować tym operatorem?

Raczej tak. To oczywiście zależy od tego, co w funkcji operatorowej robione jest dodatkowo. Jeśli w wersji przeładowanej operatora `new` - robisz coś dodatkowo – choćby statystykę narodzin i istniejących obiektów, to likwidacja obiektu operatorem globalnym, nie uaktualni Twojej statystyki oznajmiając zgon.

Dla wtajemniczonych:

Skoro funkcje operatorowe `new` i `delete` są funkcjami z przydomkiem `static` (czyli wywoływane są bez wskaźnika `this`) dlatego nie mogą być one funkcjami wirtualnymi klasy.

To dlatego, że istotą funkcji wirtualnej jest rozpoznawanie (na podstawie wskaźnika `this`), na rzecz obiektu której to klasy (podstawowej czy pochodnej) została wywołana ta funkcja. Funkcje statyczne nie mogą być więc wirtualne. Czyli nasze operatory `new` i `delete` także nie.

18.17 Operatory postinkrementacji i postdekrementacji, czyli koniec z niesprawiedliwością

Gdy mówiliśmy o przeładowaniu jednoargumentowych operatorów inkrementacji i dekrementacji zaznaczyłem, że chodzi o operatory przedrostkowe, czyli takie, które występują przed nazwą obiektu. Innymi słowy chodzi o pre-inkrementację i pre-dekrementację.

```
++i
--i
```

Tymczasem dla typów wbudowanych mamy jeszcze „końcówkową" wersję tych operatorów.

```
i++
i--
```

Operatory te przez długi czas były dyskryminowane w kwestii możliwości przeładowania. Po prostu `operator++` mógł mieć tylko jedna formę - tę przedrostkową (preinkrementacja). Nie można było zatem wobec obiektu danej klasy wykonać tego, co łatwo wykonywało się dla typów wbudowanych.

Pewnie pomyślałeś: „–Bez przesady, przecież można bez tego żyć. Nie jest to aż tak ważny operator."

Cóż, kwestia przyzwyczajenia. Pamiętasz, jak dla typów wbudowanych za pomocą wskaźnika odczytywaliśmy elementy tablicy ? Wystarczyło wyrażenie

```
*(wsk++)
```

i za jednym zamachem odnosiliśmy się do elementu tablicy oraz przeskakiwaliśmy na następny element!

Pomyślałeś: „–No tak, w przypadku wskaźnika jest to bardzo potrzebne, czy jednak jest to tak potrzebne wobec obiektu jakiejś klasy?"

Jak to nie? Przypomnij sobie nasz *zręczny wskaźnik* – klasę, która zajmowaliśmy się niedawno. Klasa ta reprezentowała obiekty będące jakby wskaźnikami. Takimi jak zwykłe, tylko mądrzejszymi. I nagle tu się okazuje, że do mądrzejszego wskaźnika nie można zastosować postinkrementacji, a do głupszego tak!

Te powody sprawiły, że do C++ wprowadzono możliwość przeładowania także i operatorów 'końcówkowych' czyli *występujących za nazwą* obiektu.

Jak to jest zrobione ?

Otóż ponieważ `operator++` **przed**rostkowy jest już zajęty, aby zrobić coś, co pozwoli na definicję drugiego operatora++ dopuszczamy się pewnego oszustwa:

Mimo, że jest to przecież operator pracujący na jednym argumencie, definiujemy go jako operator dwuargumentowy. Nic w tym dziwnego – są przecież operatory, które mają formę i jedno- i dwuargumentową. Jest przecież operator

& jednoargumentowy – pobranie adresu

& dwuargumentowy – iloczyn bitowy

Natomiast operator++ jest tylko jednoargumentowy. Normalnie dla innych operatorów próba definicji operatora wyłącznie jednoargumentowego (np. `operator!`) jako dwuargumentowego byłaby błędem. Kompilator od razu protestuje. Tu jednak, w wypadku `operatora++`, dzieje się to za błogosławieństwem Bjarne S., który presją środowiska zmuszony był wymyślić operator postinkrementacji.

Oto przykład definicji tego operatora dla naszej klasy `wekt`:

```
wekt wekt::operator++(int)
{
     x = x + 1 ;
     y = y + 1 ;
     z = z + 1 ;
     return *this ;
}
```

Jest to funkcja składowa klasy `wekt`, a więc pierwszy argument przychodzi jako wskaźnik `this`. Drugi argument jest, jak widzimy, typu `int`. Oczywiście my tej liczby do funkcji operatorowej nie posyłamy – jest ona generowana automatycznie i oczywiście nieużywana w definicji tej funkcji. W liście argumentów formalnych widzimy tylko nazwę typu. Tak robiliśmy, gdy nie chcieliśmy korzystać z jakiegoś wysłanego do funkcji argumentu (por. paragraf: Nienazwany argument, str. 90)

Mając taką funkcję operatorową możemy zastosować nasz operator w wyrażeniach

```
wekt w(3, 5, 6) ;
```

```
++w                    pre -inkrementacja
w++                    post-inkrementacja
```

Zapis ten równoważny jest następującemu:

```
w.operator++()         pre -inkrementacja
w.operator++(0)        post-inkrementacja
```

Oszustwo polega więc na tym, że gdy kompilator zauważy zapis

```
w++
```

uznaje go za zapis

```
w++ 0                  // normalnie nielegalny !
```

co w normalnych warunkach jest przecież błędem. Jednak widząc to kompilator wywołuje dwuargumentowy operator++.

Zatem dzięki temu mamy dwa różne `operatory++`

- Gdy symbol ++ stoi przed nazwą obiektu – to kompilator wybiera wersję jednoargumentową tego operatora.
- Gdy symbol ++ stoi za nazwą obiektu – kompilator wywołuje wersję dwuargumentową.

Ważne jest tu sformułowanie: **dwa różne**. Bowiem ciało jednej i drugiej wersji może być całkowicie odmienne.

Wszystko, co powiedzieliśmy o operatorze postinkrementacji, dotyczy analogicznie operatora postdekrementacji

Jego przykładowa definicja dla klasy `wekt` wygląda tak:

```
wekt wekt::operator--(int)
{
     x = x - 1 ;
     y = y - 1 ;
     z = z - 1 ;
     return *this ;
}
```

18.18 Rady praktyczne dotyczące przeładowania

Jako się rzekło, mechanizm przeładowania jest możliwością, z której możesz skorzystać lub nie. Chodzi teraz o to, żeby – jeśli już zdecydujemy się na przeładowanie jednego lub kilku operatorów – żeby zrobić to mądrze, posługując się jakąś logiką. Oto kilka rad:

Nie ma sensu przeładowywać wszystkich operatorów dla danej klasy. Może się bowiem okazać, że wykonałeś kawał dobrej, solidnej, nikomu niepotrzebnej roboty.

Które operatory zatem przeładowywać?

Przed przystąpieniem do pracy trzeba się zastanowić, jak taka klasa wygląda z zewnątrz – to znaczy jakie wykonuje się operacje na obiektach danej klasy. Czyli jakie musi ona mieć: publiczne funkcje składowe. Kiedy już to jest jasne, można się zastanowić, które z tych funkcji wygodniej byłoby przeprowadzać za pomocą operatorów. Wybór jest prosty: chodzi o to, z jakim symbolem operatora ta akcja się kojarzy. Pewne operatory od razu narzucą się same, (np. w wypadku klasy `wektorek` porównanie długości dwóch wektorów za pomocą operatorów `==` , `<` , `>`) inne zaś operatory wymusi na nas „wygodnictwo" (- jak w wypadku przeładowania operatora `[]` dla wielkiej tablicy złożonej na dysku).

Nie staraj się przeładowywać na siłę. Jeśli nazwa funkcji składowej lepiej opisuje działanie tej funkcji, niż robi to wygląd operatora, to lepiej pozostać przy funkcji.

Oczywiście jeśli chodzi o dodawanie, to od razu narzuca się `operator+`, ale w przypadku zagwizdania – `operator!` jest już bardzo odległym skojarzeniem.

Nie cuduj i nie przeładowuj bez sensu. Jeśli wpadniesz na wesoły pomysł, żeby w Twojej klasie `operator*` robił odejmowanie, `operator-` dodawanie, a `operator+` mnożenie, Twoje poczucie humoru szybko się wyczerpie, gdy zobaczysz po tygodniu swój własny zapis

```
a + b *(c - d * a)
```

Przeładowanie powinno służyć raczej uproszczeniu czytania, a nie produkcji łamigłówek. Cała wspaniałość przeładowania polega na zbliżeniu zapisu operacji na klasach, do prostoty zapisu operacji na typach wbudowanych. Powtarzam: prostoty !

Jeśli przeładowałeś operator + oraz operator = to nie sądź, że tym samym masz automatycznie `operator+=` albo `operator++`. Są to zupełnie inne funkcje operatorowe i jeśli chcesz się nimi posługiwać wobec obiektów danej klasy, to musisz je także przeładować.

Przykładem na granicy żartu jest przeładowanie operatora - oraz > i spodziewanie się, że tym samym mamy operator -> .

Mimo całej dowolności treści przeładowanych funkcji operatorowych - staraj się zachować logikę pewnych zależności między operatorami.

Jeśli dla typów wbudowanych poniższe wyrażenia są równoważne

```
a = a + 1
a += 1
a++
```

to dobrą praktyką jest trzymanie się tej konsekwencji dla klasy, która ma takimi operatorami się posługiwać. Chodzi tu o nic więcej, jak o siłę przyzwyczajenia, ale jest to wzgląd bardzo ważny.

Podobnie z zależnością operatorów dostępu do składników klasy

```
wskaznik -> skladnik
(*wskaznik) . skladnik
wskaznik[0] . skladnik
```

Jeśli operator jest „nieszkodliwy" dla typu wbudowanego – to znaczy nie zmienia wartości zmiennej, na której pracuje, to staraj się, by jego odpowiednik dla klasy również niczego nie zmieniał wewnątrz obiektu.

Na przykład jednoargumentowy `operator-` zastosowany wobec obiektu

```
int i = 4 ;
     (-i)
```

nie zmienia wartości zmiennej `i`. Jest ona nadal ta sama (4). To tylko wyrażenie (-i) jako całość ma wartość –4

Wartość zwracana przez operator jest bardzo ważna. Dzięki temu, że operator nie tylko wykonuje działanie, ale też zwraca rezultat możliwe są wyrażenia „kaskadowe"

```
a + b + c + d
```

gdzie najpierw odbywa się jedno dodawanie, potem jego rezultat staje się składnikiem drugiego dodawania i kolejny rezultat staje się składnikiem trzeciego.

Prawie wszystkie operatory mogą zostać przeładowane na dwa sposoby - jako funkcja globalna lub funkcja składowa. Który z nich wybrać ? Porozmawiajmy o tym.

18.19 Pojedynek: Operator jako funkcja składowa, czy globalna

Skoro ten sam operator można zdefiniować jako funkcję składową albo funkcję globalną, to nasuwa się pytanie: jak lepiej zrobić?

Jednoznacznej odpowiedzi nie ma. Zależy to od tego, czego oczekujemy od operatora. Ogólnie można powiedzieć, że:

> Jeśli operator zmienia w jakiś sposób obiekt, na którym pracuje, to powinien być zdefiniowany jako funkcja składowa jego klasy.
> Operatory te wtedy zwracają l-wartość.

*Przykładem są tu takie operatory jak =, ++, --, czy też wszystkie operatory w stylu +=, *=, itd. Operatory te (przynajmniej w stosunku do typów wbudowanych) modyfikują obiekt stojący po ich lewej stronie.*

> Jeśli natomiast operator sięga po obiekt po to, by pracować z nim bez modyfikowania go –
> to wówczas raczej stosuje się operator w postaci funkcji globalnej.

Przykładem takich operatorów są choćby +, –, /, &, ! Nie modyfikują one obiektu (obiektów) koło których stoją. Argument biorący udział w sumowaniu – sam nie ulega przecież zmianie.

Skąd jest taka zasada? Wynika ona po prostu z naszych przyzwyczajeń do tego, jak te operatory zachowują się w stosunku do typów wbudowanych.

> Jeśli operator ma dopuszczać, by po jego lewej stronie stał typ wbudowany, to nie może być funkcją składową. Musi być globalną.

Chodzi o zapis

```
x + 2
2 + x
```

gdzie x jest obiektem jakiejś klasy. Tego drugiego wyrażenia nie da się stosować, gdy `operator+` jest funkcją składową. Aby przekonać się dlaczego – wystarczy tę linijkę zapisać sobie w postaci jawnego wywołania operatora

```
2.operator+(x)                                  // !!!
```

To oczywiście bezsens dlatego, że nie można zdefiniować operatora jako funkcji składowej klasy `int`. Klasy `int` po prostu nie ma – jest to typ wbudowany.

Jeśli jednak funkcja operatorowa jest zrealizowana jako funkcja globalna, to taki zapis jawnego wywołania operatora wygląda następująco:

```
operator+(2, x)
```

Z tej zasady wynika następna będąca jej uogólnieniem.

> Gdy chcemy dopuścić dwa sposoby używania operatora,
>
> ```
> KlasaX objx ;
> KlasaY objy ;
>
> objx + objy
> objy + objx
> ```
>
> to definiujemy ten operator jako funkcję globalną.
> Dokładniej: jako dwie operatorowe funkcje globalne

```
operator+(KlasaX, KlasaY) ;
operator+(KlasaY, KlasaX) ;
```

To, czy obie będą realizować tę samą akcję, zależy już od tego, co napiszemy w ciele tych funkcji operatorowych. Jeśli napiszemy to samo w obu, to matematycy powiedzą, że dodawanie takich obiektów jest przemienne.

Dokonując wyboru sposobu realizacji operatora należy także pamiętać, że:

> Jeśli używamy operatora, który zdefiniowany jest jako funkcja składowa, wówczas w stosunku do jego pierwszego argumentu nie może zajść żadna niejawna konwersja.

Mowa o tym argumencie, który zostaje przesłany za pomocą wskaźnika `this`.

Czasem to dobrze, czasem źle. Jeśli chcemy, by na **obu** argumentach `a` i `b`, w wyrażeniu

```
a + b
```

mogły zajść w razie potrzeby niejawne konwersje, to wówczas należy zdefiniować operator jako funkcję globalną. Jeśli nie życzymy sobie, by na pierwszym argumencie zaszła jakakolwiek konwersja, to definiujemy funkcję operatorową jako funkcję składową w klasie tego argumentu.

Łatwo to sobie uzmysłowić i zapamiętać tak. Oto zapis wyrażenia `(a+b)` w formie jawnego wywołania funkcji operatorowej:

```
a.operator+(b)          // gdy operator jest f. składową
operator(a, b)          // jeśli operator jest f. globalną
```

Konwersje niejawne mogą zostać wykonane tylko dla tych obiektów, które są wewnątrz nawiasu wywołania funkcji `operator+` .

W pierwszej wersji obiekt a nie jest w nawiasie, więc na nim niejawna konwersja nie może się odbyć.

18.20 Zasłona spada, czyli tajemnica operatora <<

Właściwie już od pierwszych stron tej książki posługujemy się zapisem, w którym występuje operator <<

```
cout << "Witamy na pokladzie \n" ;
```

Skądinąd wiemy, że dwuargumentowy `operator<<` jest, w stosunku do typu wbudowanego `int`, operatorem powodującym przesunięcie bitów o żądaną liczbę pozycji. Jak to więc możliwe, że taki zapis

```
int m = 2 ;
          cout << m ;
```

powoduje wyprowadzenie na ekran liczby zapisanej w zmiennej typu `int`? Odpowiedź jest prosta. Mamy tu do czynienia z najzwyklejszym przeładowaniem operatora << . Zapis ten jest inaczej rozumiany tak:

```
cout.operator<<(m) ;
```

`cout` jest egzemplarzem obiektu klasy, która się nazywa `ostream`. [skrót od: Output STREAM – strumień wyjściowy]. To dla tej klasy dokonano przeładowania operatora. Przeładowanie jest możliwe, gdy jednym z argumentów jest obiekt typu zdefiniowanego przez użytkownika. Takim typem zdefiniowanym – choć bibliotecznym – jest właśnie klasa `ostream`.

W naszym zapisie po lewej stronie operatora << stoi obiekt klasy `ostream`, a po prawej typ wbudowany `int`. Wywoływana jest wówczas funkcja operatorowa zajmująca się wypisaniem na ekran liczby `int`.

Pytanie: Kto napisał tę funkcję operatorową?

Odpowiedź: Twórcy klasy `ostream`. Nie jest ona częścią samego języka C++, ale zawiera ją biblioteka standardowa. Jeśli jakaś firma produkuje kompilator

C++, to nie do pomyślenia jest, by do niego nie dołączyła biblioteki, w której znajdzie się definicja tej klasy. To, jak jest zbudowana ta klasa, jest dla nas nieistotne. My chcemy tylko z niej łatwo korzystać i mieć możliwość wypisywania na ekranie liczby typu `int`. Przeładowanie operatora << właśnie daje tę łatwość.

Oczywiście jest kilka wersji przeładowania operatora << na okoliczność pracy z różnymi typami argumentów np. `float`, `char*` . Posługiwaliśmy się już tym wielokrotnie, wypisując na ekranie liczby typu `float`, czy stringi.

Jednak ta, napisana kilka lat temu w USA, klasa biblioteczna `ostream` i dostarczona nam w wersji binarnej – nic nie wie o naszej klasie `wektorek`, którą napisaliśmy sobie wczoraj po kolacji. Dlatego nie możemy sobie obiektu klasy `wektorek` postawić obok operatora << w instrukcji

```
wektorek w ;
          cout << w ;          // jeszcze niepoprawne
```

Tu z pomocą może nam przyjść przeładowanie operatora << . Możemy po raz kolejny przeładować operator << . Jest on dwuargumentowy, więc argumenty będą typu `ostream` i `wektorek`.

Już słyszę jak protestujesz: „–Jak to? -Tak bez wiedzy i zgody klasy `ostream`?"

Tak! Właśnie bez wiedzy i zgody klasy `ostream`.

Pamiętasz – mówiłem, że funkcja operatorowa może być zwykłą globalną funkcją. Nie musi być nawet zaprzyjaźniona z klasą.

> *Tu właśnie okazuje się, że gdyby musiała być zaprzyjaźniona to byłoby źle. Dlatego, że w niej musiałaby być deklaracja przyjaźni z naszą klasą `wektorek`. Rozumiesz co to znaczy? To znaczy, że abyśmy mogli skorzystać z przeładowania operatora << na rzecz tej klasy `ostream` i klasy `wektorek` - programista, piszący te kilka lat temu w USA klasę `ostream`, musiałby umieścić w swojej klasie deklarację przyjaźni z naszą klasą `wektorek`.*

Dzięki temu, że przyjaźń nie jest wymagana do przeładowania – piszemy sobie funkcję `operator<<` taką, w której argument typu `ostream` stoi na pierwszym miejscu, a nasz argument typu `wektorek` na drugim.

Co tracimy przez to, że klasa `ostream` nie deklaruje z nami przyjaźni?

Tracimy dostęp do niepublicznych składników klasy `ostream`. Tyle, że nam na tym dostępie wcale nie zależy. Po prostu nie interesują nas wszystkie trybiki i kółka tej klasy. My chcemy tylko jej używać.

Jest więc oczywiste, że nasza funkcja nie może być funkcją składową klasy `ostream`. Po prostu nie możemy modyfikować wnętrza tej klasy bibliotecznej. Zatem ta możliwość odpada.

Są jednak jeszcze 2 inne możliwości realizacji tej funkcji `operator<<`. Może to być:

- – funkcja globalna,
- – funkcja składowa klasy `wektorek`.

Ta ostatnia ewentualność odpada z prostego powodu: pierwszym argumentem naszego operatora << musi być argument klasy `ostream`. Tymczasem, gdy-

byśmy ów operator uczynili funkcją składową klasy `wektorek`, to pierwszym argumentem byłby ukryty wskaźnik `this` do obiektu klasy `wektorek`.

Nic w tym strasznego, w końcu kolejność argumentów można by przestawić, ale sam popatrz, jak wówczas wyglądałby zapis:

```
wektorek w ;
          w << cout ;                    // !!!
```

Jest to zapis dziwaczny. Nie chcemy tak, bo już przyzwyczailiśmy się do odwrotnego.

Zostaje więc ewentualność druga: realizacja funkcji operatorowej jako funkcji globalnej. W tym wypadku, nie ma wymogów co do kolejności argumentów. Możemy wybrać jak chcemy, oczywiście wybieramy kolejność:

(ostream, wektorek)

Właściwie już by można napisać funkcję `operator<<` , ale wstrzymajmy się sekundę. Zastanówmy się, co z klasy `wektorek` będzie wypisywane na ekran: tylko składniki publiczne, czy też także niepubliczne.

Jeśli bowiem chcemy korzystać ze składników niepublicznych klasy `wektorek`, to klasa `wektorek` powinna nam na to pozwolić, czyli powinna zadeklarować przyjaźń z naszą funkcją operatorową.

Jeśli będziemy korzystać tylko z publicznych składników klasy `wektorek`, to przyjaźń nie jest potrzebna. Każdy może odczytać publiczne składniki, zatem może to też globalna funkcja `operator<<`.

Oto krótki program:

```
#include <iostream.h>
//////////////////////////////////////////////////////////
class wektorek {
public :
     float x, y, z ;
     //—— konstruktor
     wektorek(float a=0, float b=0, float c=0): x(a),
                                                y(b), z(c)
     { }
} ;
//////////////////////////////////////////////////////////
/*******************************************************/
// globalna funkcja operatorowa
// realizujaca przeładowania << dla naszej klasy wektorek
/*******************************************************/
ostream & operator<<(ostream & ekran , wektorek & w)//❶
{
     ekran << "wspolrzedne wektora : " ;
     ekran << "(" ;
     ekran << w.x ;                                  // ❹

     ekran << ", " << w.y                            // ❺
           << ", " << w.z << ")" ;
     return ekran ;                                  // ❻
}
/*******************************************************/
main()
```

```
{
     wektorek  w(1,2,3) ,
               v ,
               k(-10, -20, 100);

     cout << "Oto nasze wektory \nwektor w -" ;
     cout << w ;                                          //❷

     cout << "\nwektor v -" << v
          << "\nwektor k -" << k
          << endl ;

     cout << "Wywolanie jawne \n" ;
     operator<<( cout , w) ;                              //❸
}
```

Po wykonaniu tego programu na ekranie pojawi się

```
Oto nasze wektory
wektor w -wspolrzedne wektora : (1, 2, 3)
wektor v -wspolrzedne wektora : (0, 0, 0)
wektor k -wspolrzedne wektora : (-10, -20, 100)
Wywolanie jawne
wspolrzedne wektora : (1, 2, 3)
```

Ciekawsze punkty programu

❶ Zwróć uwagę na argumenty przesyłane do funkcji operatorowej. Porównajmy to z wywołaniem tej funkcji operatorowej

```
cout << w ;
```

czyli

```
operator<<( cout, w) ;
```

Oba te typy wywołań funkcji operatorowej występują w `main` w miejscach oznaczonych jako ❷ i ❸.

Zatem w definicji funkcji operatorowej ❶ widzimy, że argument `cout` odebrany zostaje przez referencję. To przezwisko w funkcji brzmi `ekran`. Także i obiekt klasy `wektorek` przesyłamy przez referencję. To przyspiesza przesłanie dużych obiektów. Wektorek nie jest duży, więc równie dobrze można by przesłać go przez wartość.

❹ Symbole `<<` , które widzisz wewnątrz funkcji operatorowej, to już nie jest wywołanie wersji przeładowanej. Zauważ, że po ich lewej stronie stoi obiekt klasy `ostream`, a po prawej stronie czasem typ `char*` (stringi), czasem typ `float` (składniki x, y, z).
Co prawda przeładowania na okoliczność argumentów

```
(ostream, float)
```

nie robiliśmy, ale zrobił to za nas programista, który pisał tę klasę `ostream` — bowiem `char*` i `float` są typami wbudowanymi, a więc powszechnie znanymi już w czasie pisania tej klasy.

❻ Zastanówmy się, co powinien zwracać jako rezultat taki `operator<<` . Sprawa jest prosta, nie musi nic zwracać. Jego rolą jest tylko wypisywanie na ekran. Jednak jeśli zdecydujemy się zwracać rezultat będący referencją do obiektu klasy `ostream`, to dzięki temu zamiast zapisu

```
cout << w ;
cout << v ;
cout << k ;
```

możemy także stosować zapis

```
cout << w << v << k ;
```

Jak to możliwe? Otóż, ostatni zapis możemy zrozumieć jako †)

```
( (cout << w) << v) << k ;
```

Już widać odpowiedź. Wyrażenie

```
(cout << w)
```

jako całość musi mieć rezultat będący odpowiednikiem `cout`.

Powyższą linijkę można zapisać też jako

```
(operator<<(cout, w) )
```

Zatem, abyśmy osiągnęli nasz cel, funkcja operatorowa powinna zwrócić jako rezultat – referencję do pierwszego przysłanego do niej argumentu.

Tak robimy w naszym wypadku. Zwróć uwagę na linijkę ❻, a także na deklarację typu rezultatu zwracanego przez funkcję `operator<<` ❶.

Zastosowany przez nas sposób jest zgodny z tym, co zastosował projektujący standardową klasę `ostream`, dzięki tej zgodności możemy w jednej kaskadzie mieszać jego i nasze przeładowanie.

```
cout << "tekst" << w << 5 ;
```

W naszej klasie `wektorek` składniki `x`, `y`, `z` są publiczne. Co by było, gdyby były prywatne?

Jeśli klasa `wektorek` chce, by operator << mógł do tych składników bezpośrednio zajrzeć, musi je operatorowi udostępnić za pomocą deklaracji przyjaźni. Słowem: w takim wypadku w definicji klasy powinna się znaleźć deklaracja przyjaźni z operatorem

```
ostream & operator<<(ostream &ekran , wektorek &w);
```

To wszystko.

†) Przypominam, że operator << jest **zawsze** lewostronnie łączny.

Można by zapytać: no to właściwie na rzecz której klasy przeładowaliśmy operator – na rzecz klasy `ostream`, czy klasy `wektorek`?

Tak sprawy stawiać nie można. Tak samo, jakbyś przy wyrażeniu

```
2 + 3.14
```

zapytał dla jakiego typu pracuje właściwie znaczek +. Dla typu `int` czy dla typu `float`?

Operator przeładowany jest na okoliczność, gdy po jego lewej stronie znajdzie się obiekt klasy `ostream`, a po jego prawej obiekt klasy `wektorek`. Wymaga to takiej funkcji operatorowej, której pierwszym argumentem będzie typ `ostream`, drugim typ `wektorek`.

Funkcję operatorową, która pracuje na dwóch argumentach klasy `A` i `B` można zrealizować na trzy sposoby:

1) albo jako funkcję składową klasy A,

2) albo jako funkcję globalną,

3) albo jako funkcję składową klasy `B`.

W naszym wypadku pierwszy sposób odpadł, bo nie chcieliśmy grzebać w bibliotecznej klasie `ostream`, którą dostaliśmy w wersji binarnej[†)].

Trzeci sposób odpadł, bo uparliśmy się, że argumentem z lewej strony znaku << ma być typ `ostream`, a nie typ `wektorek`. Jak wiadomo funkcja składowa ma zawsze jako pierwszy argument – obiekt swojej własnej klasy.

Został drugi sposób – jako funkcja globalna. Tak też postąpiliśmy.

Myślę, że rozumiesz, iż sprawa przeładowania operatora >> wyrażeniu

```
wektorek w ;
     cin >> wekt ;
```

wygląda analogicznie. Z tą tylko różnicą, że mamy tu do czynienia z klasą `istream` (input stream – ang. strumień wejściowy), a jej konkretnym obiektem jest właśnie `cin`. Oto realizacja tej funkcji operatorowej:

```
istream & operator>>(istream & klawiatura , wektorek &w)
{
     cout << "Wspolrzedna x : " ;
     klawiatura >> w.x ;
     cout << "Wspolrzedna y : " ;
     klawiatura >> w.y ;
```

†) Dla wtajemniczonych: ostatecznie byłyby na to bezpieczne sposoby – przez dziedziczenie.

```
    cout << "Wspolrzedna z : " ;
    klawiatura >> w.z ;

    return klawiatura ;
}
```

Dzięki przeładowaniu możemy wpisywać ręcznie dane o współrzędnych. Jeśli w programie znajdzie się sekwencja

```
wektorek s ;

    cout << "Podaj dane dla wektora s \n" ;
    cin >> s ;
    cout << "Wypisujemy to co dostalismy \n"
         << s ;
```

Wówczas na ekranie zobaczymy

```
Podaj dane dla wektora s
Wspolrzedna x : 11
Wspolrzedna y : 22
Wspolrzedna z : 33
Wypisujemy to co dostalismy
wspolrzedne wektora : (11, 22, 33)
```

O tych operatorach mówić będziemy dokładniej w rozdziale poświęconym operacjom wejścia/wyjścia.

Tutaj tylko nadmienię, że o wiele ładniej by było, gdyby oba zdefiniowane przez nas operatory << i >> były „symetryczne". To znaczy tekst wypisywany na ekran był identyczny z tekstem, którego spodziewamy się przy wczytywaniu z klawiatury. Słowem: na klawiaturze powinniśmy wystukać tylko

```
(11, 22, 33)
```

a nie oczekiwać dodatkowych pytań o każdą ze współrzędnych.

18.21 Rzut oka wstecz

Tym sposobem zakończyliśmy rozdział, który mógł Ci się wydać najtrudniejszym rozdziałem tej książki. Chciałbym Cię jednak uspokoić. Przeładowanie operatora to technika, na którą zdecydujesz się tylko czasem. Wówczas wrócisz do tego rozdziału, by przeczytać o tym operatorze, który potrzebujesz.

Spróbujmy uporządkować sobie w pamięci zagadnienia z tego rozdziału.

- Rozmawialiśmy o przeładowywaniu operatorów jedno- i dwuargumentowych. Jest to technika bardzo prosta i nie niosąca żadnych niebezpieczeństw.
- Poznaliśmy operator przypisania oferowany nam w prezencie przez kompilator i zobaczyliśmy, że w stosunku do klas, których składnikami są wskaźniki - taki operator może być niezadowalający.
- Poznaliśmy przeładowanie operatora [ ], którego przeładowanie nie jest niczym trudnym, ale jeśli chcielibyśmy, by wyrażenie z tym operatorem

mogło czasem stać po lewej stronie znaku =

```
a[4] = 10;
```

to musimy spełnić warunek, by w definicji tego operatora pamiętać o znaczku &

- Poznaliśmy przeładowanie operatora ( ), które jest najłatwiejszym z możliwych. Dowiedzieliśmy się, że tylko ten operator pozwoli wysłać do funkcji operatorowej więcej niż dwa argumenty.
- Poznaliśmy przeładowanie operatora ->, który może nam się przydać wtedy, gdybyśmy chcieli zdefiniować klasę obiektów zachowujących się jak wskaźniki.
- Poznaliśmy przeładowanie operatorów `new` i `delete`, o którym zwykle w pierwszej chwili myśli się, jak o czymś trudnym - jednak jest to stosunkowo proste.
- Dowiedzieliśmy się jaką sztuczkę trzeba zastosować, by mieć operatory postinkrementacji i postdekrementacji. W gruncie rzeczy chodzi tylko o to, by wewnątrz nawiasu napisać `int` i nic więcej!
- Zobaczyliśmy jak przeładowano operatory >> i << by służyły do wypisywania na ekran i wczytywania z klawiatury. Nie jest to nic trudnego - i dość prosto można to robić w stosunku do obiektów dowolnej klasy. Dlatego radzę pobawić się takim przeładowaniem w stosunku do jakiejś wymyślonej przez siebie prostej klasy.

Już to kiedyś mówiłem, ale teraz przypomnę: przeładowanie operatorów nie wprowadza „nowej jakości" do języka C++. To tylko inny sposób wywoływania funkcji. Sposób, który może nam uprościć zapis.

Jest to więc sympatyczne udogodnienie, ale nie wejście w inny wymiar. W ten inny wymiar wejdziemy teraz dzięki zagadnieniom omawianym w następnych dwóch rozdziałach.

Koniec Tomu Drugiego

19 Dziedziczenie

19.1 Istota dziedziczenia

Dziedziczenie to technika pozwalająca na definiowanie nowej klasy przy wykorzystaniu klasy już wcześniej istniejącej.

Załóżmy, że mamy klasę

```
class punkt {
public:
     float x, y ;

     punkt(float, float );          // konstruktor
     void wypisz() ;
     void przesun(float, float) ;
};
```

Załóżmy też, że bardzo napracowaliśmy się, aby zdefiniować tę klasę z jej wszystkimi funkcjami składowymi. Wreszcie wszystko mamy. I oto w programie wynika konieczność użycia dodatkowej klasy – podobnej do tej, z tym, że różniącej się w kilku szczegółach.

Czy trzeba wobec tego pisać definicję klasy od nowa? Czy nie można by po prostu powiedzieć: „Chcę mieć klasę taką, jak tamta, z małymi różnicami. Oto te różnice..."

Dobra wiadomość: można tak zrobić! Służy do tego specjalna technika zwana dziedziczeniem, albo inaczej: tworzeniem klas pochodnych.

Sprawa jest bardzo prosta. Jeśli na przykład chcemy, by nowa klasa miała dodatkowy składnik – daną

```
char opis[10] ;
```

oraz inną wersję funkcji `wypisz` – taką, która wypisze nie tylko współrzędne punktu, ale i jego nazwę (schowaną w tablicy opis) – to tę nową klasę definiujemy tak:

```
class lepszy_punkt : public punkt {
public :
     char opis[10] ;
                                  //———————————————— konstruktor
     lepszy_punkt(float =0, float =0, char* =NULL);
     void wypisz() ;              // <—— dodatkowa funkcja
} ;
```

Mamy dzięki temu nową klasę. Klasa `lepszy_punkt` wywodzi się od klasy `punkt`. Mówimy, że klasa `lepszy_punkt` jest **klasą pochodną** klasy `punkt`. Natomiast klasa `punkt` jest dla klasy `lepszy_punkt` **klasą podstawową**.

W klasie pochodnej możemy:

- – zdefiniować dodatkowe dane składowe,
- – zdefiniować dodatkowe funkcje składowe,

  Definiowanie nowych funkcji składowych – bez definiowania dodatkowych danych składowych także ma sens. Jest to jakby wyposażenie klasy w nowe zachowania;
- – zdefiniować składnik (najczęściej funkcję składową), który istnieje już w klasie podstawowej.

  Powoduje to jakby korektę czegoś, co nam z klasy podstawowej nie bardzo odpowiadało i czego nie chcemy w tym kształcie dziedziczyć.

Zwróć uwagę na pierwszą linijkę definicji klasy `lepszy_punkt`. Po dwukropku jest napisane, że klasa `lepszy_punkt` wywodzi się od klasy `punkt`. To wyrażenie po dwukropku to tzw. **lista pochodzenia**. Na tej liście umieszczona jest informacja od czego wywodzi się dana klasa pochodna.

Dzięki temu zapisowi klasa pochodna dziedziczy wszystkie składniki klasy `punkt`. Zatem składnikami klasy `lepszy_punkt` są teraz także np.:

```
x, y, void przesun(float, float)
```

Do powyższych składników możemy odnosić się tak, jakby były one zdefiniowane w zwykły sposób w obrębie klasy `lepszy_punkt`.

Na liście pochodzenia obok nazwy klasy podstawowej `punkt` widzimy też specyfikator dostępu – w tym przypadku jest to słowo `public`. Szczegółowo o tym będziemy mówić w jednym z następnych paragrafów. Tu tylko, dla wyjaśnienia dodam, że ten specyfikator dostępu odpowiada za to, czy odziedziczone nieprywatne składniki klasy podstawowej mają w klasie pochodnej wejść w skład części `private`, `protected`, czy `public`.

Czy oznacza to, że dziedziczenie klasy to jakby przelanie zawartości jednego garnka do drugiego, trochę większego?

Nie. Albowiem w rezultacie składniki nie mają tego samego zakresu.

Dziedziczenie to jakby zagnieżdżanie się zakresów

Składniki odziedziczone mają zakres klasy podstawowej, a na to nakłada się zakres klasy pochodnej. Odpowiada to jakby takiemu zagnieżdżaniu bloków

```
{                        // <———— zakres klasy podstawowej
     int x ;             // składnik odziedziczony
```

```
        int k ;
        {                          // ←———zakres klasy pochodnej
            int x ;                // skł. klasy pochodnej
            int m ;
        }
    }
```

Jak widzimy, jeśli w klasie podstawowej i klasie pochodnej są składniki o tej samej nazwie, wówczas w zakresie klasy pochodnej składnik z tej klasy zasłania odziedziczony składnik z klasy podstawowej. Skoro występuje zasłanianie, to oczywiście nie ma mowy o zlewaniu czegoś do jednego garnka. To właśnie zagnieżdżanie zakresów.

Wróćmy do naszych klas `punkt` i `lepszy_punkt`. Zauważ, że w definicji klasy podstawowej `punkt` jest taka sama funkcja składowa `wypisz`. W rezultacie w klasie pochodnej są dwie funkcje `wypisz`. Funkcja `wypisz` z klasy pochodnej zasłania funkcję `wypisz` z klasy podstawowej. Powtarzam: zasłania. Nie jest to przeładowanie, gdyż funkcje mają inny zakres ważności. Pierwsza ma zakres ważności klasy podstawowej, a druga zakres ważności klasy pochodnej.

> Jeśli mimo wszystko to do Ciebie nie przemawia, to spójrz na listę argumentów obu funkcji. Tutaj akurat tak się dobrze składa, że te listy argumentów są identyczne. Czy pamiętasz, jak kiedyś mówiliśmy: w jednym zakresie ważności nie może być funkcji o identycznej nazwie i identycznej liście argumentów. To chyba najlepszy dowód na to, że nie ma tu przeładowania.

Istnieją więc dwie funkcje `wypisz`: pierwsza mająca zakres klasy podstawowej, a druga mająca zakres klasy pochodnej. Ile razy pracując na obiektach klasy pochodnej wywołana zostanie funkcja `wypisz`, to kompilator aktywuje tę z klasy pochodnej.

Czy to znaczy, że - jeśli jakiś składnik (dana lub funkcja) jest zasłonięty - to nie można się do niego odnosić?

Można. Trzeba jednak uczynić to posługując się kwalifikatorem zakresu. Chodzi o dwa dwukropki zwane inaczej - operatorem zakresu. W naszym wypadku do zasłoniętej funkcji `wypisz` odnosimy się tak,

```
lepszy_punkt obiekt ;          // definicja obiektu

obiekt.wypisz();               // wywołanie funkcji wypisz
                               // z klasy pochodnej „lepszy_punkt"

obiekt.punkt::wypisz();        // wywołanie funkcji wypisz
                               // z klasy podstawowej „punkt"
```

Wywołując funkcję `wypisz` bez tego kwalifikatora zakresu – uruchomimy tę z klasy pochodnej.

Jeszcze raz okazało się, że nie jest to przelanie z mniejszego garnka do większego. W klasie pochodnej tworzy się jakby część, która jest – że tak powiem – dziedzictwem po klasie podstawowej. To bardzo ważna właściwość.

> Definiując obiekt klasy pochodnej powinniśmy pamiętać, że równocześnie w jego wnętrzu tkwi odziedziczony fragment będący jakby obiektem klasy podstawowej.

Bardzo ważna uwaga:

Nie należy jednak sądzić, że to obiekty mogą tworzyć obiekty pochodne. Dziedziczenie dotyczy **klas** (czyli typów danych), a nie jakichś konkretnych egzemplarzy obiektów.

Porównaj:

Jeśli mamy klasę „Volkswagen" i chcemy by z niej wywodziła się klasa „Volkswagen_kabriolet", to nie możemy się spodziewać, że w fabryce samochodów bierze się zwykły obiekt klasy Volkswagen odcina mu się dach i powstaje obiekt klasy Volkswagen_kabriolet.
Każdy wie, że robi się inaczej. Bierze się plany konstrukcyjne klasy „Volkswagen" (czyli definicję klasy Volkswagen), następnie dokonuje się na tych planach zmian zaznaczając różnice (definicja klasy pochodnej przy wykorzystaniu klasy podstawowej). Powstają nowe plany zwane „Volkswagen_kabriolet". Dopiero na podstawie tych nowych planów (definicja klasy pochodnej) fabryka samochodów może konstruować samochody klasy Volkswagen_kabriolet.

19.2 Dostęp do składników

Nasz ostatni przykład był najprostszy z możliwych: wszystkie składniki obu klas były publiczne. Istnieją jednak sposoby określania w jaki sposób składniki klasy podstawowej zostają dziedziczone. To znaczy określania, jaki w rezultacie mają dostęp w klasie pochodnej. Mogą tu być dwie sytuacje.

19.2.1 Prywatne składniki klasy podstawowej

Są dziedziczone – (jak wszystkie inne składniki), ale w zakresie klasy pochodnej nie ma do nich dostępu. A więc niby są, ale nie można ich ruszyć.[†)]

Wiem co pomyślałeś: „-No to po co mi takie składniki?"

Zauważ jednak, że napisałem, iż nie ma się do nich dostępu **w zakresie klasy pochodnej.** To nic. Mogą być przecież odziedziczone jakieś funkcje składowe nieprywatne. Ponieważ funkcje te mają zakres klasy podstawowej – one mogą pracować na prywatnych składnikach swojej klasy. Z kolei jeśli same są nieprywatne, to z zakresu klasy pochodnej możemy je uruchomić. Mogą więc one dla nas zrobić to, czego nie mogliśmy zrobić bezpośrednio. Przypomina się tutaj porzekadło o wyjmowaniu z ognia kasztanów cudzymi rękami.

Oto przykład:

†) Dla wtajemniczonych: zakładam, że nie ma żadnej deklaracji przyjaźni klasy pochodnej z klasą podstawową

```
#include <iostream.h>
//////////////////////////////////////////////////////////
// klasa podstawowa
//////////////////////////////////////////////////////////
class ryba {
private :
     int a ;
protected:
     int prots ;
public:
     int pubs ;

     void wstaw(int m ) { a = m ; }
     int czytaj() { return a ; }
};
//////////////////////////////////////////////////////////
// klas pochodna
//////////////////////////////////////////////////////////
class rekin : public ryba {
     float x ;
public :
     void funk();
} ;
/*******************************************************/
void rekin::funk()
{
     x = 15.6 ;                                    // ❶
     //a = 6 ;                                     // ❷
     wstaw(6) ;                                    // ❸
     cout << "\n wyjety funkcja 'czytaj' skladnik a="
          << czytaj() ;                            // ❹
     prots = 77 ;                                  // ❺
     pubs = 100 ;
     cout << "\n bezposr. odczytany skladnik protected = "
          << prots
          << "\n bezposr. odczytany skladnik public = "
          << pubs ;
}
/*******************************************************/
main()
{
rekin wacek ;
     wacek.funk() ;
}
```

Po uruchomieniu programu na ekranie zobaczymy

```
wyjety funkcja 'czytaj' skladnik a=6
bezposr. odczytany skladnik protected = 77
bezposr. odczytany skladnik public = 100
```

Komentarz

Na początku widzimy klasę `ryba`, w której są składniki `private`, `protected` oraz `public`. Są w niej też dwie funkcje składowe pracujące na składniku prywatnym o nazwie `a`.

Klasa `rekin` – jak to widzimy na liście pochodzenia – jest klasą pochodną od klasy `ryba`. Ma ona funkcję `funk`. Wnętrze tej funkcji interesuje nas najbardziej. We wnętrzu tej funkcji znajdujemy się w zakresie ważności klasy pochodnej. W tej funkcji sytuacja przedstawia się następująco:

❶ Dostęp do prywatnych składników klasy pochodnej jest możliwy. To wiemy i stosujemy od dawna.

❷ Dostęp do prywatnych składników klasy podstawowej jest niemożliwy. Te linijkę musiałem ująć w komentarz, gdyż inaczej kompilator protestował.

❸ Jedynym sposobem wpisania tam tej liczby 6, jest wywołanie funkcji składowej klasy `ryba`, która to funkcja ma prawo to zrobić. (Robi to przecież w swojej klasie).

❹ Podobnie, w celu odczytania zapisanej tam wartości, posługujemy się (inną) funkcją składową klasy podstawowej.

❺ Natomiast do składników `protected` i `public` klasy podstawowej mamy dostęp bezpośredni.

W przykładzie tym wybiegliśmy nieco do przodu. Za to wiemy już, jak zachowują się inne składniki klasy podstawowej. Sformułujmy to jeszcze raz.

19.2.2 Nieprywatne składniki klasy podstawowej

czyli składniki `protected` i `public` są dostępne w zakresie klasy pochodnej – bezpośrednio.

Tutaj po raz pierwszy wyjaśnia się sekret, jaka jest różnica między składnikiem `protected`, a składnikiem `private` – które do tej pory traktowaliśmy identycznie – tak, jakby oba miały dostęp `private`.

Słowo `protected` znaczy, że składnik jest zastrzeżony dla klas pochodnych. To znaczy, że:

- dla całego świata ma być on tak, jakby był `private` - czyli niedostępny,

natomiast

- dla wszystkich klas pochodnych od danej klasy będzie on dostępny tak, jakby był `public`.

Zatem: słowo `protected` zostało wymyślone na okoliczność dziedziczenia, po to, by udostępnić klasom pochodnym to, co nie jest dostępne szerokiej publiczności.

Wniosek z tego paragrafu:

klasa podstawowa przez odpowiednią specyfikację dostępu słowami `public`, `protected`, `private` może decydować, które składniki zamierza udostępnić szerokiej publiczności, które zastrzega tylko dla swoich klas pochodnych, a które zachowuje tylko na swój prywatny użytek.

19.2.3 Klasa pochodna też decyduje

Istnieje jeszcze drugi mechanizm za pomocą którego klasa – tym razem *pochodna* - decyduje sobie jak chce odziedziczyć nieprywatny składnik.

Podkreślam wyraźnie: **nieprywatny**. Co do prywatnego, to nic się nie da zrobić – jest on dziedziczony, ale zapieczętowany.

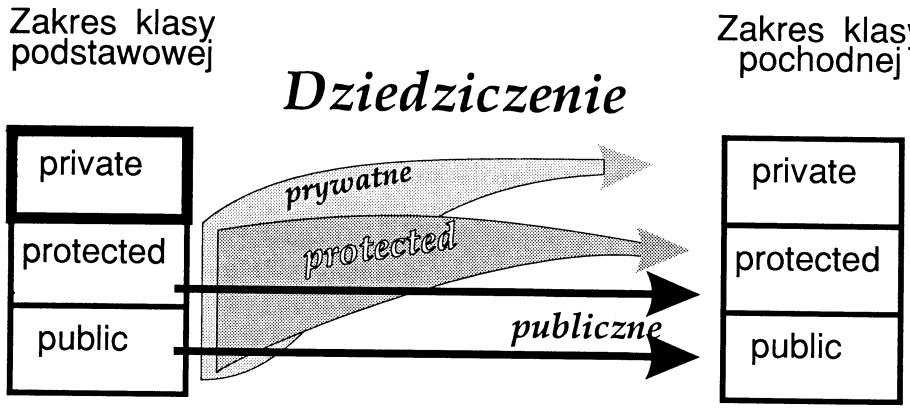
Sprawa dotyczy więc składników `public` i `protected`. Oba są jednakowo dostępne w zakresie klasy pochodnej. Klasa pochodna może sobie jednak wybrać czy na przykład chce, by odziedziczony składnik `public` był u niej także `public`. Może przecież zdecydować włożyć go u siebie do szufladki `private` i nikomu już go nie pokazywać.

Wyboru takiego dokonuje się przy definicji klasy pochodnej. Zauważ, że w pierwszej linijce definicji, na liście pochodzenia tuż przed nazwą klasy podstawowej `ryba` jest słowo `public`. Inną ewentualnością było by umieszczenie tam słowa `private` lub `protected`.

Jaką rolę spełniają te określenia?

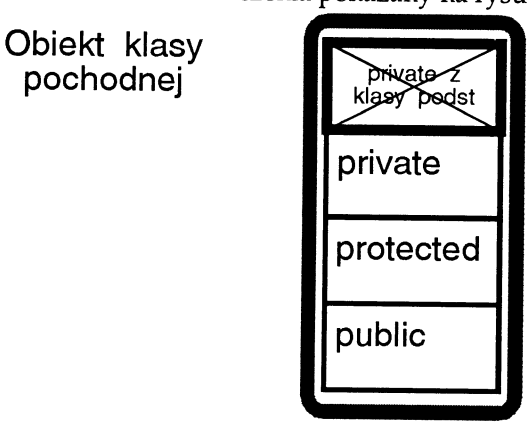
- Jeśli przy definiowaniu klasy pochodnej na liście pochodzenia poprzedzimy nazwę klasy podstawowej słowem `public` – wówczas składniki `public` z klasy podstawowej mają w klasie pochodnej również dostęp `public`, a odziedziczone składniki `protected` są w klasie pochodnej także `protected`.
- Jeśli na liście pochodzenia przed nazwą klasy podstawowej stoi określenie `protected`, wówczas składniki `public` oraz `protected` są w klasie pochodnej jako `protected`.
- Jeśli na liście pochodzenia przed nazwą klasy podstawowej stoi określenie `private`, wówczas składniki `public` oraz `protected` są w klasie pochodnej jako `private`.

Nie przejmuj się, nie trzeba tu niczego uczyć się na pamięć – wszystko staje się jasne, gdy spojrzymy na zamieszczony rysunek.



Zwracam jednak uwagę, że strzałki pokazują to, do którego obszaru dostępu klasy pochodnej następuje dziedziczenie, a nie to, czy dziedziczenie następuje w ogóle. Dziedziczone są wszystkie składniki. Brak strzałki przy obszarze `private` w klasie podstawowej oznacza, że zmiana rodzaju dostępu nie następuje. Odziedziczone elementy mają nadal dostęp `private`, ale w zakresie ważności klasy podstawowej. Ten zakres jest zasadniczo niedostępny z zakresu klasy pochodnej†).

Oto rezultat dziedziczenia pokazany na rysunku:



Ogólnie można powiedzieć, że:

☞ na dziedziczenie prywatne decydujemy się wtedy, gdy chcemy, by nie było publicznego dostępu do odziedziczonych składników klasy podstawowej. Nie jest to taki zły pomysł, szczególnie jeśli zależy nam na prawdziwej enkapsulacji (szczelnym zamknięciu w kapsułę). Dobrze napisana klasa nie powinna w zasadzie ujawniać niepotrzebnie swoich trybików i kółek. Nie tyle dla utrzymania tajemnicy, co dla prostoty późniejszej pracy z nią.

†) *„zasadniczo niedostępny"* – gdyż są sytuacje, kiedy dostęp jest możliwy. Na przykład wtedy, gdy klasa podstawowa deklaruje przyjaźń z klasą pochodną.

Dla wtajemniczonych:

> Dziedziczenie prywatne stosujemy wtedy, gdy nie chcemy by na obiekty klasy pochodnej kiedykolwiek mówiono, że są szczególnym rodzajem obiektów klasy podstawowej. Mimo, że tak jest w istocie - chcemy ten fakt zatajać.

Domniemania

Jeśli w definicji klasy pochodnej na liście pochodzenia przed nazwą klasy podstawowej nie umieścimy ani słowa `private`, ani słowa `public` ani `protected` – to przez domniemanie przyjęte zostanie dziedziczenie `private`.

Jeśli mamy do czynienia ze strukturą – to w analogicznym wypadku – przez domniemanie przyjmuje się tam dziedziczenie `public`.

```
class przodek { /*...*/ } ;

class klasa_dziedzic : przodek { /*...*/ } ;
struct struktura_dziedzic : przodek { /*...*/ } ;
```

W obu wypadkach zostały opuszczone słowa określające sposób dziedziczenia. Przez domniemanie kompilator przyjmie, że definicje tych klas pochodnych zapisaliśmy tak

```
class klasa_dziedzic : private przodek { /*...*/ };
struct struktura_dziedzic : public przodek {/*...*/};
```

19.2.4 Po znajomości, czyli udostępnianie wybiórcze

Wiemy już, że klasa pochodna może albo wszystkie odziedziczone składniki ukryć (dziedziczenie `private`, `protected`), albo pozostawić do nich taki dostęp, jaki miały oryginalnie.

Decyzja taka może czasem nie być łatwa. – No bo z jednej strony chcielibyśmy ukryć to i tamto, a z drugiej strony musi być dostęp `public` lub `protected` do kilku innych.

Co wtedy zrobić? Jest prosty sposób. Klasa pochodna dziedziczy klasę podstawową prywatnie (`private`). Natomiast te nazwy składników, które mimo wszystko dostają „przepustki" – należy w klasie pochodnej wyspecyfikować po etykiecie `protected` lub `public`. Powtarzam: same nazwy! Nie trzeba przypominać, że to jest nazwą funkcji, a tamto nazwą zmiennej, czy tablicy jakiegoś typu.

Zapis taki nazywa się deklaracją dostępu.

```
/////// klasa podstawowa ////////////////////////////////////
class przodek {
protected :
     int n ;
     float x ;
     void funprot(float*, int &) ;
public :
     int szer ;
     float * funpubl(int m) ;
};
```

```
//////// klasa pochodna ////////////////////////////////////
class potomek : private przodek {
     // ...
protected :
     przodek::x ; // <————————deklaracje dostępu
     przodek::funprot ;
public :
     przodek::szer ;
     przodek::funpubl ;
};
//////////////////////////////////////////////////////////////
```

Dzięki deklaracjom dostępu – mimo, że klasa `przodek` została odziedziczona prywatnie – wybrane nazwy mogą być udostępnione jako `private` lub `protected`.

Mankamentem tej techniki jest to, że nie możesz rozróżnić nazw przeładowanych - deklaracja dostępu wymienia przecież tylko nazwę (np. funkcji) nic nie mówiąc o argumentach.

Uwaga:

> Deklaracja dostępu może tylko powtórzyć dostęp, jaki nazwa miała w swojej klasie podstawowej. Czyli: jeśli tam nazwa była `protected` - to w klasie pochodnej może być tylko `protected`. Jeśli tam nazwa była `public` – to w klasie pochodnej może być tylko `public`. Krótko mówiąc: Deklaracją dostępu nie możemy ani zaostrzyć dostępu ani go rozluźnić.

Wybiórcze udostępnianie za pomocą deklaracji dostępu nie dotyczy składników prywatnych klasy podstawowej. Z tymi nic się nie da zrobić. **Dotyczy jedynie składników nieprywatnych, które zostały odziedziczone prywatnie.**

19.3 Czego się nie dziedziczy

Jeśli klasa ma wszystkie składniki nieprywatne i jeśli posłuży nam do zdefiniowania klasy pochodnej, to w świetle tego, co do tej pory powiedzieliśmy — wszystkie składniki zostaną odziedziczone do klasy pochodnej i będą mogły być używane tak, jakby były zdefiniowane w klasie pochodnej.

Teraz czas powiedzieć, że dla naszego dobra – nie wszystkie!

Nie zostaną w ten sposób odziedziczone

- 1) konstruktory,
- 2) operator przypisania (operator=),
- 3) destruktor.

19.3.1 Niedziedziczenie konstruktorów

Powód tego zrozumieć bardzo prosto. Obiekt klasy pochodnej to jakby obiekt klasy podstawowej, plus coś jeszcze. To „coś jeszcze", to składniki zdefiniowane w klasie pochodnej. O tych składnikach konstruktor klasy podstawowej nic nie wie. Posłużenie się takim konstruktorem (z klasy podstawowej) sprawiłoby, że

inicjalizowane byłyby tylko składniki-dane pochodzące z klasy podstawowej, a reszta zignorowana.

Na to zgodzić się nie możemy. Albo zrobić całość, albo nie robić nic ! Dlatego więc obowiązuje zasada, że konstruktory klasy podstawowej nie stają się automatycznie konstruktorami klasy pochodnej.

Konstruktory klasy pochodnej trzeba sobie po prostu zdefiniować. Nie jest to żmudne, bowiem do inicjalizacji części odziedziczonej można wywołać konstruktor z klasy podstawowej. O szczegółach powiemy niebawem.

19.3.2 Niedziedziczenie operatora przypisania

Te same względy sprawiają, że operator przypisania (`operator=`) klasy podstawowej nie staje się automatycznie operatorem przypisania klasy pochodnej.

Operator ten służy przecież do przypisania czegoś składnikom obiektu klasy podstawowej. Nie wie on nic o nowych składnikach, które zostały zdefiniowane w klasie pochodnej, więc ewentualne przypisanie byłoby niekompletne.

Tutaj trzeba uważać

Jeśli zapomnimy o tym niedziedziczeniu `operatora=` i mimo wszystko wykonamy przypisanie do obiektu klasy pochodnej (która sama operatora przypisania nie ma), to oczywiście `operator=` z klasy podstawowej nie zadziała. Za to zadziała operator generowany automatycznie przez kompilator. Będziemy więc mieli przypisanie „składnik po składniku", a to nie zawsze musi nam odpowiadać!

19.3.3 Niedziedziczenie destruktora

Jak wiemy, rolą destruktora jest ostateczne uporządkowanie na chwilę przed zlikwidowaniem obiektu. Destruktor często unieważnia to, co zrobił konstruktor. Skoro jest tak ściśle powiązany z konstruktorem - który musi być definiowany osobno dla klasy pochodnej – dlatego destruktor klasy podstawowej nie może być automatycznie użyty, jeśli nie zdefiniujemy nowego w klasie pochodnej.

Fakt niedziedziczenia konstruktorów i destruktora jest moim zdaniem łatwy do zapamiętania. Po prostu ich nazwy są związane z nazwą ich klasy (konstruktor nazywa się tak, jak jego klasa). Jeśli tworzymy klasę pochodną od tej klasy, to intuicyjnie jest oczywiste, że konstruktorowi klasy podstawowej nie zostanie zmieniona nazwa na nazwę właściwą konstruktorowi klasy pochodnej.

Natomiast ryzyko zapomnienia o niedziedziczeniu operatora przypisania jest większe.

19.4 Dziedziczenie kilkupokoleniowe

Dziedziczenie może być – że tak powiem – kilkupokoleniowe. To znaczy klasa pochodna może być równocześnie klasą podstawową dla innej klasy pochodnej.

Łatwo to sobie uzmysłowić za pomocą analogii w życiu rodzinnym: mój dziadek to jakaś klasa podstawowa. Z tej klasy wywodzi się mój ojciec - który jest klasą pochodną od klasy podstawowej „dziadek". Tymczasem ja wywodzę się od klasy „ojciec", – który jest dla mnie klasą podstawową.

Przy takich analogiach nie należy zapominać, że nie jest ona zupełna. W życiu bowiem dziadek, ojciec i ja, to konkretne obiekty, a nie klasy. Przypominam, że w C++ dziedziczenie dotyczy klas, a nie obiektów.

```
class A {
      // — ciało klasy A
} ;

class B : public A {
      // — ciało klasy B
} ;

class C :public B {
      // — ciało klasy C
}
```

Z klasy A wywodzi się klasa B, a z niej z kolei wywodzi się klasa C. Możemy to przedstawić w postaci grafu.

A
↑
B
↑
C

Strzałka na tym grafie oznacza jakby słowa: „pochodzi od".

Jak widać klasa C wywodzi się z klasy B bezpośrednio, ale także wywodzi się pośrednio od klasy A.

Mówimy, że dla klasy C klasa A jest klasą podstawową *pośrednio*. Tak jak mój ojciec jest dla mnie klasą podstawową bezpośrednią, a mój dziadek klasą podstawową pośrednią.

Bardziej precyzyjna definicja brzmi tak:

> Dla klasy K - klasa PRZODEK jest klasą podstawową bezpośrednią wtedy, gdy ta klasa PRZODEK znajduje się na liście klas podstawowych w definicji klasy K.
> W przeciwnym razie ta klasa podstawowa jest klasą podstawową pośrednią.

Przy tym kilkupokoleniowym (inaczej jakby „kaskadowym") dziedziczeniu widzimy znaczenie dziedziczenia prywatnego lub publicznego.

- Dziedziczenie publiczne powoduje, że składniki będące na samej górze kaskady (dziadek), są dostępne na samym dole (ja).
- Zastosowanie dziedziczenia prywatnego na którymś etapie powoduje, że w tym miejscu zamyka się następnym pokoleniom dostęp do odziedziczonych składników. (Od tej pory są jakby dziedziczone dalej w zalakowanej kopercie).

Czasem może być to korzystne – np. gdy definiujemy klasę biblioteczną będącą pochodną od wielu innych (roboczych) klas i zależy nam na tym, aby jak najmniej składników było dostępnych i widzianych z zewnątrz tej klasy bibliotecznej. Także z ewentualnych następnych pokoleń, bo ta klasa biblioteczna może przecież posłużyć użytkownikowi jako klasa podstawowa do jego własnej.

19.5 Dziedziczenie – doskonałe narzędzie programowania

Dziedziczenie jest jedną z najwspanialszych cech obiektowo orientowanych języków programowania.

Zastanówmy się do czego może nam się dziedziczenie przydać.

1) Oszczędność pracy

Dziedziczenie przyda się wtedy, gdy mamy już klasę, która jest podobna do takiej jaką potrzebujemy. Oszczędzamy sobie bardzo dużo pracy definiując tylko różnice, a nie wszystko od nowa.

Nie chodzi tu nawet o nasze lenistwo. Zauważ, że aby zdefiniować klasę pochodną nie musimy znać kodu źródłowego klasy podstawowej. Nie jest to ważne, jak klasa podstawowa robi to, czy tamto. Albo to akceptujemy i dziedziczymy na ślepo, albo nie akceptujemy i (mimo, że dziedziczymy także) zasłaniamy nowym składnikiem. Np. definiujemy nową funkcję składową, która zrobi dokładnie to, o co nam chodzi. Możemy też dopisać zupełnie nową funkcję składową, która wywoła jakąś funkcję odziedziczoną, aby wykonała część pracy, po czym sama coś poprawi lub uzupełni.

Zauważ, że dzięki dziedziczeniu upraszcza się nam sprawa. Nie musimy bowiem wiedzieć i rozumieć, jak klasa podstawowa realizuje jakieś funkcje. Jeśli to ktoś inny napisał tę klasę podstawową, to nie musimy ślęczeć nad wydrukami i myśleć: „w którym miejscu trzeba zrobić poprawkę, aby funkcja zmieniła się na taką, która nam odpowiada?"

Fakt ten, to jakby jeszcze bardziej wyszukany aspekt enkapsulacji: żeby dokonać zmian nie musimy dostawać się do wnętrza kapsuły (tutaj: klasy podstawowej). Aby to zrobić, potrzebujemy tylko definicję klasy podstawowej w wersji źródłowej — ta jest zwykle w specjalnym pliku nagłówkowym.

Natomiast definicje funkcji składowych tej klasy (czyli ich ciała) mogą być już nawet w skompilowanej wersji binarnej.

Ważny wniosek:

> Dzięki temu oddzielamy od siebie dwie sprawy: to, jak dana klasa jest zrealizowana, od tego, jak się nią posługiwać - nawet w celu dziedziczenia. Aby użyć klasy do tworzenia klas pochodnych, nie musimy wiedzieć dokładnie za pomocą jakich kruczków oblicza się w klasie to, czy tamto.

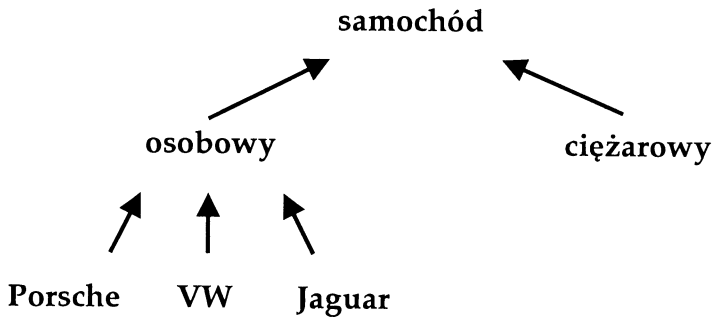
Ta zasada jest zbliżeniem programowania do życia codziennego.

2) Hierarchia

W poprzednim punkcie mówiliśmy o sytuacji, gdy mamy klasę, która przypomina to, co chcielibyśmy mieć. Dzięki dziedziczeniu możemy tanim kosztem mieć klasę, która jest tym, o co nam chodzi dokładnie.

Jest jednak coś o wiele ważniejszego. Bardzo często w programie mamy do czynienia z klasami, które są w jakiejś zależności logicznej.

Na przykład mamy klasę samochód, następne klasy to samochód ciężarowy, klasa samochód osobowy, klasa samochód osobowy marki Porsche. Każdą z tych klas można zdefiniować osobno, jednak od razu narzuca się, że samochód osobowy to szczególny przypadek samochodu, a samochód marki Jaguar to przecież rodzaj samochodu osobowego.
Powstaje wówczas coś w rodzaju hierarchii klas



Także i na tym grafie strzałkę czytamy jako: „wywodzi się od".
Hierarchia taka, a ogólniej: proces dziedziczenia pozwala na wprowadzenie relacji między poszczególnymi klasami. Zamiast mnożenia osobnych klas mamy teraz pewną logikę – jakaś klasa jest szczególnym rodzajem innej klasy bardziej ogólnej.

> Umiejętne skonstruowanie takiej hierarchii jest jedną z podstawowych cech dobrego obiektowo orientowanego programu.

Ustawienie klas w hierarchię dziedziczenia, to nie tylko oszczędność czasu przy definiowaniu klas znajdujących się na dole hierarchii. Ważniejsze jest to, że obiekty, które znajdują się na dole hierarchii, mogą być czasem traktowane tak, jakby były obiektami z góry hierarchii.

Przykładowo: Jeśli mamy funkcję pracującą dla obiektu klasy samochód, to można ją również użyć do pracy z obiektem klasy VW. W myśl tego, co powiedzieliśmy powyżej – VW może być także traktowany jako samochód.

Hierarchia wprowadza więc bardzo naturalne relacje między klasami.

3) Klasy ogólne

Możemy przy budowaniu klas nastawiać się na to, że jakaś klasa przeznaczona jest wyłącznie do dziedziczenia. Sama klasa nie oznacza nic sensownego, ale chcemy ją mieć po to, by mogła być dziedziczona przez inne klasy.

Taka klasa może np. służyć do pracy z jakimiś typami, które dopiero później zostaną zdefiniowane. Choćby w momencie dziedziczenia tej klasy do klasy pochodnej.

Przykładem takiej klasy ogólnej jest kolejka. W takiej kolejce mogą ustawić się ludzie (pamiętasz naszą klasę `osoba`?). W kolejce mogą też na przykład „stać" na przykład fragmenty ekranu, które trzeba sukcesywnie odświeżać.

Kolejka sama z siebie nie istnieje. Do tego potrzebne są obiekty (jakiejś klasy), które w takiej kolejce będą ustawiane.

Nie jest to tylko inny wariant sprawy omówionej w poprzednim punkcie. Tam obiekt klasy pochodnej był szczególnym przypadkiem klasy podstawowej. Tutaj tak nie jest. Człowiek stojący w kolejce nie jest szczególnym przypadkiem kolejki.

Trzeba pamiętać, że dziedziczenie jest tylko sposobem definiowania nowych klas – takim, które wprowadza między klasami pewne relacje.

Dziedziczenie nie daje obiektom klasy pochodnej żadnych większych praw wobec obiektów klasy podstawowej. Konkretniej: obiekt klasy pochodnej nie ma dostępu do niepublicznych składników obiektu klasy podstawowej. Jeśli chce się to umożliwić – klasa podstawowa powinna zadeklarować przyjaźń z klasą pochodną. Czyli tak, jak w wypadku jakiejkolwiek obcej klasy.

Klasa podstawowa i klasa pochodna mogą mieć – każda swoich – przyjaciół.

Aż się prosi, by teraz podać przykładowy program, w którym jest jakaś hierarchia. Zaczekajmy z tym jednak do poniższego paragrafu.

19.6 Kolejność wywoływania konstruktorów

W obrębie obiektu klasy pochodnej tkwi w pewnym sensie obiekt klasy podstawowej – tak, jak w obiekcie klasy samochód osobowy jest coś z samochodu.

Konstruowanie obiektu klasy pochodnej to skonstruowanie obiektu klasy podstawowej i dobudowanie tego, co jest właściwe klasie pochodnej. Pracują więc dwa konstruktory. Skoro tak, to może się nasunąć pytanie, który z konstruktorów wykona się najpierw. Czy konstruktor klasy podstawowej czy klasy pochodnej?

Odpowiedź jest intuicyjnie wyczuwalna: najpierw rusza do pracy konstruktor klasy podstawowej.

Może być też dodatkowo taka sytuacja, że składnikiem obiektu klasy pochodnej są obiekty innych klas (obiekty składowe). Obiektem składowym klasy samochód osobowy jest często wbudowane *radio*. Nie ma to nic wspólnego z dziedziczeniem. Radio nie jest przecież szczególnym rodzajem samochodu! Jeśli obiekt radio ma swój konstruktor, to kiedy on się wykona?

Odpowiedź na takie pytania można zawrzeć w jednym zdaniu:

> Klasa uszanuje najpierw starszych, potem swoich gości, a dopiero na samym końcu zajmie się sobą.

Ładnie to brzmi, ale co to znaczy? Starsi to przodkowie, czyli klasy podstawowe. Im kto starszy, tym bardziej godny szacunku, czyli klasa dziadek będzie miała pierwszeństwo przed klasą ojciec. Najpierw ruszy do pracy konstruktor klasy dziadek, potem kolejno konstruktor klasy ojciec. Gdy już cała starszyzna zostanie obsłużona ruszają do pracy konstruktory ewentualnych gości tej klasy – czyli konstruktory obiektów składowych goszczących w klasie pochodnej. Na samym końcu rusza do pracy konstruktor klasy pochodnej.

Nasi goście na końcu, ale jeśli dziadek miał swojego gościa to oczywiście ta sprawa zostanie załatwiona już przy konstrukcji dziadka.

Konstruktor klasy pochodnej

wygląda tak, jak zwykły konstruktor, z tym, że na liście inicjalizacyjnej można umieścić wywołanie konstruktora klasy podstawowej.

Przypominam, że lista inicjalizacyjna znajduje się przy **definicji** konstruktora

```
class A {
public:
     A() ;                          // konstr domniemany
     A(float) ;                     // inny konstruktor
     // ...
};
//////////////////////////////////////////////////////
class B : public A {
public:
     B() ;                          // deklaracja konstruktora
};
//////////////////////////////////////////////////////
// definicja konstruktora klasy B
B::B(int x) : A()          // <— lista inicjalizacyjna z wywołaniem
                           // konstruktora klasy podstawowej
{
     // ... ciało konstruktora
}
```

Jeśli klasa podstawowa nie ma żadnego konstruktora, wówczas oczywiście na liście inicjalizacyjnej nie umieszczamy niczego.

Natomiast jeśli klasa podstawowa ma konstruktor domniemany (czyli nie wymagający żadnych argumentów) i właśnie ten konstruktor chcielibyśmy w tym wypadku uruchomić, to możemy go nawet nie umieszczać na liście inicjalizacyjnej – bowiem w razie braku na liście inicjalizacyjnej jakiegokolwiek konstruk-

tora klasy podstawowej – automatycznie uruchamiany jest jej konstruktor domniemany.

Jeśli na liście inicjalizacyjnej nie umieściliśmy wywołania konstruktora klasy podstawowej, a klasa podstawowa w zestawie swoich konstruktorów nie ma konstruktora domniemanego – to wówczas kompilator uzna to za błąd.

W sumie zasada jest taka: na liście inicjalizacyjnej można pominąć wywołanie konstruktora bezpośredniej klasy podstawowej tylko wtedy, gdy:

- klasa podstawowa nie ma żadnego konstruktora.
- ma konstruktory, a wśród nich jest konstruktor domniemany.

Przykład

Zagadnienie kolejności uruchamiania konstruktorów klas podstawowych i składowych zilustrujmy przykładem. Mamy tu do czynienia z następującą hierarchią

Środek transportu
↑
samochód
↑
Mercedes

Dodatkowo w klasie samochód składnikiem jest obiekt klasy silnik
W klasie Mercedes składnikiem jest klimatyzacja. Graf ten dokładniej można narysować tak

Środek transportu
↑
samochód {silnik}
↑
Mercedes {klimatyzacja}

W naszym przykładzie pokażemy konstrukcję i destrukcję dwóch obiektów:

a) obiektu klasy samochód – który ma klasę podstawową środek transportu, oraz gości w swojej klasie obiekt klasy silnik,

b) obiektu klasy Mercedes, którego klasa wywodzi się bezpośrednio od klasy samochód (czyli pośrednio od klasy środek transportu). Obiekt klasy Mercedes gości w sobie składnik klasy klimatyzacja.

Konstruktory i destruktory będą „gadatliwe", co pozwoli nam prześledzić kolejność ich pracy

```
#include <iostream.h>
//////////////////////////////////////////////////////////
class silnik
{
protected :
     int  typ ;
```

```
public :
     silnik(int n) : typ(n)
     {    cout << "\tKonstruktor silnika "
                  "(skladnik samochodu)\n"; }

     ~silnik()
     {    cout << "\tDestruktor silnika "
                  "(skladnik samochodu)\n" ; }
} ;
//////////////////////////////////////////////////////////
class klimatyzacja
{
     int temperat ;
public:
     int nic ;
     klimatyzacja(float t) : temperat(t)
     {
      cout << "\t\tKonstruktor klimatyzacji "
                  "(skladnik mercedesa)\n" ;
     }

     ~klimatyzacja()
     {
      cout << "\t\tDestruktor klimatyzacji "
                  "(skladnik mercedesa)\n" ;
     }
};
//////////////////////////////////////////////////////////
class sr_transportu
{
protected:
     float poz_x ,        // bieżące współrzędne
           poz_y ;        // np. geograficzne
public:
     sr_transportu()
     { cout << " Konstruktor sr_transportu\n" ; }
     ~sr_transportu()
     { cout << " Destruktor sr_transportu\n" ; }
};
//////////////////////////////////////////////////////////
class samochod : public sr_transportu
{
protected :
     int aa ;
     silnik jego_silnik ;
public :
                                                    //❶
     samochod(int typ_silnika)
              : jego_silnik(typ_silnika), sr_transportu()
     {
          cout << "\tKonstruktor samochodu \n" ;
     }

     ~samochod()
     {
          cout << "\tDestruktor samochodu \n" ;
```

```
     }
} ;
//////////////////////////////////////////////////////////
class mercedes : public samochod {
protected :
     float xxx ;
     klimatyzacja casablanca ;
public :
     mercedes(float x, int typ_motoru, int klim) :
                    xxx(x) , samochod(typ_motoru),    // ❷
                    casablanca(klim)
     {    cout << "\t\tKonstruktor mercedesa\n" ; }

     ~mercedes()
     {    cout << "\t\tDestruktor mercedesa\n" ; }
} ;
/*********************************************************/
main()
{
     {
          cout << "Kreacja obiektu klasy samochod \n" ;

          samochod czarny(500) ;
          cout << "\nobiekt czarny samochod istnieje \n"
          "teraz bedzie likwidowany !\n\n" ;
     }
     cout << "obiekt samochod zlikwidowany\n\n" ;
     {
          cout << "Kreacja obiektu klasy mercedes \n" ;
          mercedes popielaty(6.5, 1200, 22.5) ;
          cout << "obiekt Mercedes istnieje \n"
          "teraz bedzie likwidowany !\n" ;
     }
     cout << "obiekt Mercedes zlikwidowany\n" ;
}
```

Na ekranie zobaczymy

```
Kreacja obiektu klasy samochod
 Konstruktor sr_transportu
     Konstruktor silnika (skladnik samochodu)
     Konstruktor samochodu

obiekt czarny samochod istnieje
teraz bedzie likwidowany !

     Destruktor samochodu
     Destruktor silnika (skladnik samochodu)
 Destruktor sr_transportu
obiekt samochod zlikwidowany

Kreacja obiektu klasy mercedes
 Konstruktor sr_transportu
     Konstruktor silnika (skladnik samochodu)
     Konstruktor samochodu
          Konstruktor klimatyzacji (skladnik mercedesa)
```

```
          Konstruktor mercedesa
obiekt Mercedes istnieje
teraz bedzie likwidowany !
          Destruktor mercedesa
          Destruktor klimatyzacji (skladnik mercedesa)
     Destruktor samochodu
     Destruktor silnika (skladnik samochodu)
 Destruktor sr_transportu
obiekt Mercedes zlikwidowany
```

Komentarz

Program dzieli się na dwie części, w których widzimy konstrukcję i destrukcję obiektów klas: `samochod` oraz `mercedes`.

Patrząc na ekran widzimy, że obiekt klasy samochód konstruowany jest tak, iż najpierw rusza do pracy konstruktor klasy podstawowej `sr_transportu` (przodek), po nim konstruktor obiektu będącego składnikiem klasy `samochod` czyli silnik (gość), a na końcu dopiero rusza do pracy konstruktor `samochodu`.

❶ Zauważ listę inicjalizacyjną przy konstruktorze klasy `samochód`. Widzimy tu wywołanie konstruktora klasy `silnik` (gość) i klasy `sr_transportu` (przodek). Nie ma znaczenia w jakiej kolejności umieścimy te konstruktory na liście. Zawsze

- –najpierw zostanie wywołany konstruktor klasy podstawowej,
- –następnie konstruktory gości (jeśli jest ich kilku, to w kolejności na liście).
- –następnie lista zostanie przeglądnięta raz jeszcze i zostaną wykonane wszystkie znajdujące się na niej inicjalizacje.

❷ Na liście inicjalizacyjnej konstruktora klasy `mercedes` jest inicjalizacja składnika `xxx`. Mimo, że jest pierwsza na liście wykona się ostatnia, bo ważniejszy jest konstruktor przodka `samochód` i składnika `klimatyzacja`.

Zauważ bardzo ważną rzecz. Na liście inicjalizacyjnej konstruktora klasy `mercedes` nie ma wywołania konstruktora klasy `sr_transportu`. To dlatego, że na liście inicjalizacyjnej umieszcza się tylko konstruktory klas podstawowych *bezpośrednich*.

Na tej liście nie można umieścić wywołania konstruktora klasy `sr_transportu`, gdyż konstrukcja i inicjalizacja składników pochodzących od środka transportu dokonała się już raz dzięki konstruktorowi klasy `samochód`. To on spowodował wywołanie tego konstruktora. Składniki te już więc istnieją i nie można próbować inicjalizować ich po raz drugi.

Destrukcja obiektu następuje w odwrotnej kolejności.

19.7 Przypisanie i inicjalizacja obiektów w warunkach dziedziczenia

Wspomnieliśmy, że operatora przypisania się nie dziedziczy. Podobnie nie dziedziczy się odpowiedzialnego za inicjalizację konstruktora kopiującego. (Nie dziedziczy się przecież żadnych konstruktorów).

Jak zatem zachowuje się klasa pochodna w wypadku przypisania lub inicjalizacji ?

Zanim odpowiemy na to pytanie przypomnijmy, że obiekt klasy pochodnej to obiekt klasy podstawowej, plus dodatkowe składniki, które zdefiniowaliśmy w klasie pochodnej.

Praca polegająca na przypisaniu (lub inicjalizacji) składa się więc jakby z dwóch części:

- – przypisanie (lub inicjalizacja) dla części odziedziczonej,
- – przypisanie (lub inicjalizacja) części nowej, właściwej tylko klasie pochodnej.

Jasne jest chyba, że dla przypisania lub inicjalizacji części odziedziczonej, wygodnie by było posłużyć się operatorem przypisania lub konstruktorem kopiującym z klasy podstawowej. Oczywiście jest to możliwe tylko wtedy, jeśli one istnieją i w dodatku są dostępne (czyli nieprywatne).

Wiemy, że klasa pochodna operatora przypisania i konstruktora nie dziedziczy. Klasa pochodna więc ich nie ma – chyba, że je dla klasy pochodnej zdefiniujemy.

Rozważmy wypadki:

a) gdy klasa pochodna nie definiuje swojego operatora przypisania,

b) gdy klasa pochodna nie definiuje swojego konstruktora kopiującego.

19.7.1 Klasa pochodna nie definiuje swojego operatora przypisania

Jeśli klasa pochodna nie definiuje swojego operatora przypisania, wówczas kompilator postara się o automatyczne wygenerowanie operatora przypisania

```
klasa & klasa::operator=( klasa & ) ;
```

pracującego według zasady: przypisanie „składnik po składniku".

Metoda „składnik po składniku"

Metoda ta oznacza, że w stosunku do składników, które są typów wbudowanych, zrobi się ich kopie. Jeśli chodzi o składniki typów zdefiniowanych przez użytkownika (klasy), to zostaną one przypisane za pomocą operatorów przypisania ze swoich klas. Także klasa podstawowa – jeśli ma ona operator przypisania, to dla przypisania tej części obiektu będącej dziedzictwem – użyty zostanie operator przypisania jej klasy.

Jeśli jednak klasa podstawowa ma operator przypisania, ale prywatny - oznacza to, że jej obiekty przypisuje się w jakiś specyficzny sposób, i klasa nie życzy sobie, by robił to ktokolwiek nieupoważniony. Kompilator wówczas zrozumie to i nie wygeneruje dla klasy pochodnej operatora przypisania.

Różnica między przypisaniem „bit po bicie" a „składnik po składniku"

Nie jest to, jak widać, ślepe przypisanie metodą „bit po bicie" – bo to zrobiłoby identyczną kopię. Przecież jeśli jakiś składnik zdecydował, że ma być przypisywany za pomocą swego specjalnego operatora – to kompilator generując operator przypisania klasy pochodnej powinien ten operator przodka/składnika użyć.

Zasada „składnik-po-składniku" to nie tylko bezmyślne skopiowanie jednego obiektu na drugi. W myśl tej zasady, jeśli składnik jest obiektem innej klasy (gość), a ta klasa ma dostępny (nieprywatny) operator przypisania - to zostanie on uruchomiony dla tego składnika.

Podobnie w wypadku części będącej dziedzictwem po klasie podstawowej. Zasada kopiowania „składnik po składniku" mówi, że dla tego dziedzictwa uruchomiony zostanie operator przypisania właściwy klasie podstawowej. Pod warunkiem, że istnieje i jest dostępny.

W dawniejszych wersjach języka C++ nie stosowano metody „składnik po składniku" – tylko metodę „bit po bicie".

`const` i referencje są przeszkodą

Gdyby klasa miała jakiś składnik `const` lub składnik będący referencją, to operator nie będzie wygenerowany. Po prostu dlatego, że do tych składników nie wolno nic przypisywać. Mogą być tylko inicjalizowane raz na zawsze. Ich obecność sugeruje, że nie jest to zwykła sytuacja.

Jeśli chcemy mimo wszystko mieć operator przypisania – musimy go sami zdefiniować. Jak to się robi? Zanim zilustruję to przykładem – porozmawiajmy o sytuacji gdy...

19.7.2 Klasa pochodna nie definiuje swojego konstruktora kopiującego

Jeśli klasa pochodna nie definiuje swojego konstruktora kopiującego - to kompilator postara się sam wygenerować dla tej klasy konstruktor kopiujący

```
klasa::klasa( klasa &) ;
```

- pracujący według zasady „składnik po składniku"

O tym, czym jest kopiowanie według zasady „składnik po składniku", mówiliśmy w poprzednim paragrafie. W myśl tej zasady – jeśli klasa podstawowa ma konstruktor kopiujący, to generowany automatycznie konstruktor klasy pochodnej – posłuży się nim.

Kiedy konstruktor kopiujący klasy pochodnej nie może być generowany automatycznie?

- Wtedy, gdy składniki tej klasy (goście) lub klasa podstawowa (przodek) mają jakiś konstruktor kopiujący, który jest niedostępny (prywatny) †).

Oznacza to, że z jednej strony w ich wypadku jest jakiś specjalny sposób konstruowania ich obiektów, a z drugiej – że nie życzą sobie by robił to ktokolwiek nieupoważniony.

Dla kompilatora jest to przesłanka, by nie generować automatycznie konstruktora kopiującego klasy pochodnej. Jeśli użytkownik chce go mieć, to niech zdefiniuje sobie go sam – podejmując przy tym decyzje co zrobić z tamtymi dziwnymi sytuacjami.

Jak się tym zająć – porozmawiamy niebawem.

19.7.3 Inicjalizacja i przypisywanie według obiektu wzorcowego będącego `const`

Aby w przypisaniu czy inicjalizacji móc posłużyć się obiektem wzorcowym, który jest typu `const`, należy mu zagwarantować nietykalność.

Omówione przed chwilą generowane automatycznie operator przypisania i konstruktor kopiujący

```
klasa &   klasa::operator=(klasa &);    // operator przypisania
          klasa::klasa(klasa &) ;       // konstruktor kopiujący
```

takiej gwarancji nie dają. Czyli posługując się nimi nie możemy po **prawej** stronie znaku = postawić obiektu z przydomkiem `const`.

Aby tak było, kompilator musiałby wygenerować takie funkcje

```
klasa &   klasa::operator=(const klasa &); //operator przypisania
          klasa::klasa(const klasa &) ;    //konstruktor kopiujący
```

Tak też zrobi, ale tylko wtedy, gdy spełniony jest drakoński warunek że:

> – wszystkie klasy podstawowe tej klasy oraz składniki tej klasy – mają takie operatory przypisania / konstruktory kopiujące, które przyjmują argumenty `const`.

Innymi słowy warunek jest spełniony wtedy, gdy wszyscy współpracownicy operatora przypisania / konstruktora kopiującego, zagwarantują argumentowi nietykalność.

19.7.4 Definiowanie konstruktora kopiującego i operatora przypisania dla klasy pochodnej

Poprzednio mówiliśmy o sytuacjach, kiedy konstruktor kopiujący i operator '=' generowane są automatycznie.

†) Jest prywatny, a przyjaźń z klasą pochodną nie jest zawarta.

Teraz porozmawiamy o tym, jak się samemu zająć ich definicjami. Ponieważ dawno już nie było przykładów – zacznijmy od tego. W przykładzie zauważysz przeładowanie operatora << . Nie ma to nic wspólnego z treścią tego paragrafu, ale skoro już znamy owo wspaniałe narzędzie, to praktykujmy posługiwanie się nim.

```
#include <iostream.h>
enum kolor { czarny, czerwony, niebieski, bialy } ;
//////////////////////////////////////////////////////////
class pojazd                                            // ❶
{
protected:
     int ilosc_kol ;
public:
     pojazd(int ile) : ilosc_kol(ile) {}
     pojazd(const pojazd &) ;
     pojazd & operator=(const pojazd & wz) ;
     // ...
};
//////////////////////////////////////////////////////////
class automobil : public pojazd                         // ❷
{
     int stan_licznika ;
     kolor kolor_karoserii;
public :
     automobil(int kola, kolor k , int licznik):pojazd(kola)
     {
          stan_licznika = licznik ;
          kolor_karoserii = k ;
     }
     automobil(const automobil &);
     automobil & operator=(const automobil &ww) ;
     friend ostream &
          operator<<(ostream & ekran,
                                   const automobil & obj) ;
} ;
/*********************************************************/
/* definicje konstruktorow kopiujacych dla obu klas      ***/
/*********************************************************/
pojazd::pojazd(const pojazd & wzor)
{
     ilosc_kol = wzor.ilosc_kol ;
}
/*********************************************************/
automobil::automobil(const automobil & wzorzec)
                                   : pojazd(wzorzec)
{                                                       // ❸
     stan_licznika = 0 ;
     kolor_karoserii = wzorzec.kolor_karoserii ;
}
/*********************************************************/
/*** definicje operatorow przypisania dla obu klas      ****/
/*********************************************************/
pojazd & pojazd::operator=(const pojazd & wz)
{
     ilosc_kol = wz.ilosc_kol ;
```

```
     return *this ;                                          // ❹
}
/*****************************************************/
automobil & automobil::operator=(const automobil &wzor)
{                                                            // ❺
     //————————————————————trzy sposoby zrobienia tego samego
     //————————- wywołanie jawne ————————————————
     (*this).pojazd::operator=(wzor) ;

     //————————- niejawnie: operator = z kasy podstawowej wywołujemy tak
     pojazd *wskpojazd = this ;
     (*wskpojazd) = wzor ;
     //————————albo tak (referencjami)————————————————
     pojazd & refpojazd = *this ;
     refpojazd = wzor ;

     // dalej tylko dokańczamy robotę
     kolor_karoserii = wzor.kolor_karoserii ;                // ❻
     stan_licznika = wzor.stan_licznika + 2;
     return *this ;
}
/*****************************************************/
/* Aby sie oswoic z przeladowywaniem operatora <<          ***/
/*****************************************************/
ostream & operator<<(ostream & ekran, const automobil & obj)
{
     ekran << " Liczba kol " << obj.ilosc_kol
          << ", Licznika " << obj.stan_licznika
          << ", kolor nr " << obj.kolor_karoserii <<endl;
     return ekran ;
}
/*****************************************************/
main()
{
automobil moj(4, bialy, 30000) ;
automobil twoj = moj ;                                       // ❼

     cout << "Lista istniejacych automobili \n"
          << "moj " << moj                                    // ❽
          << "twoj " << twoj ;

     automobil dziwak(3, bialy, 5000) ;
     cout << "Jeszcze jeden o nazwie dziwak :\n"
          << dziwak ;

     dziwak = moj ;                                          // ❾

     cout << "dziwak po przypisaniu :\n"<< dziwak ;

     const automobil muzealny(4, czerwony, 25000) ; // ❿
     dziwak = muzealny ;

     cout << "Muzealny : " << muzealny ;
     cout << "Dziwak po... " << dziwak ;
}
```

Po wykonaniu tego programu na ekranie pojawi się

```
Lista istniejacych automobili
moj Liczba kol 4, Licznika 30000, kolor nr 3
twoj Liczba kol 4, Licznika 0, kolor nr 3
Jeszcze jeden o nazwie dziwak :
 Liczba kol 3, Licznika 5000, kolor nr 3
dziwak po przypisaniu :
 Liczba kol 4, Licznika 30002, kolor nr 3
Muzealny : Liczba kol 4, Licznika 25000, kolor nr 1
Dziwak po... Liczba kol 4, Licznika 25002, kolor nr 1
```

Komentarz

❷ Jak widzimy z listy pochodzenia – klasa `automobil` jest pochodną od klasy `pojazd` ❶, którą dziedziczy w sposób publiczny.

Pytanie: czy klasa zamiast `automobil` mogłaby się nazywać krócej: `auto`?

Nie! Słowo `auto` jest słowem kluczowym w językach C i C++. Wiedziałem, że o tym zapomnisz, bo jest rzadko używane (patrz str. 98).

❸ Oto definicja konstruktora kopiującego klasy `automobil`. Jak widzimy, na liście inicjalizacyjnej jest wywołanie konstruktora kopiującego klasy podstawowej. Tym, czego konstruktor kopiujący klasy `pojazd` nie zainicjalizuje, czyli składnikami z części pochodnej – zajmujemy się w ciele konstruktora.

Jeszcze jedno: kiedy konstruuje się nowy samochód na wzór już istniejącego to, mimo wierności kopiowania, nie ma sensu nastawiać mu licznika przejechanych kilometrów na tę samą wartość, którą ma ten starszy. Decydujemy więc, że nowo narodzony `automobil` niech ma licznik = 0. Tak dla hecy, choćby po to, by nie była to bliźniacza kopia, którą mógłby nam przecież zrobić konstruktor generowany automatycznie.

Stawiając sprawę bardziej formalnie:

> Sytuacja taka zachodzi zwykle, gdy obiekt ma jakiś stan wewnętrzny, który świadczy, że już sobie trochę „pożył". Natomiast nowy, ten właśnie inicjalizowany obiekt powinien mieć stan wewnętrzny charakterystyczny dla nowo wykreowanego obiektu. Jeśli mamy taką sytuację, to jest to przesłanka do zdefiniowania tego konstruktora.

❹ Definicja operatora przypisania dla klasy podstawowej – nic ciekawego, znamy to. Przypominam, że zwyczajowo rezultatem takiej funkcji operatorowej jest referencja do obiektu, na którym operator pracował. Umożliwia to potem kaskadowe używanie tego operatora, czyli zapis:

```
a = b = c = d ;
```

❺ Definicja operatora przypisania klasy pochodnej `automobil`. Oczywiście nie ma tu mowy o jakiejś liście inicjalizacyjnej, bo to przecież nie jest konstruktor. Tu jest pewien problem: jak sobie uprościć życie tak, by tę część obiektu, która jest dziedzictwem od klasy pojazd – przypisać za pomocą operatora przypisania z klasy podstawowej?

Czyli: jak na rzecz obiektu klasy `automobil` wywołać operator przypisania z klasy `pojazd`? Ten nasz obiekt klasy `automobil` to oczywiście `(*this)`. Na jego rzecz mamy wywołać `operator=` z klasy `pojazd`. Oczywiście najłatwiej jest to zrobić stosując jawne wywołanie

```
(*this).pojazd::operator=(wzor) ;
```

> Czy wolno nam tak zrobić? Tak. Dzięki zasadzie, o której już niedawno wspominaliśmy. Mianowicie – obiekt klasy pochodnej może być w pewnych sytuacjach traktowany jak obiekt klasy podstawowej. `Automobil` to przecież także `pojazd`!

Funkcji, która spodziewa się nie obiektu, ale wskaźnika do klasy podstawowej lub referencji do klasy podstawowej – można wysłać wskaźnik lub referencję do klasy pochodnej.

To właśnie robimy tutaj: `wzor` jest referencją obiektu klasy `automobil`, a mimo to, ta referencja może zostać wysłana do funkcji spodziewającej się referencji do `pojazd`u.

Jeśli chodzi o wywołanie tej funkcji na rzecz obiektu `(*this)`, będącego przecież obiektem klasy `automobil` – to sprawa jest zupełnie prosta. Funkcja `pojazd::operator=` jest także składnikiem jego klasy `automobil` tyle, że jest zasłonięta przez funkcję operatorową `operator=` klasy pochodnej. Jeśli jednak posłużymy się operatorem zakresu, to zasłonięte nam się odsłania.

Jednak jawne wywołanie operatora nie wydaje mi się eleganckie. Spójrz zatem jeszcze raz do ciała operatora. Pokażemy tam **dwa inne sposoby zrobienia tego samego.** Funkcja operatorowa z klasy podstawowej ruszy do pracy wtedy, gdy po lewej stronie znaku przypisania zobaczy obiekt klasy podstawowej. Wobec tego trzeba ją oszukać. Jednym ze sposobów jest choćby zdefiniowanie sobie wskaźnika do obiektu klasy podstawowej i ustawienie go tak, by pokazał na obiekt klasy pochodnej.

```
pojazd *wskpojazd = this ;
```

Jest to najzupełniej legalna operacja. Wskaźnik klasy podstawowej może pokazywać na obiekt klasy pochodnej. Podobnie z referencją.

Nic w tym dziwnego. W końcu jak przyjeżdżasz *samochodem* do myjni samochodowej i mówisz „Proszę umyć ten mój *pojazd* " – to nie oszukujesz.

Zapis

```
(*wskpojazd) = wzor ;
```

jest właśnie wydaniem takiego polecenia. Jeśli chcemy wiedzieć jaki operator przypisania ruszy tutaj do pracy, to wystarczy zastanowić się co stoi po lewej stronie znaku =. Stoi tam obiekt pokazywany wskaźnikiem na *pojazdy*. Zatem ruszy operator dla klasy `pojazd`.

W następnych linijkach jest powtórzenie tego samego przy zastosowaniu chwytu z referencją do obiektu klasy `pojazd`. Wszystkie trzy pokazywane sposoby są równoprawne.

❻ Operator przypisania klasy `pojazd` przypisał tylko do tych składników, które były mu znane. Nic nie wie on przecież o składnikach dodanych w klasie

`automobil`. Dlatego po jego pracy musimy uzupełnić to, czego on nie mógł zrobić.

Tutaj też nie przypisuję na ślepo licznika – żeby nie była to bliźniacza kopia „składnik po składniku".

❼ Tu w `main` widzimy znak '='. Jaka funkcja tutaj ruszy do pracy?

Mówiliśmy o tym wcześniej, ale nie zaszkodzi jeszcze raz przypomnieć, że jedyną sytuacją, gdy na widok znaku '=' rusza konstruktor kopiujący jest linijka definicji obiektu. To właśnie ta sytuacja. Definiujemy obiekt, który ma być taki, jak inny obiekt pokazywany mu na wzór. Wszystkie dalsze znaki '=' w tym programie to już wywołanie operatora przypisania.

❽ Prawda, że opłacało się przeładować operator << wypisywania na ekran? Teraz zapis mamy tak prosty, jakbyśmy chcieli wypisać na ekran liczbę `int`.

Drobna uwaga

> *Ponieważ operator ten ma wypisywać na ekran składniki, które w klasie `automobil` są `protected`, dlatego musi dostać do tego od klasy `automobil` pozwolenie. Tym pozwoleniem jest deklaracja przyjaźni złożona w definicji klasy `automobil`. Składnik `ilosc_kol` z klasy podstawowej `pojazd` wchodzi do klasy pochodnej jako `protected` dlatego `operator<<` nie ma problemu z jego odczytaniem – wystarcza na to zgoda klasy `automobil`. Gdyby jednak ten składnik był w klasie `pojazd` prywatny, to ta zgoda by nie wystarczyła. Aby mógł być odczytany — w klasie `pojazd` musiałaby być deklaracja przyjaźni z naszą funkcją `operator<<` .*

❾ Oto wywołanie operatora przypisania.

❿ Definicja obiektu z przydomkiem `const`. Obiekt ten ma także służyć za wzór do przypisywania i konstruowania. Aby to było możliwe, musiałem w umieścić słowa `const` w różnych miejscach w programie. – Wszędzie tam, gdzie wysyłamy ten obiekt przez referencję do różnych funkcji. Występujące tam w deklaracjach argumentów formalnych słowa `const` są obietnicami tych funkcji, że nie będą nic zmieniać w obiekcie. Bez usłyszenia tych wszystkich obietnic, kompilator nie dopuściłby do użycia tego stałego obiektu wzorcowego w linijce inicjalizacji ani przypisania. (Mimo, że przecież stałby on po prawej stronie znaku =).

19.8 Dziedziczenie wielokrotne

Dotychczas zajmowaliśmy się dziedziczeniem jednokrotnym – czyli takim, gdzie klasa pochodna ma tylko jedną bezpośrednią klasę podstawową. Teraz dowiemy się, że:

> Klasa może wywodzić się bezpośrednio od więcej niż jednej klasy. Takie dziedziczenie nazywamy dziedziczeniem wielokrotnym.

Posługując się analogią rodzinną można powiedzieć, że klasa może mieć nie tylko ojca, ale także matkę. I „ojciec" i „matka" są dla niej bezpośrednimi klasami podstawowymi. Z nich **bezpośrednio** się wywodzi.

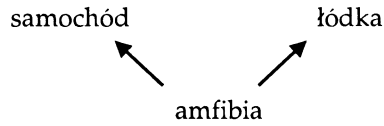
Dziedziczenie takie jest rzadziej używane niż dziedziczenie jednokrotne.

Zaletą dziedziczenia wielokrotnego jest to, że dzięki niemu możemy powiązać ze sobą niezależne od siebie typy klas. Przydaje się to na przykład wtedy, gdy klasa pochodna ma mieć dwoistą naturę

Oto przykład takiego dziedziczenia. Występują tu klasy podstawowe samochód i łódka. Zobacz co z nich powstaje:

```
class samochod {
     // ...
} ;
//-----------------------
class lodka {
     // ...
};
//-----------------------
class amfibia : public samochod, public lodka
{
     // ...
} ;
```

Na grafie wygląda to następująco:



Jak widać, dziedziczenia wielokrotnego dokonuje się przez umieszczenie na liście pochodzenia – więcej niż jednej klasy podstawowej. Lista może być dowolnie długa, klasa może wywodzić się od dowolnej ilości klas podstawowych.

Trzeba jednak pamiętać o kilku rzeczach

1) Dana klasa podstawowa może na liście pochodzenia pojawić się tylko raz. Zatem błędem byłoby powiedzenie, że amfibia pochodzi od samochodu, łódki i jeszcze raz od samochodu.

2) Definicja klasy umieszczonej na liście pochodzenia – musi być już znana kompilatorowi. Nie wystarcza deklaracja zapowiadająca (zwiastująca):

```
class samochod ;
```

Czyli nie wystarcza powiedzenie: „jakby co, to samochód jest nazwą klasy". Kompilator musi w tym momencie już znać samą definicję klasy podstawowej, a więc wiedzieć wszystko o typie jej składników. Ta wiedza jest mu niezbędna przy pracy nad właśnie definiowaną klasą pochodną.

3) Na liście pochodzenia przed nazwami klas podstawowych są określenia sposobu dziedziczenia (`private`, `protected`, `public`). Podobnie jak w znanym nam dziedziczeniu jednokrotnym, decydują one o tym, jaki dostęp będą miały odziedziczone nieprywatne składniki klas podstawowych.
Domniemanie jest identyczne jak przy dziedziczeniu jednokrotnym. Opuszcze-

nie tego słowa przed nazwą klasy podstawowej uznawane jest jako umieszczenie tam słowa `private`. W wypadku struktury (która jest rodzajem klasy) - przez domniemanie zakłada się dziedziczenie `public`.

Pamiętać należy, że na liście pochodzenia klasa podstawowa musi mieć **własne** określenie sposobu dziedziczenia. Nie możemy go opuścić sądząc, że obowiązuje to określenie, które napisaliśmy przed poprzednią klasą podstawową. Kompilator wówczas założy domniemany sposób dziedziczenia – czyli `private` (a dla struktur `public`).

Tak więc zapis

```
class amfibia : public samochod, lodka {
     // ...
} ;
```

Odpowiada zapisowi

```
class amfibia : public samochod, private lodka {
     // ...
} ;
```

Domniemane określenie jest `private`, ale nie radzę w tym wypadku korzystać z tego domniemania.

Praktyka wykazuje, że – nie wiadomo dlaczego – wszyscy programiści w tym wypadku sądzą, że domniemane jest określenie `public`. Najlepiej więc przyjmijmy zasadę, że zawsze piszemy to określenie sposobu dziedziczenia `private` lub `public`, a nie zdajemy się na domniemanie kompilatora.

19.8.1 Konstruktor klasy pochodnej przy wielokrotnym dziedziczeniu

W powyższym przykładzie dla prostoty nie mówiliśmy o konstruktorach.

Konstruktor klasy pochodnej przy wielokrotnym dziedziczeniu zawiera w liście inicjalizacyjnej ewentualne wywołania konstruktorów swych bezpośrednich klas podstawowych.

Wywołanie takie można opuścić jeśli chodzi nam o wywołanie konstruktora domniemanego (czyli takiego, który można wywołać bez żadnych argumentów). Oczywiście jeśli któraś z klas podstawowych nie ma żadnego konstruktora, to jasne jest, że na liście inicjalizacyjnej nie ma też odnośnego wywołania.

Na liście inicjalizacyjnej może się więc znaleźć kilka wywołań konstruktorów klas podstawowych. Po jednym dla każdej klasy podstawowej.

Ma się rozumieć, że przy konstruowaniu obiektu na podstawie tej listy inicjalizacyjnej nadal obowiązuje zasada, iż najpierw uszanowani są starsi, potem goście. Skoro starszych (bezpośrednie klasy podstawowe) jest kilkoro – ich konstruktory będą wywoływane w kolejności, w jakiej one występują na **liście pochodzenia klasy**. Potem dopiero wywołane zostaną konstruktory gości (składniki będące obiektami innych klas) - w kolejności, w jakiej występują na tej **liście inicjalizacyjnej konstruktora**.

Oto przykład klas łódka, samochód, amfibia uzupełnionych konstruktorami.

```
#include <iostream.h>
//////////////////////////////////////////////////////////
class samochod
{
protected :
     int a ;
public:
     samochod(int arg) : a(arg)
     {
          cout << "Konstruktor samochodu\n" ;
     } ;
} ;
//////////////////////////////////////////////////////////
class lodka {
protected :
     int b ;
public:
     lodka(int x) : b(x) {
          cout << "Konstruktor lodki \n" ;
     }
} ;
//////////////////////////////////////////////////////////
class amfibia : public samochod, public lodka        // ❶
{
public :
     amfibia() : samochod(1991) , lodka(4)           // ❷
     {
          cout << "Konstruktor amfibii \n" ;
     }

     void pisz_skladniki() {
          cout << "Oto odziedziczone skladniki\na = "
               << a << "\t b = " << b << endl ;
     }
} ;
/*********************************************************/
main()
{
amfibia aaa ;
     aaa.pisz_skladniki();
}
```

Na ekranie pojawi się

```
Konstruktor samochodu
Konstruktor lodki
Konstruktor amfibii
Oto odziedziczone skladniki
a = 1991   b = 4
```

Komentarz

Program nie robi jeszcze nic sensownego poza wypisaniem odziedziczonych składników. To przyda nam się w następnym paragrafie.

❶ Lista pochodzenia klasy `amfibia`.

❷ Teraz zwróć uwagę na listę inicjalizacyjną konstruktora klasy `amfibia`. Są na niej wywołania obu bezpośrednich klas podstawowych.

19.8.2 Ryzyko wieloznaczności przy dziedziczeniu

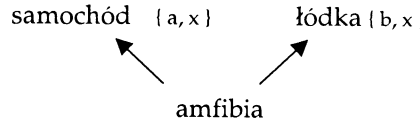
Przy wielokrotnym dziedziczeniu pojawia się problem wieloznaczności.

> Wieloznaczność to sytuacja, gdy wyrażenie odnoszące się do jakiegoś składnika klasy podstawowej, równie dobrze może odnosić się do innego składnika drugiej klasy podstawowej.

Wyobraź sobie, że w naszym poprzednim przykładzie w klasie samochód i w klasie łódka jest składnik o identycznej nazwie. Przykładowo – niech w klasie samochód będzie dodatkowy składnik

```
int x ;
```

oraz w klasie łódka będzie składnik o identycznej nazwie. Na grafie możemy to przedstawić następująco



Klasa amfibia dziedziczy te oba identycznie nazywające się składniki.

Czy jest to błąd? Jeszcze nie – nazwy tych składników mają przecież różny zakres ważności: zakres klasy samochód – względnie klasy łódka.

Pamiętać przecież należy, że mimo, iż do składników odziedziczonych odnosić się można tak, jakby były składnikami klasy pochodnej – są one jednak składnikami klas podstawowych, które są „zaszyte" w obiekcie klasy pochodnej.

Problem powstanie jednak, gdy w obrębie funkcji składowej klasy pochodnej `amfibia` spróbujemy odwołać się do nazwy `x` – kompilator uzna to za błąd.

Błąd polega na tym, że widząc nazwę `x` kompilator nie wie, czy chodzi nam o składnik odziedziczony z klasy samochód, czy z klasy łódka. Obie możliwości są jednakowo legalne.

Aby odniesienie się do takich składników było możliwe, posługujemy się wówczas operatorem zakresu z nazwą klasy, w której jest żądany składnik.

```
samochod::x
```

lub

```
lodka::x
```

Tak samo można odnieść się do innych składników – tych, które pojawiają się tylko jeden jedyny raz, np.

```
samochod::a
```

jednak nie jest to konieczne, albowiem wyrażenie `a` jednoznacznie określa, o który składnik chodzi – jest przecież tylko jeden składnik o nazwie `a`.

Co by było, gdyby w klasie samochód składnik o nazwie `x` był `private`, natomiast w klasie łódka składnik `x` był `public` lub `protected`? Mówiliśmy,

że składnik `private` jest dziedziczony jakby w zalakowanej kopercie – z zakresu klasy pochodnej nie ma do niego dostępu. Wobec tego w klasie pochodnej mielibyśmy odziedziczony jeden składnik o nazwie `x` – niedostępny (bo w kopercie), a drugi (ten z odziedziczony z klasy łódka) dostępny bez problemu.

Czy wobec tego nadal odniesienie się z klasy pochodnej `amfibia do` nazwy `x` byłoby błędne – bo wieloznaczne?

Pomyślałeś pewnie, że to już zupełnie inna sytuacja. A jednak nie. Byłby to mimo wszystko błąd. Wynika to z zasady, że:

> Najpierw sprawdzana jest jednoznaczność, a dopiero potem ewentualny dostęp.

Dlatego kompilator napotkawszy w klasie amfibia na odniesienie się do nazwy `x` zaprotestuje już na etapie sprawdzania jednoznaczności.

Operator zakresu to nie jest dobre rozwiązanie

Posłużenie się operatorem zakresu w celu uniknięcia wieloznaczności ma tę wadę, że ewentualne dalsze klasy pochodzące od klasy `amfibia` dziedziczą to ryzyko wieloznaczności. Czyli jeśli zdefiniujemy sobie pochodzącą od amfibii klasę amfibia_odrzutowa, to wewnątrz tej klasy odniesienie się do nazwy `x` jest nadal błędem.

Sprawa jest tym gorsza, że definiując kolejną klasę pochodną będącą jakby prawnukiem w tej hierarchii możemy już nie pamiętać dokładnie jakie kwalifikatory zakresu należy postawić przed nazwą. Może nam się nie chcieć wgłębiać w te historie rodzinne. Tak zresztą powinno być: używając klasy nie musimy znać całej jej genealogii. Powinna ona nam ułatwiać życie, a nie utrudniać.

Uwaga dla wtajemniczonych:

> *Dodatkową wadą jest to, że jeśli taka wieloznaczna nazwa jest nazwą funkcji wirtualnej, to poprzedzenie jej operatorem zakresu anuluje cały mechanizm wywołania funkcji wirtualnej (polimorfizm).*

Jest sposób na uniknięcie takich błędów wieloznaczności:

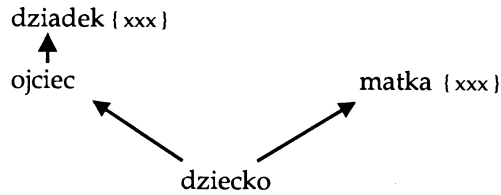
W klasie pochodnej zdefiniować można składnik o tej samej nazwie. Zasłoni on oba składniki z klas podstawowych. Tym pomocniczym – zasłaniającym - składnikiem niech będzie funkcja składowa klasy pochodnej. W jej wnętrzu odniesiemy się do tego składnika z klasy podstawowej, o który nam chodzi. To odniesienie się może być już z użyciem operatora zakresu

```
int amfibia::x()
{
     return samochod::x ;
}
```

Rozwiązanie to jest o tyle dobre, że teraz jeśli klasa `amfibia` ma dalsze klasy pochodne, to nie wystąpi już wieloznaczność. Następne pokolenia nie muszą już pamiętać o ryzyku wieloznaczności. Wieloznaczność została usunięta.

19.8.3 Bliższe pokrewieństwo usuwa wieloznaczność

Wiemy, że klasa może mieć dowolną liczbę bezpośrednich klas podstawowych, konieczne są zatem pewne zasady określenia wieloznaczności. Załóżmy, że mamy taką hierarchię klas:



Widzimy, że skoro i matka i dziadek mają składniki o nazwie `xxx`, to klasa `dziecko` dziedziczy dwukrotnie tę nazwę `xxx`. (Wszystko jedno czy to nazwa funkcji, czy danej).

Czy jest to dozwolone? Czy nie następuje wieloznaczność?

Do tej pory na pewno nie, bowiem wieloznaczność może wystąpić dopiero przy próbie odniesienia się do nazwy `xxx`.

Ale: czy poprawne jest w obrębie klasy `dziecko` **odwołanie się** do nazwy `xxx`?

> Tak, wieloznaczności nie ma. Droga do nazwy `xxx` w klasie podstawowej `matka` jest wyraźnie krótsza, niż droga do identycznej nazwy w klasie `dziadek`. To „bliższe pokrewieństwo" przesądza sprawę.

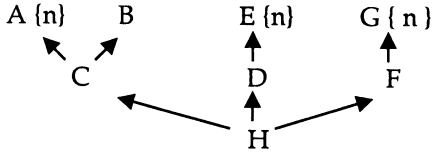
19.8.4 Poszlaki

Jeśli mimo wszystko musimy posłużyć się kwalifikatorem zakresu, by odnieść się do nazwy, to na szczęście nie musimy podawać dokładnego określenia, w której klasie dana nazwa się znajduje. Wystarczy jeśli przy operatorze będzie nazwa klasy podstawowej, od której zacząć poszukiwania.

W naszym ostatnim przykładzie, aby odnieść się do tego składnika o nazwie `x`, który został odziedziczony po dziadku, można też napisać

```
ojciec::x
```

Zobaczymy bardziej skomplikowaną hierarchię



Jeśli chodzi nam o odniesienie się do składnika `n` z klasy `A`, to możemy posłużyć się zapisem

```
A::n
```

ale także zapisem

```
C::n
```

To dlatego, że jeśli poszukiwania prowadzić się będzie w gałęzi `C`, to jednoznacznie znajdzie się składnik `n`. (Oczywiście ten `A::n`).

Podobnie z pozostałymi gałęziami

`D::n` to to samo co `E::n`
`F::n` to to samo co `G::n`

Kompilator zadowala się po prostu informacją gdzie szukać.

19.9 Pojedynek: Dziedziczenie klasy contra zwieranie obiektów składowych

Mówiliśmy już kiedyś, że konstruując klasę można jej składnikami uczynić obiekty innych klas. Klasa radio zawiera w sobie obiekty klasy tranzystor, klasa pokój zawiera obiekty krzesło, obiekt stół. Teraz poznaliśmy inny sposób użycia klas już wcześniej zdefiniowanych: dziedziczenie.

Zastanówmy się jaka jest między nimi różnica.

Kiedy zastosować którą technikę? Czy można te metody stosować wymiennie?

Aby odpowiedzieć na te pytania, wystarczy bardzo prosta regułka:

> Jeśli mówiąc o klasach obiektów używamy zwrotu „A składa się z B", wówczas mamy do czynienia z zawieraniem obiektów składowych.

> Jeśli używamy zwrotu „A jest rodzajem B", wówczas stosuje się dziedziczenie klas

Zatem jeśli mówimy: pokój **składa się** z 4 krzeseł i szafy – to jest teraz oczywiste, że następuje tu zawieranie obiektów składowych innych klas. W klasie pokój składnikiem jest obiekt klasy szafa i cztery obiekty klasy krzesła.

Jeśli mówimy: samochód **jest rodzajem** pojazdu (albo amfibia **jest rodzajem** samochodu) to jest od razu jasne, że powinno się zastosować technikę dziedziczenia.

Sposoby te nie są oczywiście wymienne. Zauważ, że o ile na amfibię można powiedzieć samochód (dziedziczenie), o tyle nie można na krzesło powiedzieć pokój ani na pokój – krzesło.

> W wypadku dziedziczenia jest, jak widać, jakaś szczególna relacja. Jedna klasa jest szczególnym rodzajem drugiej.

Z tej ostatniej właściwości będziemy w przyszłości wielokrotnie korzystali.

Odnoszenie się do składników w obu sytuacjach

Porównamy teraz sposoby odnoszenia się do składników w wypadku odziedziczenia ich, lub zawierania ich, jako części obiektów składowych. Jest to tylko

porównanie na zasadzie stwierdzenia faktów – nie jest to żaden argument na temat wyboru tej czy innej techniki.

W naszych rozważaniach zakładam, że dany składnik jest dostępny w klasie – to znaczy nie strzeże go w tej drugiej klasie słówko `private`.

- Przy dziedziczeniu odnoszenie się do odziedziczonego składnika jest tak samo proste, jakby był on składnikiem klasy pochodnej.

  Przykładowo: jeśli w amfibii chcemy coś powiedzieć o kierownicy - to mówimy po prostu: kierownica.

  ```
  kierownica
  ```

- Natomiast przy zawieraniu obiektów składowych innej klasy musimy przed nazwą składnika powiedzieć, do jakiego obiektu składowego się on zalicza. Ściślej: jakiego obiektu składowego jest on składnikiem.

  Przykładowo: W klasie pokój jeśli chcemy coś powiedzieć o drzwiach od szafy, to musimy dopowiedzieć te słowa „od szafy".

  ```
  szafa.drzwi
  ```

  Podobnie jeśli chcemy coś powiedzieć o oparciu od trzeciego krzesła, to również samo słowo „oparcie" nie wystarcza. Trzeba dodać, że od obiektu trzecie krzesło

  ```
  krzeslo[2].oparcie
  ```

Oto oba przykłady w skróconym zapisie. Najpierw dziedziczenie

```
class lodz ;
//////////////////////////////////////////////////////////
class samochod {
protected:
     int kierownica ;
} ;
//////////////////////////////////////////////////////////
class amfibia :public lodz, public samochod
{
     // ...
     void fun()
     {
          cout << kierownica ;          // <— tutaj !
     }
};
//////////////////////////////////////////////////////////
```

Teraz zawieranie obiektów innych klas

```
class szafa {
public:
     int drzwi ;
} ;
//////////////////////////////////////////////////////////
class pokoj {
     szafa gdanska ;
     void pisz()
```

```
        {
            cout << gdanska.drzwi ;        // <— tutaj !
        }
    } ;
```

19.10 Konwersje standardowe przy dziedziczeniu

Mówiliśmy już kilkakrotnie, że jeśli jedna klasa pochodzi od drugiej, to można uznać, że obiekt klasy pochodnej ma w sobie coś, co sprawia, iż można go uznać także za obiekt klasy podstawowej.

Przykładowo – jeśli patrzymy na najróżniejsze modele samochodów, to w każdym z nich jest coś, co sprawia, że można o nich powiedzieć - to jest pojazd!

Po to, aby można było się także i w C++ posłużyć takimi zależnościami — predefiniowane są następujące konwersje standardowe:

> Wskaźnik do obiektu klasy pochodnej może być niejawnie przekształcony na wskaźnik dostępnej jednoznacznie klasy podstawowej.

Podobnie z referencjami (czyli przezwiskami)

> Referencja obiektu klasy pochodnej może być niejawnie przekształcona na referencję jednoznacznie dostępnej klasy podstawowej.

Obie powyższe sytuacje są możliwe, gdy klasa podstawowa jest dziedziczona publicznie i taka konwersja jest jednoznaczna.

Wszystko to brzmi trochę skomplikowanie, więc zatrzymajmy się na tym co to znaczy

Aby to zilustrować, wyobraźmy sobie taki obrazek. Przed nami leżą dwa wskaźniki. Na jednym jest napisane: „do pokazywania na samochody", a na drugim: „do pokazywania na pojazdy".

Pytanie: jeśli na jakiś obiekt koło nas pokazałem wskaźnikiem „tylko do pokazywania na samochody" – i było to poprawne – to czy mogę na ten sam obiekt pokazać „wskaźnikiem do pokazywania na pojazdy"?

Mogę – samochód to także pojazd. To zezwolenie jest właśnie treścią naszej reguły mówiącej, że: wskaźnik do obiektu klasy pochodnej (samochód) może być niejawnie przekształcony na wskaźnik dostępnej jednoznacznie klasy podstawowej (pojazd).

Odwrotna zasada nie jest prawdziwa. Na wóz drabiniasty możemy pokazać „wskaźnikiem do pojazdu", ale nie możemy pokazać na niego „wskaźnikiem do samochodu"

Z definicją drugą, dotyczącą referencji jest podobnie. Jeśli na jakiś obiekt klasy Fiat 126, mówię przezwiskiem „maluch", to mogę również powiedzieć przezwiskiem „wóz" – które jest przezwiskiem pojazdu.

Ale nie odwrotnie: jeśli ktoś powiedział o swoim samochodzie przezwiskiem „wóz", to wcale nie oznacza, że ma coś o przezwisku „maluch".

Zasady, o których mówimy są tak sformułowane, by odpowiadały naszym doświadczeniom ze świata realnego. Dlatego są tak proste.

Pomyślałeś pewnie, że wszystko to są rozważania czysto akademickie. Nie jest tak. To, o czym mówimy teraz ma ogromne znaczenie w praktyce.

Oznacza to mianowicie, że jeśli mamy funkcję, która przyjmuje referencję obiektu klasy podstawowej, to można ją wywołać także dla obiektu klasy pochodnej. Oto taka sytuacja:

```
class samochod {                                          // ❶
public :
     int zbiornik ;
     // ..ew . coś jeszcze
};
/////////////////////////////////////////////////////////////
class VW : public samochod {                              // ❷
     // ...
} ;
/*********************************************************/
//    zwykła funkcja globalna
/*********************************************************/
void stacja_benzynowa(samochod & klient)                  // ❸
{
     klient.zbiornik = 50 ;
}
/*********************************************************/
main()                                                    // ❹
{
     samochod prawdziwy_samochod ;
     stacja_benzynowa(prawdziwy_samochod) ;               // ❺
     VW    golf_huberta ;                                 // ❻
     stacja_benzynowa(golf_huberta) ;                     // ❼
     // ...
}
```

Komentarz

Mamy, jak widać, dwie klasy.

❶ Klasa `samochod` ma składnik `zbiornik`, który symbolizuje zbiornik paliwa. Oczywiście mogą być w klasie jeszcze jakieś inne składniki, to jednak jest dla nas teraz nieistotne.

❷ Klasa `VW` (jak: Volkswagen) jest klasą pochodną od klasy samochód – a więc dziedziczy `zbiornik` paliwa.

❸ Widzimy też funkcję `stacja_benzynowa`. Jest to zwykła funkcja globalna, której argumentem formalnym jest referencja obiektu klasy samochód.

❹ W `main` widzimy definicję obiektu klasy `samochód` – obiekt ten nazywamy `prawdziwy_samochod`.

❺ Funkcja `stacja_benzynowa` zostaje wywołana dla tego argumentu. Odbiera go przez referencję i dokonuje tankowania. Spodziewała się argumentu `samochod` i go dostała. Jak dotąd – nie ma tu nic nadzwyczajnego.

❻ A teraz uwaga: mamy klasę `VW`, która jest pochodną klasy `samochod`, definiujemy obiekt tej klasy. Nazywamy go `golf_huberta`.

❼ Dzięki tej zasadzie, o której mówimy w tym paragrafie – funkcja `stacja_benzynowa` również przyjmie referencję do klasy `VW`. Nastąpi wtedy niejawna konwersja standardowa tej treści: referencja do obiektu klasy pochodnej `VW` zostanie zamieniona na referencję do obiektu klasy `samochod`. Czyli tak, jakbyśmy napisali

```
stacja_benzynowa( (samochod &) golf_huberta) ;
```

W życiu codziennym taka konwersja jest bardzo częsta. Mimo, że Hubert ma Volkswagena może go zaparkować na parkingu samochodowym – a nie jedynie na parkingu „dla VW"

> Programowanie obiektowo orientowane w języku C++ dąży do tego, by możliwie blisko odwzorować zachowania i sytuacje ze świata realnego. To bowiem ułatwia programowanie. Dlatego też C++ został wyposażony w te konwersje.

To samo dla wskaźnika

Pokazaliśmy jak odbywa się konwersja referencji – teraz pokażemy jak odbywa się konwersja wskaźnika do obiektów klasy pochodnej na wskaźnik do obiektów klasy podstawowej.

Mamy inną funkcję globalną

```
void spalanie(samochod *wskaz_sam)
{
     wskaz_sam->zbiornik -= 3 ;          //spalenie 3 litrów
}
```

Argumentem formalnym tej funkcji jest wskaźnik do obiektu klasy `samochod`. Znaczy to, że do funkcji tej można przysłać argument będący adresem obiektu klasy `samochod`. Na mocy zasady, którą poznaliśmy w tym paragrafie - można też tej funkcji wysłać adres obiektu klasy pochodnej od klasy `samochód`. Klasa `VW` jest taką klasą, zatem poprawne są oba poniższe wywołania

```
spalanie(&prawdziwy_samochod) ;
spalanie(& golf_huberta) ;
```

Dlaczego tylko przy dziedziczeniu publicznym?

Wypada teraz wytłumaczyć się z faktu dlaczego taka konwersja może nastąpić tylko wtedy, gdy klasa podstawowa została odziedziczona publicznie, czyli gdy pierwsza linijka definicji klasy `VW` wygląda tak

```
class VW : public samochod
{
     // ...
}
```

Gdyby na tej liście pochodzenia było słowo `private`, (albo nic nie było, co – przez domniemanie – uznane jest za dziedziczenie `private`), wówczas wspomniana konwersja nie może zajść automatycznie.

Czy intuicyjnie „czujesz" dlaczego? Jeśli nie, to przywołajmy sobie taki obrazek, gdzie dziedziczenie jest `private`.

Otóż w Deutsche Oper na przedstawieniu Wolnego Strzelca (K. M. Webera) jest akt rozgrywający się nocą w mrocznej puszczy. Na olbrzymiej scenie widzimy wielką romantyczną dekorację: ogromne zwalone drzewa, góry, strumienie – wszystko w niesamowitej atmosferze czarów. Słowem: przedsionek piekła. Scena ta musi być ustawiona w miarę sprawnie, więc większość elementów dekoracji ma kółka i wjeżdża w czasie przerwy na scenę. Jest tam też ogromna góra – wielka i ciężka. Ona też ma ukryte kółka, a dodatkowo silnik i układ kierowniczy. Operator sceny w odpowiednim momencie siada za kierownicą i sprawnie wjeżdża tym fragmentem dekoracji na scenę.

Czy silnik jest elektryczny czy spalinowy – nie wiem – dla naszego przykładu załóżmy, że jest spalinowy. Skoro ta góra ma koła, silnik i kierownicę, to znaczy jest rodzajem pojazdu samochodowego. „Jest rodzajem" – czyli: dziedziczy go.

Z drugiej jednak strony, nikt z widzów nie powinien tego zauważyć. Dlatego wszystko jest ukryte – czyli dziedziczenie jest prywatne. Jeśli nawet któryś z widzów to przeczuwa – nie powinien potrafić wskazać na te składniki ukrytego samochodu.

To by było dziedziczenie prywatne. Ciągnijmy dalej tę analogię. Pewnego razu wyjeżdżamy tą górą z teatru i zajeżdżamy do pobliskiej stacji benzynowej, po to by nabrać paliwa do zbiornika. Kierownik stacji widzi, że przed nim stoi góra. Czy obsłuży taką górę?

Nie! Każe nam się puknąć palcem w czoło i zadzwoni po policję (sygnalizacja błędu w czasie kompilacji).

Góra nie zostanie więc potraktowana przez funkcję `stacja_benzynowa` jako potomek samochodu, bo ten „samochód" jest w tej górze sprytnie ukryty.

Więc co? Nic się nie da zrobić? Nie można wytłumaczyć kierownikowi stacji tajemnicy tej góry? Można, ale to już będzie *jawna* konwersja, podczas gdy my tutaj mówimy o niejawnych, czyli mogących zajść automatycznie. Hubert mógł swoim Golfem zajechać do stacji benzynowej i bez tłumaczenia czegokolwiek zostać obsłużonym. Góra z ukrytym (`private`) samochodem w środku – tego nie może.

Przełóżmy to na język C++. Oto klasa:

```
class operowa_gora : private samochod {
    // ...
};
```

Jak widać, dziedziczenie jest prywatne. Jeśli teraz w programie wystąpi instrukcja

```
operowa_gora gora_akt2 ;
    // ...
    stacja_benzynowa(gora_akt2);          // !!!
```

To kompilator uzna to za błąd. Konwersja niejawna nie może tutaj zajść, jako że dziedziczenie było prywatne. Jeśli jednak zastosujemy jawną konwersję

```
stacja_benzynowa( (samochod &) gora_akt2);
```

To kompilator już nie zaprotestuje. Uzna, że skoro piszemy to jawnie, to pewnie wiemy co robimy.

19.10.1 Panorama korzyści

Konwersje standardowe, o których mówiliśmy ostatnio, pozwalają nam korzystać z faktu, iż klasa pochodna jest w logicznej relacji w stosunku do klasy podstawowej. To jest jeden z najważniejszych aspektów dziedziczenia. Dziedziczy się nie tylko z lenistwa, aby sobie oszczędzić pracy przy nowej klasie — dziedziczy się głównie po to, by operować klasami ustawionymi w pewien układ zależności czyli hierarchię.

Do tej pory, poza momentem definiowania klasy pochodnej- nie można było w naszych programach zauważyć korzyści z takiej hierarchii. Teraz, kiedy mamy do dyspozycji wspomniane konwersje standardowe – możemy wreszcie w pełni korzystać z dobrodziejstwa, jakie nam to ustawienie klas w hierarchie daje.

Kiedy zachodzą takie sytuacje

Konwersje standardowe wskaźnika (wzgl. referencji) do obiektu klasy pochodnej na wskaźnik (referencję) obiektu publicznej klasy podstawowej mogą zajść w sytuacjach typowych dla innych konwersji standardowych, czyli:

- a) przy przysłaniu argumentów do funkcji (jako referencję lub wskaźnik),
- b) przy zwracaniu przez funkcję rezultatu będącego referencją lub wskaźnikiem,
- c) przy przeładowanych operatorach,
- d) wyrażeniach inicjalizujących.

Wyjaśnijmy po kolei te przypadki

ad a) – To już znamy z niedawnego przykładu ze stacją benzynową: **funkcję, która spodziewała się argumentu będącego referencją** do klasy podstawowej, można było wywołać z argumentem będącym obiektem (lub referencją do obiektu) klasy pochodnej. Podobnie ze wskaźnikami.

ad b) To samo zachodzi **w przypadku rezultatu zwracanego przez funkcję.** W funkcji, która zwraca referencję (wzgl. wskaźnik) do obiektu jakiejś klasy podstawowej, można koło słowa `return` postawić referencję (wskaźnik) obiektu klasy pochodnej.

ad c) Jeśli mamy **operator, który pracuje na referencji obiektu klasy podstawowej** - czyli innymi słowy jest to operator, który spodziewa się, że po którejś z jego stron stać będzie obiekt (lub jego referencja) jakiejś klasy podstawowej, to

możemy tam postawić obiekt klasy pochodnej od niej. Operator potraktuje ten obiekt tak, jakby był on obiektem klasy podstawowej. Nic w tym specjalnie nowego. Przecież naszą funkcję `stacja_benzynowa` mogliśmy zrealizować w postaci operatora.

Ważne jest tylko to, że konwersja wystąpi tylko w stosunku do tych argumentów, które są wysyłane do funkcji i są w nawiasie jako argumenty formalne wywołania funkcji.

Warto pamiętać, że jeśli funkcja operatorowa jest funkcją składową jakiejś klasy, to wspomniana konwersja standardowa nie nastąpi w wypadku pierwszego ukrytego argumentu – czyli obiektu na rzecz, którego wywołuje się operator. Jeśli więc nam na tym zależy, to operator trzeba zdefiniować jako funkcję nieskładową (globalną). Wspominaliśmy już o tym przy omawianiu przeładowania operatorów (str. 485).

ad d) **Przy wyrażeniach inicjalizujących** - może nastąpić taka sytuacja, że chcemy zdefiniować obiekt klasy `pojazd` i równocześnie go zainicjalizować. Rozglądamy się po programie i znajdujemy, że najlepiej jako wzorzec do inicjalizacji nadaje się pewien konkretny obiekt klasy `samochód`. Jako obiekt klasy pochodnej ma on co prawda więcej składników-danych, ale teraz jest to dla nas nieistotne. Interesuje nas jego trzon – czyli to, co jest w nim z pojazdu. To tym właśnie zainicjalizować chcemy obiekt klasy `pojazd`. Konwersja standardowa sprawia, że jest to możliwe.

```
samochod s ;
     // ...
pojazd p = s ;        // niejawna konwersja
```

Przypominam po raz milionowy, że znak = w linijce inicjalizacji jest jedynym przypadkiem, gdy do pracy rusza nie `operator=` przypisania, tylko konstruktor kopiujący klasy `pojazd`.

```
pojazd::pojazd(pojazd &)
```

W nawiasie widzisz, że argument jest referencją obiektu klasy podstawowej `pojazd`. My natomiast z prawej strony znaku = postawiliśmy obiekt klasy pochodnej `samochod`.

Pewnie zawołałeś teraz: „–Przecież to już omówiliśmy – toż to przypadek a) !" Brawo, masz rację.

Oczywiście odwrotna inicjalizacja nie jest możliwa. To znaczy nie można zbudować samochodu, mając za wzorzec obiekt klasy pojazd

```
pojazd p ;
     // ...
     samochod s = p ;              // błąd !!!
```

Dygresja:

Gdyby nam bardzo zależało, by ta ostatnia instrukcja była mimo wszystko możliwa, to powinniśmy sobie zdefiniować operator konwersji z typu `pojazd` na typ `samochod`. Mówiliśmy już o tych sprawach i jak wiesz jest to bardzo łatwe. Z tym, że nie są już to konwersje standardowe, a

konwersje definiowane przez użytkownka. Tu mówimy tylko o standardowych.

Ogólny wniosek jest taki: dzięki istnieniu tych konwersji standardowych (tyczących klas pochodnych) pewne fragmenty programu mogą być bardziej uniwersalne. Możemy używać tych fragmentów kodu, które były przeznaczone dla kogo innego – owo wielokrotne używanie to jeden z aspektów *reusability*

Uwaga dla programistów C

Mam nadzieję, że zauważyłeś już doniosłość tego faktu. W klasycznym języku C funkcja mogła być wywołana z argumentem będącym wskaźnikiem do jakiegoś obiektu.

```
void narysuj( struct plansza *wskaz) ;
```

Jeśli nawet za chwilę zbudowaliśmy bardzo podobną strukturę opisującą np. menu – to funkcji tej nie dało się już użyć do tego celu. Trzeba było po raz drugi napisać podobną funkcję:

```
void narysuj_menu( struct menu *wskaz) ;
```

Natomiast w C++, jeśli definiując strukturę `menu` powiemy, że jest ona pochodną od struktury `plansza`, to funkcja `narysuj` może pracować także dla obiektów struktury `menu`. Oszczędzamy pracy, oszczędzamy kodu – program robi się mniejszy !

19.10.2 Czego robić nie można

Zachwycając się hierarchią zapewniałem, że dziedziczenie pozwala nam traktować obiekty klas pochodnych jako szczególny rodzaj obiektów klasy podstawowej. Innymi słowy, samochód to szczególny rodzaj pojazdu.

> Zauważ jednak, że mówiąc o konwersjach standardowych zawsze podkreślałem, że konwersje te mogą zajść dla wskaźników pokazujących na takie obiekty lub dla referencji takich obiektów. Ani słowem nie powiedziałem, że to sam obiekt klasy pochodnej może być zamieniony na obiekt klasy podstawowej. Byłoby to fatalne w skutkach.

Wyobraźmy sobie taką sytuację. Funkcja spodziewa się, że zostanie przysłany do niej przez wartość obiekt klasy pojazd:

```
void fun(pojazd, int) ;
```

Przysłanie obiektu jest przez wartość (a nie przez referencję – nie ma tam przecież znaczka &). Funkcja „wie", że przysłany do niej obiekt ma rozmiary np. 33 bajty. Jako drugi argument przyjdzie przysłana przez wartość liczba `int`.

Wszystko działa świetnie, ale w pewnym momencie postanawiamy do funkcji wysłać (oczywiście przez wartość) obiekt klasy pochodnej `samochod`. Kompilator zaprotestuje. Dlaczego? W końcu samochód to także pojazd !

Oto dlaczego: `samochod` to obiekt klasy pochodnej i ma zwykle więcej składników niż obiekt klasy podstawowej. Niech ma u nas np. 70 bajtów. Funkcja wywołana dla takiego dużego obiektu zostaje niemal zasypana tymi dodatkowymi bajtami. Pracując dotychczas wiedziała, że na stosie czeka ją prezent wielkości 33 bajtów. W następnych bajtach jest już oczekiwana przez nią liczba `int` – drugi argument.

Tymczasem bez wiedzy funkcji na stos został rzucony obiekt `samochód` o wielkości np. 70 bajtów. Funkcja odbiera pierwsze 33 bajty i z nich stara się zrobić obiekt klasy `pojazd`. Następnie sięga po 34 bajt – bo tam powinna zacząć się liczba `int` – drugi argument. Tam jednak go nie ma, bo są tam są resztki obiektu klasy samochód!

Obrazek był jednak czysto hipotetyczny. Kompilator bowiem nigdy nie dopuści do podobnej sytuacji – od razu zaprotestuje. Zapamiętaj :

> Obiekty klas pochodnych mogą być traktowane jako obiekty swych klas podstawowych – wtedy tylko, gdy pracujemy na ich adresach. Wskaźnik zawiera zapisany adres ; adresem swego rodzaju jest też referencja.

Zatem w naszym wypadku opisana sytuacja możliwa jest tylko wtedy, gdy funkcja wygląda tak:

```
void fun(pojazd & przezw, int i);
```

albo[†)]

```
void fun(pojazd * wskazn, int i);
```

Jeśli funkcja spodziewa się adresu klasy podstawowej, a prześlemy jej adres klasy pochodnej – problemu nie ma – jeden i drugi adres mają taki sam rozmiar. Adres ten zostaje zdjęty ze stosu, a drugi argument (ten `int`) może być bezbłędnie odszukany.

Podsumowanie

Do funkcji spodziewającej się (przysłanego przez wartość) *obiektu klasy podstawowej* **nie da się wysłać** *obiektu klasy pochodnej*. Kompilator do tego nie dopuści.[††)]

---

†) Przypominam, że referencja i wskaźnik nie są rozróżniane przy przeładowaniu. Zatem: albo-albo! Takie funkcje nie mogą się pojawić jednocześnie w tym samym zakresie.

††) Chyba, że jest zdefiniowana przez użytkownika konwersja zamieniająca obiekt klasy pochodnej na obiekt klasy podstawowej. Wówczas jednak naprawdę do funkcji zostanie wysłany chwilowy obiekt klasy podstawowej. Ten obiekt chwilowy zrobiony zostanie naszym operatorem konwersji. TU jednak ciągle mówimy

Jeśli natomiast funkcja spodziewa się *adresu* (czyli wskaźnika lub referencji) obiektu klasy podstawowej, a wyślemy jej adres obiektu klasy pochodnej **to problemu nie ma** – zadziałają (niejawnie) konwersje standardowe.

Jeszcze jedna niemożliwość

Dowiedzieliśmy się, że może standardowo nastąpić taka konwersja:

pochodna* ——————→ podstawowa*

Wskaźnik do obiektu klasy pochodnej zamienia się na wskaźnik do obiektu klasy podstawowej. Czyli wskaźnik do samochodu może być obsłużony tam, gdzie oczekuje się wskaźnika do pojazdów. Jeszcze inaczej: do funkcji można wysłać adres obiektu klasy pochodnej, nawet jeśli ona spodziewa się adresu obiektu klasy podstawowej

Nie jest legalna jednak taka konwersja:

pochodna** ——x——→ podstawowa**

czyli wskaźnik do wskaźnika samochodu nie może być obsłużony tam, gdzie obsługuje się wskaźnik do wskaźnika pojazdów.

Wiem, to zupełnie nie działa na wyobraźnię, więc powiem to inaczej. Nielegalna jest taka konwersja:

*pochodna[ ] ——x——→ *podstawowa[ ]

czyli tablica samochodów nie może być obsłużona w funkcji, która obsługuje tablicę pojazdów.

To dlatego, że **tablica** samochodów nie jest rodzajem **tablicy** pojazdów. **Tuzin** samochodów nie jest rodzajem **tuzina** pojazdów. To po prostu tylko tuzin. Ktoś kiedyś dowcipnie powiedział, że torba pełna jabłek nie jest rodzajem torby pełnej owoców. A teraz w terminach C++:

> Do funkcji spodziewającej się adresu tablicy obiektów klasy podstawowej nie można wysłać adresu tablicy obiektów klasy pochodnej.

Sam zobacz -

jeśli jakaś funkcja obsługuje tablicę pojazdów, to potrafi skakać po jej elementach (bo wie, że następny element jest np. 53 bajty dalej). Taka funkcja nie może obsłużyć tablicy samochodów, tablicy wózków dziecinnych, tablicy promów kosmicznych - bo nie może wiedzieć, jak wielkie są elementy wszystkich takich tablic, czyli ile bajtów ma przeskoczyć, by przejść do następnego elementu.

W tym miejscu kończy się program obowiązkowy tego rozdziału.

---

tylko o konwersjach standardowych.

19.10.3 Konwersje standardowe wskaźników do pokazywania we wnętrzu klasy

Ten paragraf proponuję zdecydowanie opuścić przy pierwszym czytaniu. Rzeczy, o których tu rozmawiamy, są bardzo ciekawe, ale dopóki nie oswoisz się z konwersjami omówionymi poprzednio – nie radzę sobie tym zaprzątać głowy.

Zrozumienie tego paragrafu nie jest niezbędne do czytania dalszej części książki.

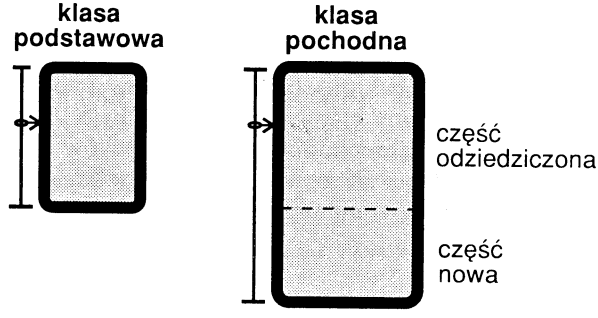
(Zakładam tu, że przeczytałeś rozdział o wskaźnikach do składników klasy – str. 388).

W skrócie można ująć to tak: jest to dokładne odwrócenie poprzedniej zasady.

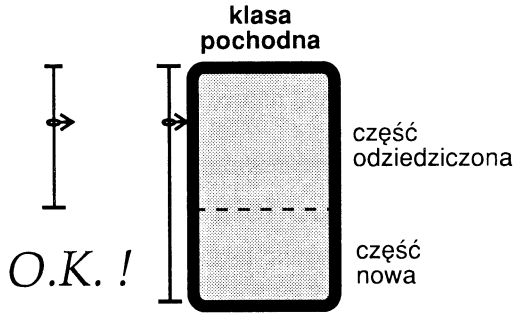
> Wskaźnik do pokazywania na składniki we wnętrzu klasy podstawowej może być niejawnie zamieniony na wskaźnik do pokazywania na składniki klasy pochodnej.
>
> Jest to możliwe, gdy klasa podstawowa jest dziedziczona publicznie i taka konwersja jest jednoznaczna.
>
> Odwrotna zamiana nie następuje automatycznie.

Aby to wytłumaczyć posłużmy się rysunkami. Zobaczymy na nich obiekty klasy podstawowej i pochodnej. Przypominam, że wskaźnik do pokazywania na składniki klasy zawiera informację określającą nie **gdzie** jest dany składnik, ale **jak daleko** od początku obiektu klasy dany składnik zwykle się znajduje. Czy pamiętasz jeszcze naszą historyjkę o planach konstrukcyjnych samolotu `concorde`?

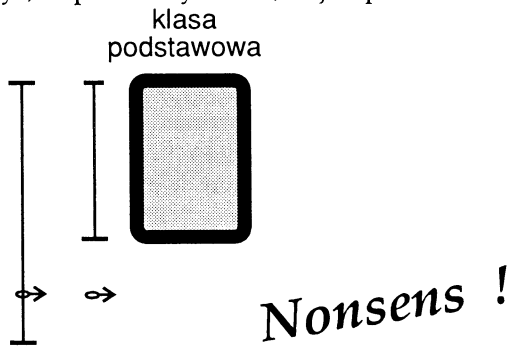
Rysunek pokazuje klasę podstawową i pochodną. Wskaźniki pokazujące na ich składniki są zobrazowane jako suwaki po ich lewej stronie.



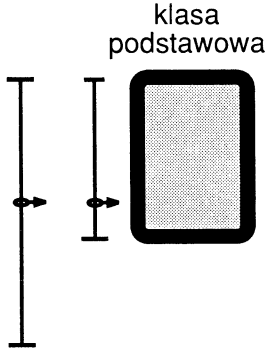
Następny rysunek pokazuje to, co jest przedmiotem tego paragrafu: informacja niesiona przez wskaźnik do składników klasy podstawowej nadaje się do ustawienia wskaźnika do składników klasy pochodnej. Skoro tak, to istnieje konwersja standardowa mogąca zamienić pierwszy z tych wskaźników na drugi.



Odwrotna zamiana nie zawsze miałaby sens i dlatego nie ma takiej konwersji standardowej. Jeśli chcemy to uczynić – musimy zrobić to jawnie i na własne ryzyko. Może się zdarzyć, że pokażemy na coś, co jest poza obiektem.



Jednak taka konwersja przeprowadzona jawnie ("na siłę") nie zawsze musi być błędem. Jeśli wskaźnik składników klasy pochodnej akurat ustawiony jest tak, że pokazuje na coś, co jest odziedziczone od klasy podstawowej – to błąd nie nastąpi.



Przykładowe zastosowanie tej konwersji

to oczywiście sytuacja, gdy wskaźnik tego typu wysyłamy do funkcji. Załóżmy, że funkcja spodziewa się, iż otrzyma wskaźnik do pokazywania na składniki klasy pochodnej (np. Mercedes), a my wysyłamy do niej wskaźnik ustawiony na składnik samochodu (np. składnik kierownica). Ponieważ w Mercedesie też jest kierownica, dlatego konwersja typu wskaźnika jest oczywista.

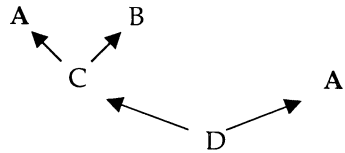
Przypomnienie

Nie ma standardowej konwersji wskaźnika tego typu na wskaźnik `void*` z uwagi na zupełnie inną naturę tego wskaźnika.

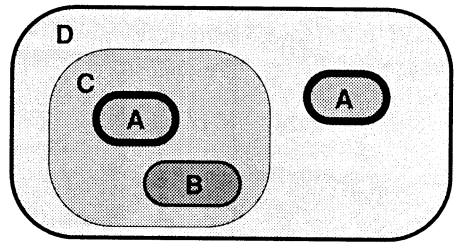
19.11 Wirtualne klasy podstawowe

W tym paragrafie mówić będziemy o specyficznym sposobie dziedziczenia. Jeśli czytasz tę książkę po raz pierwszy, to przeczytaj ten paragraf mając jednak świadomość, że jest to *curiosum*. Jest i taki sposób dziedziczenia – ale nie stosuje się go często. Paragraf jest łatwy, jednak przy pierwszym czytaniu te wszystkie sprawy mogą u Ciebie wyrobić pogląd, że dziedziczenie to coś okropnie skomplikowanego. Tymczasem w praktyce jest to wszystko dość proste.

Mówiliśmy, że przy tworzeniu klasy pochodnej na liście bezpośrednich klas podstawowych dana klasa może się pojawić tylko raz. Nic jednak nie przeszkadza, by taka klasa wystąpiła jeszcze raz w hierarchii, jako dalszy przodek. Oto przykład takiego grafu

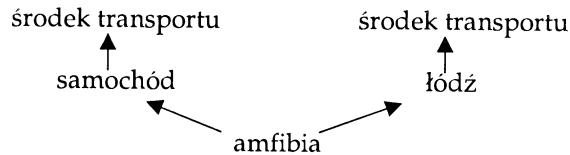


W rezultacie takiego dziedziczenia w obiekcie klasy D mamy kilka obszarów będących „dziedzictwem po przodkach". Obiekt klasy D można sobie schematycznie wyobrazić tak



Zatem w obrębie obiektu klasy D występują dwukrotnie składniki odziedziczone od klasy A.

Czasem to dobrze, czasem źle. Zależy to od tego, co dana klasa reprezentuje. Rozważmy dziedziczenie przedstawione na poniższym grafie:



Klasa środek transportu może wyglądać np. tak

```
class srodek_transportu {
     // ...
protected:
     float polozenie_x ;
     float polozenie_y ;

     void zmiana_polozenia(float delta_x, float delta_y);
     // ..
} ;
```

To, co w `amfibii` jest ze środka transportu – czyli np. jego bieżące położenie albo działanie polegające na zmianie położenia – dziedziczone jest zarówno od samochodu jak i od łodzi. Nie jest to dobre, bo:

- 1) Dwukrotnie odziedziczyliśmy tę samą informację.
- 2) Jest to ryzykowne. Jeśli graf wygląda dokładnie tak, jak to przedstawiliśmy – dostęp do składników tych nie jest jednoznaczny. Próba odniesienia się do składnika `polozenie_x` może dotyczyć zarówno tego odziedziczonego od samochodu, jak i tego odziedziczonego od łodzi.

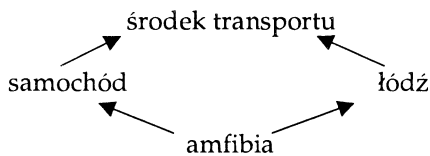
Jest jednak proste rozwiązanie tego problemu. Nazywa się: klasa podstawowa wirtualna

Słowo „wirtualna" (virtual) trudno dokładnie przetłumaczyć. Na nasz użytek umówmy się, że wirtualna oznacza „inteligentnie dziedziczona".

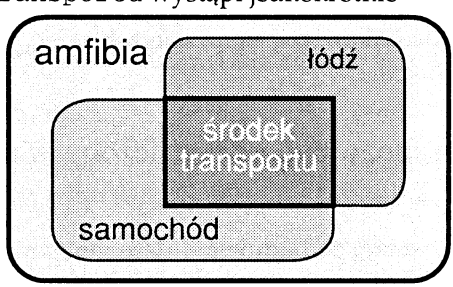
Słowo `virtual` jest przydomkiem, który pojawia się na liście klas podstawowych przed nazwą klasy.

> Dziedziczenie wirtualne przydaje się wtedy, gdy chcemy by w hierarchii był lokalny punkt dzielenia się wspólną informacją.

Jeśli klasę `srodek_transportu` dziedziczyć będziemy jako wirtualną — wówczas graf dziedziczenia wyglądał będzie tak



A zatem teraz w obiekcie klasy amfibia część odziedziczona od (dalekiego) przodka `srodek_transportu` wystąpi jednokrotnie



A oto jak tego dokonujemy. Klasa `srodek_transportu` się nie zmienia. Natomiast dziedziczące ją bezpośrednio klasy samochód i łódź dziedziczą ją teraz wirtualnie:

```
class samochod : public virtual srodek_transportu {
    // ...
} ;

class lodz : public virtual srodek_transportu {
    // ...
} ;
```

Ponieważ klasa `amfibia` nie dziedziczy bezpośrednio klasy `srodek_transportu` - więc definicja jej jest bez zmian. Na liście pochodzenia nie występuje tam przecież klasa `srodek_transportu`.

```
class amfibia : public samochod, public lodz {
    // ...
} ;
```

Zrobione! Teraz zastanówmy się co na tym zyskaliśmy:

- 1) Obiekt klasy `amfibia` się zmniejszył, bo rzeczone składniki opisujące położenie geograficzne już się nie duplikują.
- 2) Nie ma już ryzyka wieloznaczności. Co prawda są teraz dwie drogi do celu (przed samochód albo przez łódź), ale cel jest w obu przypadkach ten sam: dokładnie te same komórki w pamięci.

Warto to zapamiętać:

> Jeśli na dwa sposoby możemy dotrzeć do **tej samej** funkcji, typu, obiektu lub typu wyliczeniowego (enum) – to nie występuje wieloznaczność.

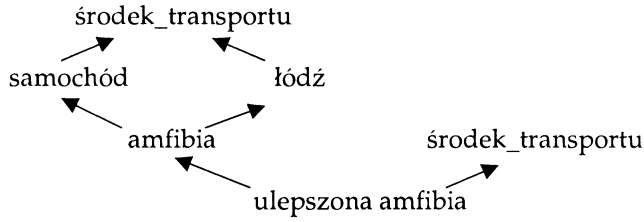
Jak widać, dziedziczenie wirtualne jest sposobem na zamianę zwykłego mechanizmu dziedziczenia. Powiedzmy jednak jasno: to dziedziczenie jest wirtualne, a nie sama klasa podstawowa. Mimo to, mówi się „wirtualna klasa podstawowa", bo to prościej niż: „klasa podstawowa dziedziczona wirtualnie".

Mówiąc tak pamiętajmy jednak, że sama klasa jest najzwyklejszą klasą. Także do jej składników odnosimy się tak samo, jakby były one ze zwykłej klasy podstawowej.

Ta sama klasa może być odziedziczona wirtualnie i niewirtualnie (czyli tradycyjnie).

Wyobraź sobie, że budujemy ulepszoną amfibię – taką, która w razie awarii może podzielić się na pół. Jedna część może iść na dno, a druga oddziela się od niej transportując rozbitków. Ma ona wówczas inną pozycję niż reszta amfibii.

Oto graf tego obrazka



a to definicja klasy

```
class ulepszona_amfibia : public srodek_transportu
{
    // ...
} ;
```

Klasa podstawowa nie jest tu dziedziczona w sposób wirtualny (brak przydomka `virtual`). Jest więc dziedziczona normalnie. W obiekcie klasy `ulepszona_amfibia` będą więc dwa różne komplety składników środka transportu. Czyli właśnie to, o co nam chodziło.

Co to naprawdę oznacza – dziedziczyć jakąś klasę wirtualnie i równocześnie niewirtualnie ?

☞

Jeśli mamy jakąś hierarchię klas, w której ta sama klasa dziedziczona jest czasem wirtualnie, a czasem normalnie – wówczas:

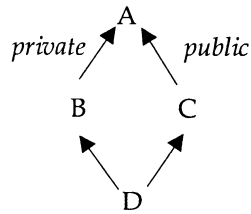
a) w obiekcie klasy pochodnej (tej na samym dole hierarchii) utworzony zostanie jeden wspólny zestaw składników, jako rezultat wszystkich wirtualnych dziedziczeń tej klasy.

b) w obiekcie tym będą także odrębne zestawy składników tej klasy podstawowej – po jednym od każdego dziedziczenia niewirtualnego (zwykłego).

19.11.1 Publiczne i prywatne dziedziczenie tej samej klasy wirtualnej

Wróćmy do przypadku najprostszego. Załóżmy, że klasa podstawowa jest w hierarchii dziedziczona tylko wirtualnie. Mogą być jednak dwa różne sposoby. Znamy je oczywiście.

Wirtualne dziedziczenie klasy podstawowej może się odbyć zarówno w sposób `private` jak i `public`. Oznacza to, że **te same** składniki znajdą się ostatecznie w obiekcie klasy pochodnej jednokrotnie - ale dziedziczone będą na dwa różne i wykluczające się sposoby



Jaki jest rezultat takiego działania? Odpowiedź jest prosta:

Wystarczy, by choć jedno dziedziczenie wirtualne tej klasy podstawowej było publiczne, a efekt jest taki, jakby wszystkie pozostałe dziedziczenia tej klasy były także publiczne.

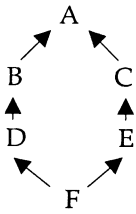
Znowu odpowiada to naszemu doświadczeniu z życia codziennego:

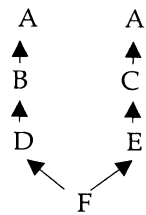
> *O tym samym fakcie dowiaduje się 3 dziennikarzy pracujących dla tej samej gazety (dziedziczą wirtualnie tę samą informację). Dwóch z nich zobowiązuje się do zachowania tego samego faktu w tajemnicy (`private`), a trzeci tajemnicy nie obiecuje (`public`). Wówczas ta informacja dostaje się do gazety i jest publikowana. Nie ma znaczenia dyskrecja dwóch pierwszych dziennikarzy. Liczy się to, że ten trzeci wygadał.*

19.11.2 Uwagi o konstrukcji i inicjalizacji w wypadku klas wirtualnych

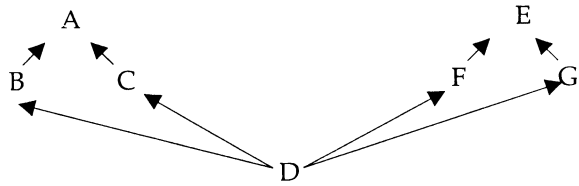
Ten paragraf przy pierwszym czytaniu radzę opuścić.

Skoro dziedziczenie wirtualne sprawia, że w rezultacie w obiekcie klasy pochodnej istnieje tylko jeden „podobiekt" będący dziedzictwem od klasy wirtualnej – zatem konstruktor tej klasy podstawowej powinien ruszyć do pracy tylko raz.





Gdy definiujemy obiekt klasy pochodnej, który ma jakieś wirtualne klasy podstawowe, to te podstawowe klasy wirtualne konstruowane są na samym początku, przed wszystkimi innymi klasami podstawowymi. Bez względu na to, na której pozycji na liście pochodzenia się pojawiły. Jeżeli w hierarchii jest więcej niż jedna klasa dziedziczona wirtualnie, to ich kolejność konstrukcji wynika z kolejności w jakiej pojawiły się na listach pochodzenia. Jest to tzw. zasada „od lewej do prawej".

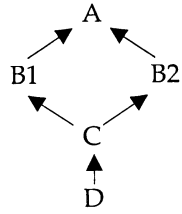


W tym wypadku najpierw będzie konstruowana klasa A a potem klasa E.

Za konstrukcję tego wirtualnego dziedzictwa odpowiada klasa „najbardziej pochodna"

Klasą najbardziej pochodną nazywamy tę klasę, która tworzy obiekt nie będący już podobiektem żadnego innego obiektu.

Czyli np. w hierarchii



gdy mamy definicje **obiektów** różnych klas z tej hierarchii, to klasami najbardziej pochodnymi są przy **ich** kreacji:

```
A    obja ;        // klasa najbardziej pochodna to A
B1   objb1 ;       // klasa najbardziej pochodna to B1
C    objc ;        // klasa najbardziej pochodna to C
D    objd ;        // klasa najbardziej pochodna to D
B2   objb2 ;       // klasa najbardziej pochodna to B2
```

To w klasie najbardziej pochodnej musimy zadbać o uruchomienie konstruktora klasy wirtualnej A.

Do tej pory taka rzecz była niedopuszczalna. Każda klasa pochodna mogła uruchomić tylko konstruktory swoich bezpośrednich klas podstawowych (czyli rodziców). Nie mogła uruchomić konstruktorów klas podstawowych pośrednich (czyli dziadków, pradziadków). Konstruktory dziadków wywoływali rodzice, a pradziadków – dziadkowie, i tak dalej, aż do szczytu hierarchii.

W wypadku dziedziczenia wirtualnego, w konstruktorze klasy najbardziej pochodnej odpowiadamy za uruchomienie konstruktora klasy wirtualnej (czyli u nas klasy A).

„–Jak to!" – zawołasz pewnie – „przecież to sprzeczność: to, która klasa jest najbardziej pochodna, zależy od klasy właśnie definiowanego obiektu. Jeśli zamierzam w programie definiować obiekty wszystkich klas z tej hierarchii, to znaczy, że każda z tych klas będzie w jakiejś sytuacji klasą najbardziej pochodną.

Wobec tego na liście inicjalizacyjnej, którego konstruktora mam w końcu umieścić wywołanie konstruktora klasy wirtualnej? Na wszystkich ? Przecież wtedy przy konstruowaniu obiektu klasy C najpierw uruchomi ten konstruktor klasa B1, potem jeszcze raz klasa B2, potem jeszcze raz klasa C."

Rzeczywiście masz rację. Tak by było w wypadku zwykłego dziedziczenia. Jednak w wypadku dziedziczenia wirtualnego – mimo, że wywołania konstruktora klasy wirtualnej będą rozsiane po całej hierarchii – kompilator *konstruując obiekt klasy* D weźmie pod uwagę tylko to wywołanie z klasy najbardziej pochodnej – czyli z klasy D. Wszystkie inne wywołania znajdujące się na listach inicjalizacyjnych konstruktorów klas podstawowych (rodziców, dziadków, itd.) – zostaną zignorowane.

Jeśli na liście inicjalizacyjnej konstruktora klasy najbardziej pochodnej nie ma wywołania konstruktora klasy wirtualnej, wówczas uruchamiany jest jej konstruktor domniemany. Oczywiście jeśli klasa wirtualna nie ma żadnego konstruktora – to problemu w ogóle nie ma.

Tu praktyczna rada

W piętnastym pokoleniu dziedziczenia programista może już nie pamiętać, że była w hierarchii jakaś klasa wirtualna. Może tego nawet po prostu nie wiedzieć – jeśli klasa jest z biblioteki.

> Definiując klasę, która ma jakieś konstruktory i ma być dziedziczona wirtualnie – dobrze jest więc wyposażyć ją także w konstruktor domniemany. Gwarantuje to nam dochowanie zasady, że użytkownik klasy nie musi znać całej hierarchii dziedziczenia, by użyć jakiejś klasy.

Gdyby konstruktora domniemanego nie było – a klasa miała inne konstruktory - to w wypadku odziedziczenia wirtualnego takiej klasy, każdy konstruktor klasy pochodnej do tysiącznego pokolenia włącznie musiałby zadbać o uruchomienie konstruktora klasy wirtualnej. Każde następne dziedziczenie musiałoby troszczyć się o tego dalekiego przodka. Wyposażenie tej klasy wirtualnej w konstruktor domniemany zwalnia nas od obowiązku pamiętania o tym.

Oto zarys definicji klas, które widzieliśmy w naszym ostatnim grafie. Wszystkie konstruktory – dla prostoty przykładu mają ciała puste.

```
//////////////////////////////////////////////////////////
class A {                          // ona będzie dziedziczona wirtualnie ❶
public:
     A() { };          // konstr domniemany
     A(int) { };             // zwykły
     //...
} ;
//////////////////////////////////////////////////////////
class B1 : public virtual A {                            // ❷
public:
     B1(): A(4) { } ;
     //...
} ;
//////////////////////////////////////////////////////////
class B2 : public virtual A {                            // ❸
public:
     B2(): A(4) { } ;
     //...
} ;
//////////////////////////////////////////////////////////
class C : public B1, public B2 {
```

```
public:
     C() : A(6), B1()                                          // ❹
     { } ;
} ;
//////////////////////////////////////////////////////////////
class D : public C {
public:
     D() : C()                                                 // ❺
     {
     } ;
} ;
//////////////////////////////////////////////////////////////
```

Ciekawsze miejsca

❶ Ponieważ podejrzewamy, że klasa `A` będzie dziedziczona wirtualnie, dlatego zaopatrujemy ją w konstruktor domniemany.

❷ Klasa `B1` dziedziczy klasę `A` w sposób wirtualny. Ponieważ jest to „pierwsze pokolenie" dziedziczenia tej klasy – dlatego nie jest to tak interesujące. Na liście inicjalizacyjnej konstruktora klasy `B1` jest oczywiście wywołanie konstruktora klasy `A` – i nic w tym dziwnego, skoro klasa A jest równocześnie bezpośrednią podstawową. Podobnie w wypadku klasy `B2`. ❸

❹ Gdyby nie istniało dziedziczenie wirtualne, to na liście inicjalizacyjnej konstruktora klasy `C` w żadnym wypadku nie mogłoby się znaleźć wywołanie konstruktora klasy podstawowej niebezpośredniej (czyli nie rodziców, a dziadka).

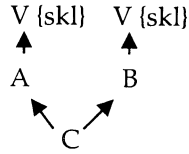
Jednak ponieważ klasa `A` jest w tym drzewie genealogicznym dziedziczona wirtualnie - to **przy definiowaniu obiektu klasy** `C` – wywołanie konstruktora klasy `A` zostanie dokonane z listy inicjalizacyjnej tego ❹ konstruktora. Wywołania umieszczone na listach inicjalizacyjnych rodziców klasy `C` czyli klas `B1` i `B2` – zostaną zignorowane.

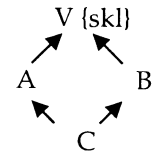
❺ Klasa `D` jest pochodną od klasy `C`. Jeśli definiujemy obiekt klasy `D`, to oczywiście klasą najbardziej pochodną jest tutaj `D`. Z listy inicjalizacyjnej konstruktora tej klasy `D` ❺ zostanie wywołany konstruktor klasy wirtualnej `A`. Ponieważ jednak na liście inicjalizacyjnej nie ma tu wywołania określonego konstruktora klasy wirtualnej `A`, wobec tego uruchomiony zostanie konstruktor domniemany z klasy `A`. Natomiast wszystkie ewentualne wywołania konstruktora klasy `A`, które mogą być napotkane w hierarchii: u rodzica `C`, u dziadków `B1`, `B2` – zostaną zignorowane.

19.11.3 Dominacja klas wirtualnych

Także i ten paragraf proponuję opuścić przy pierwszym czytaniu. Mówimy to o sprawach, których w pierwszej fazie nauki C++ nie będziesz na pewno potrzebować.

Przypominasz sobie zapewne, że wprowadzenie wirtualnego dziedziczenia klas uchroniło nas przed wieloznacznością dostępu. Widać to na poniższych grafach.



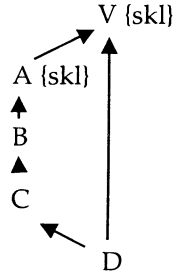


Na grafie z lewej strony – jeśli klasa `C` odnosi się do nazwy `skl` to okazuje się, że są dwie takie nazwy. Występie więc wieloznaczność. Przy ewentualnym odwołaniu się z klasy `C` do takiej nazwy `skl` - kompilator zaprotestuje.

Na grafie z prawej, gdzie klasa `V` jest dziedziczona wirtualnie, nie ma wieloznaczności – bo nazwa `skl` występuje tu tylko raz. Co prawda są do niej dwie drogi

`A::V::skl` *oraz* `B::V::skl`

ale obiema tymi drogami trafiamy do tego samego celu. Może być jednak inna sytuacja. Spójrz na odnośny graf.



W klasie `V` (dziedziczonej wirtualnie) jest składnik `skl`. natomiast dodatkowo klasa pochodna `A` także definiuje składnik `skl`. Są więc dwa składniki o tej samej nazwie.

Załóżmy, że teraz w klasie `D` użyjemy w wyrażeniu nazwy `skl`. Do którego z tych dwóch składników wówczas się odniesiemy?

Nie próbuj wymyślać odpowiedzi, bo intuicja może tutaj zawieść – Otóż nazwa A::skl **dominuje** tutaj nad nazwą `V::skl`

Co to znaczy dominuje?

> Mówimy, że nazwa `A::skl` dominuje nad nazwą `V::skl` wtedy, gdy klasa `A` jest jedną z klas pochodnych klas `V`.

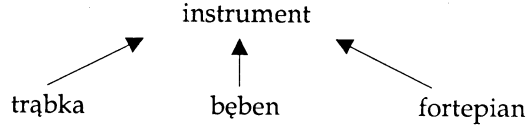
To była definicja jakby bardziej formalna. Ponieważ ja lubię definicje mniej formalne więc ująłbym to tak:

Jeśli, gdzieś w hierarchii, jedna nazwa `skl` zasłania choć raz inną nazwę `skl` – to znaczy, że ta zasłaniająca dominuje. Choćby nawet obecnie to zasłonięcie nie było skuteczne. W naszej hierarchii patrząc z klasy `B`, `C` nazwa `skl` z klasy `A` zasłoniła nazwę `skl` z klasy `V`. Nic nie szkodzi, że teraz z klasy `D` widać obie nazwy `skl`. Zostaje ta niesława: wszyscy w rodzinie i tak szeptają po kątach: „zdarzyło się kiedyś, że ten `skl` z klasy `A` zasłonił tego z klasy `V` ".

Posługiwanie się klasami wirtualnymi jest, jak widać, nieco trudniejsze niż klasami niewirtualnymi. Nie ma się czym jednak przejmować, bo tego sposobu dziedziczenia nie stosuje się często.

20 Funkcje wirtualne

W rozdziale tym mówić będziemy o wspaniałym ukoronowaniu całego mechanizmu dziedziczenia czyli o tzw. funkcjach wirtualnych. To one właśnie sprawiają, że program jest naprawdę obiektowo orientowany – inaczej mówiąc: orientuje się według obiektów.
Brzmi to bardzo patetycznie, jednak sprawa jest bardzo prosta. Dla przykładu spójrz na poniższą hierarchię.



A oto realizacja tych klas. W zasadzie nie ma w nich nic szczególnego poza jednym słówkiem `virtual` przed funkcją składową w klasie `instrument`. Nie daj się zmylić – słowo to nie stoi na liście pochodzenia – zatem nie ma nic wspólnego z poprzednim znaczeniem. Tutaj postawiłem to słowo przy deklaracji funkcji składowej.

```
/*-----------------------------------------------------------
                    plik : instrume.h
-----------------------------------------------------------*/
#include <iostream.h>
//////////////////////////////////////////////////////////////
class instrument {                                        // ❶
     int cena ;
public:
     void virtual wydaj_dzwiek()                          // ❷
     {
          cout << " Nieokreslony brzdek !\n" ;
     }
     // ...
} ;
//////////////////////////////////////////////////////////////
class trabka : public instrument {                        // ❷
```

```
public:
     void wydaj_dzwiek()
     {
          cout << " Tra-ta-ta !\n" ;
     }
     // ...
} ;
//////////////////////////////////////////////////////////
class beben : public instrument {                    // ❷
public:
     void wydaj_dzwiek()
     {
          cout << " Bum-bum-bum !\n" ;
     }
     // ...
} ;
//////////////////////////////////////////////////////////
class fortepian : public instrument {                // ❷
public:
     void wydaj_dzwiek()
     {
          cout << " Pilm-plim-plim !\n" ;
     }
     // ...
} ;
//////////////////////////////////////////////////////////
```

Jak widać, ten powyższy tekst umieściliśmy w pliku `instrume.h` po to, by móc później do niego jeszcze wracać. Nie ma to nic wspólnego z tematem tego rozdziału. Równie dobrze moglibyśmy to zrobić po staremu. Skoro jednak mamy to już w pliku nagłówkowym, to wystarczy jeśli dyrektywą `include` włączymy sobie tekst w program. Jak poniżej

```
#include "instrume.h"
void muzyk(instrument & powierzony_instrument) ;
/*******************************************************/
main()
{
instrument jakis_instrument ;                        // ❸
trabka     zlota_trabka ;
fortepian  steinway_giseli ;
beben      moj_werbel ;

     cout << "Zwykle wywolania funkcji skladowych\n"
     "na rzecz obiektow - jak dotad nic szczegolnego\n" ;

     jakis_instrument.wydaj_dzwiek() ;               // ❹
         zlota_trabka.wydaj_dzwiek() ;
      steinway_giseli.wydaj_dzwiek() ;
           moj_werbel.wydaj_dzwiek() ;

     cout <<   "Wywolanie funkcji na rzecz obiektu \n"
               "pokazywanego wskaznikiem instrumentu\n" ;

     instrument *wskinstr ;                          // ❺
```

```
    // ustawienie wskaźnika
    wskinstr = & jakis_instrument ;                    // ❻

    wskinstr-> wydaj_dzwiek() ;                        // ❼


    cout << "Rewelacja okazuje sie przy "
             "pokazaniu wskaznikiem\ndo instrumentow"
             " na obiekty klas pochodnych\n"
             " od klasy instrument ! \n" ;

    wskinstr = &zlota_trabka ;                         // ❽
    wskinstr-> wydaj_dzwiek() ;                        // ❾

    wskinstr = &steinway_giseli ;
    wskinstr-> wydaj_dzwiek() ;

    wskinstr = &moj_werbel ;
    wskinstr-> wydaj_dzwiek() ;

    cout << "Podobne zachowanie jest takze \n"
          "w stosunku do referencji \n" ;

    muzyk(jakis_instrument);                           // ❿
    muzyk(zlota_trabka);                               // ❶❷
    muzyk(steinway_giseli) ;                           // ❶❸
    muzyk(moj_werbel) ;
}
/*****************************************************/
void muzyk(instrument & powierzony_instrument)         // ❶❶
{
    powierzony_instrument.wydaj_dzwiek();
}
```

Po skompilowaniu i uruchomieniu programu na ekranie zobaczymy

```
Zwykle wywolania funkcji skladowych
na rzecz obiektow - jak dotad nic szczegolnego
          Nieokreslony brzdek !
          Tra-ta-ta !
          Pilm-plim-plim !
          Bum-bum-bum !
Wywolanie funkcji na rzecz obiektu
pokazywanego wskaznikiem instrumentu
          Nieokreslony brzdek !
Rewelacja okazuje sie przy pokazaniu wskaznikiem
 do instrumentow na obiekty klas pochodnych
 od klasy instrument !
          Tra-ta-ta !
          Pilm-plim-plim !
          Bum-bum-bum !
Podobne zachowanie jest takze
w stosunku do referencji
          Nieokreslony brzdek !
          Tra-ta-ta !
```

```
            Pilm-plim-plim !
            Bum-bum-bum !
```

Gdybyśmy jednak z tego naszego programu usunęli słowo `virtual` stojące przy funkcji `wydaj_dzwiek` w klasie podstawowej `instrument`

wówczas na „drugim" ekranie pojawi się coś takiego

```
Zwykle wywolania funkcji skladowych
na rzecz obiektow - jak dotad nic szczegolnego
      Nieokreslony brzdek !
          Tra-ta-ta !
          Pilm-plim-plim !
          Bum-bum-bum !
Wywolanie funkcji na rzecz obiektu
pokazywanego wskaznikiem instrumentu
      Nieokreslony brzdek !
Rewelacja okazuje sie przy pokazaniu wskaznikiem
 do instrumentow na obiekty klas pochodnych
 od klasy instrument !
      Nieokreslony brzdek !
      Nieokreslony brzdek !
      Nieokreslony brzdek !
Podobne zachowanie jest takze
w stosunku do referencji
      Nieokreslony brzdek !
      Nieokreslony brzdek !
      Nieokreslony brzdek !
      Nieokreslony brzdek !
```

Porozmawiajmy

❶ W programie powyższym mamy do czynienia z czterema klasami. Jedna z nich (`instrument`) jest klasą podstawową. Z niej bezpośrednio wywodzą się trzy klasy pochodne ❷ . Dla prostoty w każdej z nich jest tylko jeden składnik – funkcja składowa `wydaj_dzwiek`. W klasie podstawowej `instrument` przed deklaracją tej funkcji stoi tajemnicze słowo `virtual`.

❸ Przejdźmy teraz do analizy funkcji `main`. Widzimy tutaj definicje czterech obiektów tych klas.

❹ Następnie na rzecz tych obiektów wywołujemy funkcję `wydaj_dzwiek`. Funkcji takich jest w naszym programie cztery (po jednej w każdej klasie). Dla każdego obiektu wywołana zostanie ta wersja funkcji, która jest składnikiem klasy, do której obiekt należy. Słowem: jego własna.

I rzeczywiście: na ekranie widzimy potwierdzenie – każdy obiekt odzywa się charakterystycznym dla siebie głosem. Nic w tym nadzwyczajnego – to wszystko znamy już od dawna.

Ogólnie więc mówiąc: do tej pory mieliśmy wywołania funkcji w taki sposób

obiekt`.wydaj_dzwiek()`

więc kompilator sprawdzał do jakiej klasy należy obiekt i uruchamiał dla niego funkcję składową z właściwej mu klasy.

❺ A teraz uwaga! Będziemy posługiwać się wskaźnikiem. Przeczytajmy jego definicję: `wskinstr` jest wskaźnikiem do pokazywania na obiekty klasy `instrument`. Bardzo ważne jest, że ten wskaźnik jest wskaźnikiem do pokazywania na instrumenty, a nie wskaźnikiem do trąbki, fortepianu czy bębna. Będziemy się zatem posługiwać zapisem wskaźnikowym. Przypominam

wskaźnik-> funkcja_składowa()

❻ Najpierw więc ustawiamy wskaźnik tak, by pokazywał na `jakis_instrument` i...

❼ ...wywołujemy funkcję. Jak się spodziewaliśmy, uruchomiona zostaje funkcja `wydaj_dzwiek` będąca składnikiem klasy `instrument`. Dowodem na to jest „Nieokreślony brzdęk".

❽ Nadchodzi jednak rzecz niesłychana. Ustawiamy wskaźnik do instrumentu tak, by pokazywał na obiekt klasy `trabka`.

Czy w ogóle wolno nam było tak zrobić?

> Tak, wolno było w myśl konwersji standardowej, o której mówiliśmy w poprzednim rozdziale: wskaźnik do obiektu klasy podstawowej nadaje się także do pokazania na obiekt klasy pochodnej. W końcu trąbka to także instrument.

Bardziej formalnie: wskaźnik klasy pochodnej może być niejawnie zamieniony na wskaźnik do jednoznacznie dostępnej klasy podstawowej. – Oto dlaczego wskaźnikowi instrumentu mogliśmy przypisać adres trąbki.

Jednoznaczność jest tu zapewniona, bo jest tylko jedna klasa podstawowa. Dostępna jest ona także, bo klasy `trabka`, `fortepian`, `beben` dziedziczą ją publicznie.

Typ wskaźnika jest inny niż obiektu, ale **nie jest zupełnie różny** od typu obiektu. Łączą ich wszak więzy rodzinne. Tego pokrewieństwa wymagamy w tym rozdziale.

❾ Skoro wskaźnik jest już ustawiony, to znowu wywołujemy funkcję. Jest to linijka będąca kwintesencją tego rozdziału. Pytanie za sto złotych:

Którą funkcję składową `wydaj_dzwiek` uruchomi teraz kompilator ? Dlaczego?

Odpowiedź na pytanie: **którą** – jest prosta – wystarczy spojrzeć na ekran. Została uruchomiona funkcja z klasy trąbka. Trudniej jest odpowiedzieć na pytanie **dlaczego**.

Prześledźmy dwie możliwe drogi rozumowania kompilatora.

Mamy do czynienia z zapisem

wskaźnik -> funkcja( )

Przypomnijmy, że wskaźnik to adres jakiegoś miejsca w pamięci, plus wiedza o typie obiektów, do których pokazywania ten wskaźnik służy. Nasz wskaźnik wie o tym, że służy do pokazywania na obiekty klasy `instrument`.

Rozumowanie a)

Kompilator widzi, że ma do czynienia ze wskaźnikiem do pokazywania na `instrumenty`. Obok tego wskaźnika stoi wywołanie jakiejś funkcji składowej. Sięga więc „na ślepo" do klasy `instrument` i uruchamia tę funkcję.

Kompilator uruchomił więc funkcję właściwą typowi wskaźnika.

Rozumowanie b)

Kompilator widzi, że ma do czynienia ze wskaźnikiem do instrumentu. Nie sięga na ślepo do klasy instrument, tylko używa swojej inteligencji: nie daje się zwieść typem wskaźnika, tylko sprawdza na co on pokazuje. Orientuje się natychmiast, że wskaźnik pokazuje aktualnie na obiekt klasy `trabka` zatem: **orientując się według typu obiektu** uruchamia funkcję właściwą pokazywanemu obiektowi. (A nie właściwą typowi wskaźnika).

Które z tych rozumowań jest poprawne, a które bez sensu?

Obydwa są poprawne. Co zatem sprawiło, że kompilator wybrał wariant b)?

Sprawił to przymiotnik `virtual` umieszczony w deklaracji funkcji składowej `wydaj_dzwiek` w klasie podstawowej `instrument`.

> Z przymiotnikiem `virtual` już się spotkaliśmy przy klasach dziedziczonych wirtualnie. Pamiętasz, mówiłem wtedy, że można sobie to słowo w wolnym przekładzie zastąpić zwrotem „inteligentnie dziedziczona". Podtrzymuję to.
>
> Mam też takie odległe skojarzenie. Wirtuoz – pochodzi od włoskiego *virtuoso*: uczony, wprawny. Zatem mechanizm wirtualności to wyjątkowo wyszukany sposób dziedziczenia.
>
> Naprawdę słowo *wirtualny* znaczy: (teoretycznie) możliwy, mogący zaistnieć. I słusznie – funkcja oznaczona jako wirtualna, może (bo nie musi) być zrealizowana w klasach pochodnych jeszcze raz, lepiej, dokładniej.

Postawienie tego przymiotnika koło funkcji w klasie podstawowej mówi, że od tej pory po wszystkie dalsze pokolenia – kompilator ma użyć swojej inteligencji, gdy chodzi o wywołania tejże funkcji na rzecz obiektu pokazywanego przez wskaźnik†).

Zauważ, że gdy usunęliśmy z programu to słowo `virtual` – to kompilator nie postąpił już tak inteligentnie. Rozumował według wariantu a). Widzimy to na drugim ekranie.

Zauważ też, że jeśli funkcję wywołujemy na rzecz konkretnego *obiektu* – stosując zapis

obiekt.funkcja( )

---

†) a także – jak się za chwilę przekonamy – na rzecz obiektów, do których odnosimy się za pomocą referencji.

to kompilator zachowuje się zawsze poprawnie. Dlatego, że tutaj nie wymaga się od niego żadnej inteligencji: ma obiekt klasy `X` i uruchamia funkcję składową klasy `X`. Zawsze!

> Inteligencja jest mu potrzebna tylko wtedy, gdy na obiekt pokazujemy wskaźnikiem, a typ wskaźnika jest inny niż typ pokazywanego obiektu. Na naszym drugim ekranie widzimy, że tutaj kompilatorowi zabrakło tej inteligencji. Zabrakło bowiem tego słowa `virtual` przy deklaracji funkcji składowej. Funkcja nie była „inteligentnie dziedziczona".

Wróćmy jednak do naszego programu

Widzimy, że wskaźnik do pokazywania na `instrumenty` ustawiany jest na obiekty klasy `fortepian` lub `beben` i podobnie, jak poprzednio, kompilator używa swojej inteligencji, by uruchomić właściwe funkcje.

❿ W dalszej części `main` przekonamy się, że podobnej inteligencji kompilator użyje przy wywoływaniu funkcji składowej na rzecz obiektu określanego przezwiskiem (czyli referencją). Referencja – jak pamiętamy – to jakby inna forma określenia adresu.

Aby było inaczej, nie definiujemy sobie luźnej referencji, lecz definiujemy funkcję globalną, która przysłany do niej obiekt odbiera właśnie przez referencję.

❶❶ Oto ta funkcja. Jest to funkcja globalna, której argumentem formalnym jest referencja obiektu klasy `instrument`. Wiemy od dawna – czyli z poprzedniego rozdziału – że argumentem wywołania takiej funkcji może być również obiekt klasy pochodnej od klasy `instrument`. Konwersja standardowa zamieni przecież referencję do obiektu klasy pochodnej na referencję do obiektu klasy podstawowej – i typ argumentu będzie się już zgadzał.

Funkcja ma nazwę `muzyk`, a przysłany do niej obiekt odbierany jest jako referencja o nazwie `powierzony_instrument`. To na rzecz tak przezywanego obiektu wywoła się tutaj funkcję składową `wydaj_dzwiek`.

Wywołanie ❿ to wywołanie tej funkcji z argumentem aktualnym będącym obiektem klasy `instrument`. Tu nic nas nie dziwi: obiekt jest klasy `instrument`, referencja do niego jest także klasy instrument. `Muzyk` uruchomi więc funkcję `wydaj_dzwiek` z klasy `instrument`.

❶❷ Oto wywołujemy funkcję `muzyk` z argumentem będącym obiektem klasy `trabka`. Dzięki wspomnianej konwersji standardowej na obiekt klasy `trabka` mówi się przezwiskiem `powierzony_instrument` będącym, jak wiadomo, przezwiskiem `instrumentu`. Wewnątrz funkcji `muzyk` napotykamy instrukcję

```
powierzony_instrument.wydaj_dzwiek();  // referencja.funkcja()
```

Kompilator napotyka ten fragment kodu i się zastanawia:

> *Referencja, czyli przezwisko, które widzę jest przezwiskiem nadawanym `instrumentom`. Obok tego przezwiska stoi wywołanie funkcji składowej. Normalnie – myśli sobie kompilator – nie zastanawiałbym się wcale. Jest przezwisko `instrumentu`, więc natychmiast uruchomiłbym funkcję składową `wydaj_dzwiek` z klasy `instrument`. Pamiętam jednak, że tam, przy jej deklaracji, stoi słowo `virtual`, czyli pouczenie, że mam być*

teraz inteligentniejszy. Patrzę więc na obiekt przezwany tym przezwiskiem i orientuję się, że tak naprawdę, to przezwisko owo nadano obiektowi klasy pochodnej `trabka`. *Uruchamiam więc funkcję* `wydaj_dzwiek` *charakterystyczną dla klasy* `trabka`.

Dowodem tego jest odpowiedni wypis na ekran.

❶❸ Gdy za chwilę muzykowi powierzamy `steinway_giseli`, to przy przesłaniu argumentu do funkcji zostanie on również odebrany pod przezwiskiem `powierzony_instrument`, znowu kompilator musi powziąć decyzję. Ten sam fragment kodu zrozumiany zostanie teraz jako wywołanie funkcji `wydaj_dzwiek` z klasy `fortepian`.

20.1 Polimorfizm

Gdyby nie było słówka `virtual` przy deklaracji funkcji `wydaj_dzwiek` w klasie `instrument`, to w opisanej sytuacji zawsze wywoływana by była funkcja składowa z klasy `instrument` – ta wypisująca na ekranie `nieokreslony brzdek`.

Zatem ten fragment kodu w środku funkcji `muzyk`

referencja`.wydaj_dzwiek() ;`

w którym odbywa się wywołanie funkcji wirtualnej – wykazuje *różne formy*. Czasem

referencja`.instrument::wydaj_dzwiek() ;`

czasem

referencja`.trabka::wydaj_dzwiek() ;`

czasem

referencja`.fortepian::wydaj_dzwiek() ;`

Wielość–form to z greki: poli–morfizm. Mówimy, że wystąpił tu **polimorfizm**. Ten fragment kodu, gdzie jest wywołanie funkcji `muzyk`, zmienia się tak, że raz odpowiada wywołaniu tego, raz tamtego.

Dygresja

> To ten fragment kodu, w którym wywołuje się funkcję wirtualną, wykazuje polimorfizm (wielość form). Sama funkcja wirtualna – nie. Podkreślam to dlatego, że spotkałem w polskiej literaturze określenie funkcji wirtualnej jako rzekomo polimorficznej. Nie funkcja jest polimorficzna – to jej wywołanie jest takie!

Ale to jeszcze nie koniec dziwów. Wyobraź sobie, że ja tę funkcję `muzyk` skompilowałem i umieściłem w bibliotece. Mam już ją na dyskietce w postaci binarnej. Oddaję ją notariuszowi na przechowanie. Ty, drogi czytelniku, jeszcze nie umiesz dobrze programować w języku C++, ale wkrótce się nauczysz. Pewnego dnia zdefiniujesz sobie klasę, która będzie klasą pochodną od mojej klasy `instrument`. Na przykład tak:

```
class czytelnikofon : public instrument {
public:
     void wydaj_dzwiek()
     {
          cout << " Jazzy-jazzy ! " ;
     }
};
```

Mając plik nagłówkowy mojej klasy `instrum.h` oraz tylko wersję binarną mojej funkcji `muzyk` wywołasz ją w swoim programie dla swojego instrumentu

```
#include "instrum.h"
main()
{
     czytelnikofon czczcz ;

     muzyk(czczcz);
}
```

Jeśli skompilujesz Twój program oraz zlinkujesz go z moją funkcją `muzyk`, to w rezultacie na ekranie zobaczysz

```
Jazzy-jazzy !
```

Wynika stąd, że funkcja `muzyk`, którą złożyłem u notariusza, by dać Ci pewność, że nic w niej nie zmieniam – zareagowała poprawnie wywołaniem **Twojej** funkcji składowej.

Zawarta w niej instrukcja

```
powierzony_instrument.wydaj_dzwiek() ;
```

zamieniła się tak, że ma teraz formę

```
powierzony_instrument.czytelnikofon::wydaj_dzwiek() ;
```

Stało się to w środku funkcji muzyk mimo, że w momencie, gdy ją pisałem – nie istniał jeszcze ani Twój program, ani definicja klasy `czytelnikofon`.

Jeśli miałeś kiedyś do czynienia z programowaniem w asemblerze, to jeszcze bardziej docenisz ten cud. Mój kompilator pracując nad ciałem funkcji muzyk musiał tę linijkę wywołania funkcji `wydaj_dzwiek` zamienić na instrukcję „skocz do podprogramu" (`Jump to Subroutine`)

`JSR` *adres_właściwej_funkcji*

Ponieważ tych właściwych funkcji mogło być teraz kilka, więc w sumie powstało coś, co w zapisie podobnym do C++ przedstawiałoby się następująco (`ref` – oznacza u nas w skrócie referencję)

```
if(ref  określa obiekt klasy instrument)
{
     ref.instrument::wydaj_dzwiek() ;
}
else if(ref  określa obiekt klasy fortepian)
```

```
{
     ref.fortepian::wydaj_dzwiek() ;
}
else if(ref określa obiekt klasy trabka)
{
     ref.trabka::wydaj_dzwiek() ;
}
else if(ref  określa obiekt klasy beben)
{
     ref.beben::wydaj_dzwiek() ;
}

return ;
```

Jeśli nawet kompilator zrobił to właśnie tak, to zrobił to raz na zawsze — porównując klasę obiektu ze znanymi mu klasami pochodnymi klasy `instrument`. W tym fragmencie kodu nie ma jednak miejsca na porównanie z klasą, którą Ty może jeszcze kiedyś napiszesz. Po skompilowaniu przepadło!

A jednak musi być to zrobione jakoś sprytniej. To skompilowane ciało funkcji `muzyk` – zachowuje się tak, jakby miało tam jeszcze

```
....
else if(ref  określa obiekt klasy czytelnikofon)
{
     ref.czytelnikofon::wydaj_dzwiek() ;
}
```

A także, jakby miało fragment porównujący z klasami, o których nawet nie wiesz, bo je wymyślisz dopiero dzisiaj po kolacji.

Dlaczego to takie ważne? Oczywiście jest to efektowna sztuczka, ale jaki z niej pożytek?

Taki mianowicie, że jeśli mamy gotowy działający program operujący na klasach, a w pewnym momencie pojawi się konieczność rozszerzenia tego programu o nowe obiekty, to dodanie nowej klasy będącej klasą pochodną od już istniejącej – nie wymaga zmian w tysiącach miejsc programu wszędzie tam, gdzie decyduje się jakiej klasy jest obiekt, pokazywany wskaźnikiem lub przezywany referencją.

Przykładowo piszemy program na obsługę różnych urządzeń elektronicznych. Jest ich powiedzmy 100 typów. Jeśli jednak zechcemy dodać typ 101, który jest pochodny od jakiegoś już istniejącego, to w tych miejscach w programie, gdzie następują decyzje: „jeśli to ten typ, to..“ – nie musimy nic zmieniać.

Nawet jeśli dodamy następne 500 typów! Jest jednak warunek – te nowe klasy (typy) urządzeń muszą być pochodnymi od klas już istniejących.

Było to bardzo ogólnikowe tłumaczenie. Do spraw tych jeszcze wrócimy. Poznaliśmy jednak pewną specyficzną cechę języka C++. Nazywamy ją **rozszerzalnością** (ang.– extensibility). Program w razie potrzeby modyfikacji można łatwo rozszerzać o nowe klasy – bez konieczności gruntownych zmian reszty kodu. Pozostała część kodu potrafi na te nowe klasy poprawnie reagować.

Jest to cecha bardzo ważna – gdy pisze się program wiedząc, że będzie on w przyszłości ciągle modyfikowany.

Nic za darmo

Skoro kompilator zachowuje się tak inteligentnie w stosunku do funkcji wirtualnych, to dlaczego nie używa swojej inteligencji zawsze, w stosunku do każdej funkcji składowej?

Inaczej mówiąc: dlaczego wszystkie funkcje nie są wirtualne *a priori*?

Powód jest prozaiczny – za wirtualność się płaci:

- jeśli w klasie jest funkcja wirtualna, to obiekt tej klasy jest nieco większy,
- podejmowanie decyzji o tym, którą funkcję w danym wypadku wykonać trwa pewien dodatkowy czas.

Ten mechanizm nie zawsze jest nam potrzebny, dlatego instalowany jest tylko na nasze życzenie.

Klasy bez funkcji wirtualnych są (nieco) oszczędniejsze jeśli chodzi o zużycie miejsca. Ich funkcje składowe – w wypadku, gdy wywołujemy je dla obiektów pokazywanych wskaźnikami – wywołują się szybciej, niż gdyby ta sama funkcja została określona jako wirtualna

Są języki programowania (np. Smalltalk-80) gdzie wszystkie funkcje są z definicji wirtualne. C++ pozwala programiście na świadomy wybór.

20.2 Dalsze szczegóły

Z dotychczasowych rozważań wiemy już, że funkcja wirtualna to specjalny rodzaj funkcji składowej. Mechanizm wywołania jej jako funkcji wirtualnej ujawnia się tylko, gdy wywołujemy ją za pomocą referencji lub wskaźnika do obiektu klasy podstawowej. O tym, którą wersję funkcji wywołać, decyduje nie typ wskaźnika, ale to, na co on właśnie pokazuje. (W zwykłych funkcjach byłoby odwrotnie).

Cały mechanizm funkcji wirtualnych stosowany jest przez kompilator niejawnie. Uwalnia on więc programistę od niepotrzebnego wysiłku. Programista myśli wtedy **co** trzeba zrobić, a nie **jak** to zrobić.

Dygresja:

> *W naszej funkcji* `muzyk` *napisaliśmy tylko* **co** *zrobić:* `wydaj_dzwiek`. *Kompilator sam decydował* **jak** *na pokazywanym instrumencie wydać dźwięk.*

Odsunięcie tych decyzji od używającego klasy programisty jest jakby wyszukaną formą enkapsulacji.

Funkcja składowa jest wirtualna wtedy, gdy w definicji klasy przy jej deklaracji stoi słowo `virtual`, lub gdy w jednej z klas podstawowych tej klasy **identyczna** funkcja zadeklarowana jest jako `virtual`.

Identyczna – to znaczy z identyczną nazwą, argumentami i typem rezultatu. (Mówimy czasem – z identyczną **sygnaturą**). Musi być więc dokładnie dopasowanie. Jeśli tak nie jest, to funkcja z klasy pochodnej nie jest funkcją wirtualną.

Dygresja:

Wymóg o zwrocie tego samego rezultatu jest zupełnie oczywisty. Powstałby straszny bałagan, gdyby w danym (polimorficznym!) miejscu programu funkcja zależnie od bieżącej decyzji zwracała czasem liczbę `int`*, czasem wskaźnik do funkcji, a czasem nic.*

Słowo `virtual` może wystąpić tylko raz w klasie podstawowej i nie musi już powtarzać się przy analogicznych funkcjach w klasach pochodnych. (Chociaż może). Wszystkie funkcje o takiej sygnaturze w następnych pokoleniach (nawet tych jeszcze nie wymyślonych) będą automatycznie wirtualne.

Wirtualną może być tylko funkcja składowa. Funkcja globalna nie może być wirtualna. Łatwo to zapamiętać przypominając sobie nasze wolne tłumaczenie słowa `virtual` – „inteligentnie dziedziczona". Skoro dziedziczona, to tylko składowa, bo przecież dziedziczenie jest atrybutem klas. Globalne funkcje przecież nie dziedziczą. Ani inteligentnie, ani głupio.

Słowo `virtual` pojawia się tylko przy **deklaracji** funkcji wewnątrz ciała klasy. Jeśli definicja funkcji składowej jest poza ciałem klasy, to przy definicji funkcji nie powtarza się już słowa `virtual`.

```
class X {
     // ..
     public :
          void virtual fun(int) ;        // tylko deklaracja
} ;
/////////////////////////////////////////////////////////////
void X::fun(int i )                      // definicja funkcji
{                                        // tu już nie ma słowa virtual
     // ...
}
```

To dlatego, że to klasa ma wiedzieć czy jej funkcja jest funkcją wirtualną. Po prostu to klasa musi się do tego w pewien ukryty sposób przygotować.

Czy klasa pochodna musi koniecznie zdefiniować swoją wersję funkcji wirtualnej?

Nie, nie musi. Jeśli nie zdefiniuje, to będzie wywołana owa wersja z klasy podstawowej. To chyba oczywiste. Gdybyśmy w naszej klasie `beben` nie definiowali swojej wersji funkcji `wydaj_dzwiek`, to dla niej użyta będzie ta wersja z klasy podstawowej `instrument`.

Jaki jest dostęp do funkcji wirtualnej, która w klasie podstawowej ma inny dostęp niż w klasie pochodnej?

Konkretniej: funkcja wirtualna w klasie podstawowej `PODST` jest zdeklarowana z dostępem `public`, a tymczasem w klasie pochodnej `POCH` – `protected` lub `private`. Jaki jest wówczas dostęp do takiej funkcji w wypadku wywołania?

Zależy to tylko od wskaźnika (referencji), którym posługujemy się przy wywoływaniu. Dostęp jest taki, jaki ma ta wersja tej funkcji wirtualnej w klasie, z której jest wskaźnik (referencja).

Jeżeli więc w wywołaniu

```
wsk -> funkcja()
```

wskaźnik pochodzi z klasy podstawowej `PODST`

```
PODST *wsk ;
```

to funkcja ma dostęp taki, jak w klasie `PODST`. W naszym wypadku `public`.

Jeśli natomiast wskaźnik jest z klasy pochodnej

```
POCH *wsk ;
```

to funkcja ma taki dostęp, jaki ma w klasie pochodnej (u nas `private`).

Zwracam uwagę: nie ważne jest na co wskaźnik bieżąco pokazuje. Przy rozsądzaniu dostępu (w czasie kompilacji oczywiście) sprawdzony może być przecież tylko typ tego wskaźnika.

`static` – zabroniony

Funkcja wirtualna nie może być funkcją składową typu `static`. Przypominam, że przydomek `static` mówił, że funkcja jest wywoływana nie na rzecz jakiegoś konkretnego obiektu, ale na rzecz całej klasy. Mechanizm wywoływania funkcji wirtualnej jest jednak taki, że o tym, którą wersję funkcji wywołać, decyduje się na podstawie **obiektu**. Musi być więc wywołanie na rzecz obiektu (znanego z przezwiska, lub pokazywanego wskaźnikiem) – ale obiektu!

Do tego może dojść tylko wtedy, gdy funkcja składowa nie jest `static`.

Wirtualność nie zabrania przyjaźni

Funkcja wirtualna może być przez jakąś inną klasę zdeklarowana jako przyjaciel – `friend`. Wirtualność tu nie przeszkadza tej przyjaźni.

Ale uwaga: funkcja zaprzyjaźniona - żeby nawet w jakiejś swojej klasie była polimorficzna - nie może zachować się polimorficznie wobec naszej klasy.

Krótko mówiąc nie ma wirtualnych przyjaciół. Są przyjaciele i już. Ich ewentualna wirtualność jest tylko na użytek ich własnych klas. Nie mogą oni pracować w niższych partiach naszej hierarchii - po prostu dlatego, że przyjaźń nie jest dziedziczna.

Pomyślisz pewnie: „Szkoda, czasem by się przydało by przyjaciel instrumentu (ojca), mógł zadziałać dla fortepianu (syna) i wykonać jakąś akcję specyficzną dla fortepianu".

Zastanówmy się jednak po co nam to jest potrzebne: chcemy mieć zachowanie wirtualne - ale dostępne nie dla byle kogo, jedynie dla przyjaciół.

Gdybyś chciał mieć taki efekt, to możesz posłużyć się następującym chwytem:

> Przyjaciel niech nie robi niczego specjalnego - tylko wywołuje wirtualną funkcję w klasie instrument (ojciec). Tę funkcję wirtualną ukrywamy w klasie słowem `protected`. Mamy wówczas zachowanie wirtualne – bo to przecież zwykła funkcja wirtualna – ale możliwa do uruchomienia jedynie przez swoich i przyjaciół.

Załóżmy, że chodzi o klasę instrument. Dbamy o wszystkie instrumenty i nie dopuszczamy by na nich brzdąkał byle kto. Funkcję `wydaj_dzwiek` deklarujemy w klasie instrument jako `protected`, a funkcja `muzyk` deklarowana jest jako przyjaciel. Teraz `muzyk` ma prawo wydać „wirtualny " dźwięk na dowol-

nym instrumencie w hierarchii (mimo, że przyjaźń z `muzykiem` deklarowała tylko klasa instrument!).

20.3 Wczesne i późne wiązanie

W tym paragrafie nie powiemy nic nowego. Chodzi tu tylko o to, by nazwać zjawisko, które właśnie zaobserwowaliśmy.

Wczesne wiązanie

Jeżeli kompilujemy klasyczny program – „klasyczny", to znaczy taki bez funkcji wirtualnych – wówczas już na etapie kompilacji odbywa się powiązanie wywołań funkcji z adresami określającymi, gdzie są te funkcje.

Rezultatem takiej decyzji jest instrukcja, by skoczyć w dokładnie takie-a-takie miejsce w pamięci i wykonać znajdującą się tam funkcję. Ponieważ wszystkie te decyzje mogą się odbyć w czasie kompilacji – nazywamy to wiązaniem w czasie kompilacji – lub krócej: **wczesnym wiązaniem** (ang. – early binding).

Nie znaczy to, że w programie nie możemy nigdzie podejmować decyzji o tym, którą funkcję wykonać. Znasz przecież zapis

```
switch(wariant){
     case 1 :
          obj.funkcja1() ;          // tu jest pewne !          ❶
          break ;
     case 2 :
          obj.funkcja2() ;          // ...już w czasie kompilacji !  ❷
          break ;
     case 3 :
          obj.funkcja3() ;          // także i tu !             ❸
          break ;
}
```

Rzecz w tym, że musimy to podejmowanie decyzji sami zaprogramować.

Kompilator zaś widząc takie linijki jak ❶, ❷, ❸ – nie podejmuje tu żadnej decyzji. Gdy widzi wywołanie

```
obj.funkcja2()
```

tłumaczy to jednoznacznie i zawsze na skok do wykonania funkcji `funkcja2(void)`.

Późne wiązanie

W wypadku, gdy posługujemy się funkcjami wirtualnymi, to możemy uwolnić się od konieczności programowania podejmowania decyzji co do typu pokazywanego obiektu. Tę pracę zleca się komputerowi. Kompilator wygeneruje taki kod, że decyzja o powiązaniu wywołania funkcji z określoną wersją funkcji wirtualnej – będzie podejmowana dopiero na etapie wykonywania programu.

Mówimy, że jest to **wiązanie na etapie wykonania** lub **późne wiązanie** (late binding).

Wczesne wiązanie – to wiązanie na etapie kompilacji.
Późne wiązanie – to wiązanie na etapie wykonania.

W naszym zapisie

wskaźnik->funkcja_składowa( )

(będącym przecież wywołaniem funkcji) nie możemy na etapie kompilacji powiedzieć na jaki **typ** obiektu będzie pokazywał wskaźnik. Nie wiemy więc o wywołanie której wersji funkcji wirtualnej chodzi. Kompilator nie może tego więc, na etapie kompilacji, zastąpić instrukcją: skocz do funkcji pod takim-a-takim-adresem. Decyzja o tym musi się odbyć w czasie wykonywania programu. Kompilator więc generuje tu pewien ukryty kod, który umożliwi w trakcie wykonywania programu podjęcie takiej decyzji. A nie będzie to kod tak prymitywny, jak instrukcja `switch`. Ten ukryty gotowy jest rozsądzić nawet takie wypadki funkcji, których jeszcze nikt nie napisał i które kiedyś (może) zlinkowane zostaną z programem.

Późne wiązanie pozwala uwolnić nas od programowania aktu podjęcia decyzji. Tę pracę zrobi za nas kompilator. Programuje to (spryciarz) tak, że ten kod potrafi zareagować nawet na takie modyfikacje programu, które zrobią następne pokolenia programistów.

Z drugiej strony widzimy, że wczesne wiązanie, a konkretnie instrukcja `switch` uniemożliwia tę elastyczność. Kiedyś mówiłem, że używanie instrukcji `goto` zdradza, że się jest złym programistą. Teraz można podobnie powiedzieć o używaniu `switch` przy rozsądzaniu typu obiektu. Jeśli stosujemy konstrukcję

```
switch(typ_obiektu)
{
      case typ_A :
            ....
      case typ_B :
            ....
}
```

to utrudniamy sobie w tym miejscu ewentualną późniejszą rozbudowę programu. Lepiej więc posłużyć się tutaj funkcją wirtualną.

20.4 Kiedy dla wywołań funkcji wirtualnych mimo wszystko zachodzi wczesne wiązanie

Nie każde wywołanie funkcji wirtualnej oznacza późne wiązanie. Są sytuacje, gdzie kompilator **może sobie uprościć sprawę** i zastosować zwykłe, wczesne wiązanie. Oto kilka takich sytuacji.

Wywołanie na rzecz obiektu

Jak już wspomnieliśmy, jeśli funkcję wirtualną wywołujemy na rzecz obiektu znanego z nazwy

```
obiekt.funkcja( ) ;
```

to wszystko jest oczywiste już na etapie kompilacji. Z deklaracji nazwy tego obiektu kompilator wie do jakiej klasy on należy, wiadomo więc, którą z wersji funkcji wirtualnej należy wywołać. Następuje więc powiązanie tego wywołania z właściwą funkcją – już w czasie kompilacji.

Jawne użycie kwalifikatora zakresu

Jeśli natomiast wywołujemy funkcję wirtualną za pomocą wskaźnika lub referencji

```
wskaźnik -> funkcja( ) ;
referencja. funkcja( ) ;
```

to wówczas to, na co w danym momencie pokazuje wskaźnik czy referencja – może być wiadome dopiero w czasie wykonania programu.

> *W naszej funkcji* `muzyk` *– za każdym jej wywołaniem – referencja oznaczała coś innego, dlatego to w czasie wykonania programu musi być podejmowana decyzja w każdym konkretnym przypadku.*

Polimorfizm ujawnia się tylko przy wywołaniach za pomocą wskaźnika lub referencji. Ale jeśli zapiszemy tak:

```
wskaźnik->klasa::funkcja( ) ;
```

to mimo, że jest wskaźnik – operatorem zakresu jest tu wyraźnie określone życzenie, która wersja funkcji wirtualnej ma być uruchomiona. Chcemy tę z klasy `klasa`, a nie żadną inną. Skoro nie ma wątpliwości już na etapie kompilacji, to kompilator uznaje, że można zastosować wczesne wiązanie.

Tego sposobu z kwalifikatorem zakresu przy nazwie nie należy jednak nadużywać. Nie jest on elegancki, bo ma w sobie pewną sprzeczność. Z jednej strony użycie w wywołaniu wskaźnika (referencji) sugeruje pewną elastyczność, z drugiej strony kwalifikator zakresu '::' mówi, że żadna elastyczność nas nie obchodzi. Ma być ta wersja funkcji i już!

Program, który ma takie niekonsekwencje, trudniej później modyfikować. Z ogólnego spojrzenia na sąsiednie linijki widzimy przecież, że używa się tu wskaźnika (referencji), więc można tu oczekiwać elastyczności. Jeśli nie dojrzymy tej właśnie linijki z kwalifikatorem zakresu, to możemy pozostać w błogiej naiwności.

Praktyczna zasada jest taka:

> Przy wywołaniu funkcji wirtualnej, kwalifikator zakresu stosujemy tylko wtedy, gdy chodzi nam o sięgnięcie do składników klasy podstawowej – z funkcji składowych klasy pochodnej.
> Prościej mówiąc: by dostać się do tego, co jest zasłonięte.

Wywołanie z konstruktora (destruktora) klasy podstawowej

Trzecią sytuacją, gdy funkcja wirtualna wiązana jest na etapie kompilacji (wczesne wiązanie), jest wywołanie jej z konstruktora (destruktora) klasy podstawowej.

> Jeśli kreujemy obiekt klasy pochodnej, to wedle zwyczaju najpierw startuje konstruktor części odziedziczonej – czyli klasy podstawowej. Jeśli wewnątrz tegoż konstruktora jest wywołanie funkcji wirtualnej, to mimo, że pracujemy dla obiektu klasy pochodnej (na to pokazuje wskaźnik `this`) – uruchomiona zostanie jej wersja z klasy podstawowej. Decyzja o tym zapadnie już na etapie kompilacji (–wczesne wiązanie).

Powód takiej decyzji jest prosty: konstruktor klasy podstawowej pracuje wówczas, gdy obiekt klasy pochodnej nie jest jeszcze w całości skonstruowany. Przecież dopiero po nim rusza konstruktor klasy pochodnej. Kompilator więc nie uruchamia funkcji wirtualnej z klasy pochodnej, bo ta powinna pracować tylko na gotowym, skonstruowanym obiekcie.

Dla bezpieczeństwa uruchamiana jest zawsze ta wersja z klasy podstawowej.

To samo dotyczy destruktora. Gdy pracuje destruktor klasy podstawowej obiekt jest już częściowo rozmontowany. Zrobił to przecież pracujący przed chwilą destruktor klasy pochodnej. Dla bezpieczeństwa wywoływana jest wtedy zawsze wersja z klasy podstawowej. Również i ta decyzja podejmowana jest jako wczesne wiązanie. Sprawa jest bowiem oczywista już na etapie kompilacji.

20.5 Kulisy białej magii, czyli: Jak to jest zrobione ?

Ten paragraf jest tylko dla tych, którzy są ciekawi, jak kompilator koduje podejmowanie decyzji w wypadku wywołania funkcji wirtualnych. Przy pierwszym czytaniu proponuję ten paragraf opuścić, chyba że naprawdę nie możesz z ciekawości wytrzymać.

Nie zamierzam tu opisywać szczegółów implementacji. Podam tylko hasła.

Otóż klasa **podstawowa** w **chwili linkowania** dowie się o wszystkich tych definicjach jej funkcji wirtualnej – które wystąpiły w następnych pokoleniach dziedziczenia. Zbudowana będzie tablica wskaźników do tych wszystkich wersji funkcji wirtualnej. (W tym więc miejscu moja klasa `instrument` dowie się o tym, że Ty właśnie zdefiniowałeś swoją klasę pochodną `czytelnikofon`).

Natomiast jeszcze w czasie kompilacji w miejscach, gdzie w programie są wywołania funkcji wirtualnych, pojawia się kod uruchomienia funkcji, pokazywanej przez wskaźnik schowany w którymś elemencie tej tablicy. W którym? Jeszcze nie wiadomo. To się dopiero okaże w trakcie wykonania programu.

Czyli wywołanie

```
wskobj -> funkcja()
```

zamienia się więc na wywołanie funkcji pokazywanej wskaźnikiem – i do tego takim, który jest zapisany w tablicy `t` (tablica wskaźników do funkcji wirtualnych)

```
(* (wskobj -> t[n]) ) () ;
```

gdzie *n* – zależy od typu rozpoznanej klasy.

To wszystko. Tablica ta nie wchodzi w skład klasy podstawowej. Jest ona sobie gdzieś indziej w pamięci, a w klasie podstawowej ukrytym składnikiem jest tylko wskaźnik do niej. Dlatego taka klasa podstawowa jest tylko troszeczkę większa (o rozmiar tego wskaźnika) – niezależnie od tego, czy tablica ma 2 elementy czy sto.

Jeśli chcesz się przekonać o istnieniu tego ukrytego składnika to zdefiniuj sobie taką klasę

```
class instrument {
     int cena ;
public:
     void virtual funkcja() {}
} ;
```

Rozmiar obiektu tej klasy ocenisz choćby tak

```
cout << "obiekt klasy instrument ma rozmiar "
     << sizeof(instrument) ;
```

A teraz skasuj sobie to słówko `virtual` z deklaracji funkcji `funkcja`. Zobaczysz jak zmieni się rozmiar obiektu. Ta różnica, to właśnie obecny lub nieobecny wskaźnik do wspomnianej tablicy.

Ustaliliśmy już więc, że klasa podstawowa na etapie linkowania dowiaduje się o wszystkich realizacjach swojej funkcji wirtualnej, które pojawiły się w łańcuchu dziedziczenia aż do ostatniego pokolenia. Dowiaduje się dlatego, że powinna je umieć uruchomić. Wskaźniki do tych wszystkich funkcji zostają złożone w tablicy.

Klasa podstawowa nie wie jednak o innych funkcjach wirtualnych, które pojawiły się dopiero po raz pierwszy w dalszych pokoleniach (u jej wnuków).

Jest to zrozumiałe. Klasa `instrument` nie musi wiedzieć o funkcji wirtualnej `chwyt_barre()` zdefiniowanej po raz pierwszy w klasie gitara na użytek dalszych klas pochodzących od niej: klas `gitara_akustyczna` i klasy `gitara_elektryczna`. Klasa `instrument` nie może mieć przecież zachowania `chwyt_barre`, bo takiego zachowania nie ma przecież instrument typu trąbka czy gwizdek.

> W skrócie mówiąc: To tylko w tej klasie, gdzie pojawia się po raz pierwszy deklaracja funkcji `F` będącej `virtual` – tworzy się tablica do wszystkich funkcji wirtualnych o nazwie `F`.

O wszystkich tych trudnych sprawach nie musisz jednak wiedzieć, by móc posługiwać się funkcjami wirtualnymi. Najlepszy dowód: jak prosty był program z instrumentami!

20.6 Funkcja wirtualna, a mimo to `inline`

Choć na pierwszy rzut oka wydaje się to absurdalne – funkcja wirtualna może być zdeklarowana jako `inline` (w-linii).

Jak pamiętamy, wywołanie funkcji `inline` jest kompilowane w ten sposób, że zamiast wywoływać funkcję – wpisuje się jej ciało (całą treść) w linii, w której wywołanie nastąpiło. Nie ma tu więc żadnego wywołania funkcji – jest bezpośrednie wstawianie wszystkich jej instrukcji w dane miejsce w programie.

Nie trzeba się długo zastanawiać by wyczuć, że to absolutnie przeczy polimorfizmowi – czyli mechanizmowi elastycznego wywoływania funkcji wirtualnych.

A jednak, mimo to można funkcję wirtualną zdeklarować jako `inline`. Efekt będzie taki:

1) W sytuacjach, gdy chodzi nam o rzeczywisty polimorfizm (późne wiązanie) – czyli gdy wywołanie funkcji odbywa się dla obiektu klasy pochodnej pokazywanego wskaźnikiem (referencją) do klasy podstawowej – wtedy przydomek `inline` będzie zignorowany. Zniszczyłby przecież wspaniałość polimorfizmu.
2) Są jednak sytuacje, gdy już na etapie kompilacji wiadomo dla jakiego obiektu następuje wywołanie funkcji. (Wczesne wiązanie) – O tych sytuacjach mówiliśmy kilka stron wcześniej.
   Wtedy właśnie brane jest pod uwagę to, że funkcja wirtualna jest (dodatkowo) `inline`.

Już nawet tak robiliśmy – przypomnij sobie nasze instrumenty. Definicje wszystkich funkcji wirtualnych były przecież wewnątrz ciała klas – to przez domniemanie oznacza, że mają być `inline`!

Jednym z miejsc naszego programu, gdzie kompilator skorzysta z faktu, że funkcja jest `inline` będzie instrukcja

```
steinway_giseli.wydaj_dzwiek() ;
```

bo tutaj kompilator może zrobić wczesne wiązanie. Nie będzie jednak uwzględniona cecha `inline` w instrukcji

```
wskinstr-> wydaj_dzwiek() ;
```

bo instrukcja ta wymaga późnego wiązania (jest wskaźnik).

20.7 Pojedynek – funkcje przeładowane contra funkcje wirtualne

Polimorfizmu – czyli mechanizmu wywoływania funkcji wirtualnych nie należy mylić z przeładowaniem funkcji. Jedyne, co te sytuacje cechuje, to identyczność nazw funkcji.

Oto podstawowe różnice

Funkcje wirtualne mają wszystkie

- tę samą nazwę
- i te same argumenty,

co jest możliwe tylko wtedy, gdy każda z nich ma inny zakres ważności.

Funkcje przeładowane mają

- tę samą nazwę
- leżą w tym samym zakresie ważności

co jest możliwe tylko wtedy, gdy mają różne listy argumentów

Zagadka: Spójrz na poniższe definicje klas

```
class pojazd {
     // ...
public:
     void virtual jazda(int) ;                    //❶
} ;

class samochod : public pojazd {
     // ...
public :
     void jazda(int i) ;                          // ❷
     void jazda(void) ;                           // ❸
} ;
```

Jeśli mamy funkcję wirtualną w klasie podstawowej ❶ i pochodnej ❷, ale dodatkowo w klasie pochodnej jest jeszcze jakaś inna funkcja o tej samej nazwie ❸ (siłą rzeczy musi mieć inne argumenty). Czy jest ona wówczas także wirtualna ?

Odpowiedź: Nie. Mechanizm wywołania funkcji wirtualnej ujawni się tylko w stosunku do tej funkcji, która ma identyczne argumenty (sygnaturę), jak ta z klasy podstawowej.

Ta funkcja ❸ jest zwykłą funkcją przeładowującą nazwę `jazda` w zakresie klasy `samochod`.

Przeładowanie funkcji to wczesne wiązanie

dlatego, że już na etapie kompilacji – (na podstawie oceny listy argumentów wywołania) da się określić, która z wersji funkcji ma zostać uruchomiona.

Natomiast w wypadku funkcji wirtualnych nie da się tak określić na podstawie argumentów – wszystkie wersje tej funkcji mają przecież identyczną listę argumentów.

20.8 Klasy abstrakcyjne

Klasa abstrakcyjna to klasa, która nie reprezentuje żadnego konkretnego obiektu.

Na przykład klasa: ssak. Nie widziałem nigdy konkretnego obiektu klasy ssak. Owszem, widziałem psy, krowy, konie, ludzi, które są przecież pochodnymi od ssaka, ale samego ssaka nie widziałem. Klasa ssak sama obiektów nie ma – istnieje po to, by mieć klasy pochodne. Wyobraź sobie, że jesteś rzeźbiarzem i że

kazano Ci wyrzeźbić pomnik ssaka. Co wyrzeźbisz? Czy *to* będzie ssak? Ssak to pojęcie abstrakcyjne – jak je przedstawisz?

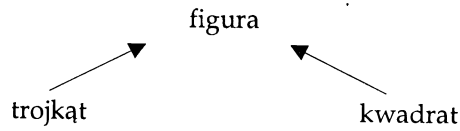
Z klasy abstrakcyjnej ssak wywodzą się klasy koń, pies, człowiek. Te klasy mają już swoje konkretne obiekty, więc nie są już abstrakcyjne. Nasunąć się może pytanie: Po co nam wobec tego klasa, która nie zamierza mieć żadnego obiektu?

Najprostsza odpowiedź brzmi: Po to, by ją dziedziczyć.

Rozważmy taką sytuację. Mamy dwie klasy: klasa trójkąt i klasa kwadrat. Nie pochodzą one od siebie więc nie można ich powiązać zależnością dziedziczenia (żadna nie jest szczególnym rodzajem drugiej). Z drugiej strony jednak obie klasy mają pewien zbiór wspólnych cech. Przykładowo: pozycję na ekranie, gdzie narysowana jest dana figura, kolor w jakim jest to narysowane, pole powierzchni.

Klasy te mają też grupę wspólnych zachowań: daną figurę można narysować, czy też przesunąć w inne miejsce na ekranie.

Gdy potrafimy takie wspólne cechy wyodrębnić, to jest to przesłanka do tego, by zdefiniować klasę abstrakcyjną opisującą tę wspólną część



Do tej pory i w klasie trójkąt i w klasie kwadrat musieliśmy definiować funkcję składową odpowiadającą za położenie figury na ekranie. Dzięki wyodrębnieniu klasy `figura` – pewne funkcje składowe (zachowania) definiowane są tylko raz. Klasy trójkąt i kwadrat włączą te zachowania do siebie za pomocą mechanizmu dziedziczenia.

Najważniejsze jest w tym wypadku to, że klasa figura nie będzie miała żadnego swojego obiektu. Służy ona tylko temu, by ją ktoś odziedziczył.

```
class figura {
protected:
     int pozycja_x ,
      pozycja_y ,
      kolor ;
public:
     void przesun(int delta_x, int delta_y){
          pozycja_x += delta_x ;
          pozycja_y += delta_y ;
     }
} ;
```

Czy nasza klasa abstrakcyjna rzeczywiście nie może mieć żadnych obiektów ? Ależ oczywiście może, oto on:

```
figura f ;
```

Tyle, że nie zamierzamy tego robić. No bo co znaczyłby taki obiekt? Zastanów się do czego może przydać się obiekt `f`? Można nim tylko wykonywać operację `przesun`. To tak, jakbyśmy cudem stworzyli obiekt klasy ssak, który by tylko ssał, ssał i ssał. Nawet by nie wyglądał jak cokolwiek, bo żeby wyglądać trzeba mieć choćby funkcję składową `pokaz_sie()`.

Jak na razie widzimy więc, że korzyść z klasy abstrakcyjnej jest taka, iż oszczędzamy pracy: wspólne cechy dla kilku klas definiujemy jednokrotnie – właśnie w klasie abstrakcyjnej.

Klasa abstrakcyjna to jakby niedokończona klasa. Dokończenie jej jest dopiero zrealizowane przez jej klasy pochodne.

Przejdźmy dalej. Jest jeszcze jedna cecha łącząca klasy `trojkat` i `kwadrat`. Jest to działanie polegające na narysowaniu tych figur na ekranie. Obie te figury mogą być na ekranie narysowane.

Zapytasz pewnie: „No to dlaczego od razu tego zachowania (funkcji składowej) nie wyodrębniliśmy i nie umieściliśmy w klasie abstrakcyjnej?"

Odpowiedź brzmi: bo każda z tych figur może być narysowana – ale każda na swój własny sposób. Sposób rysowania kwadratu jest inny niż sposób rysowania trójkąta. Nie można więc tych dwóch różnych wersji funkcji `narysuj` umieścić we wspólnej klasie abstrakcyjnej `figura`. Ich praktyczne realizacje są bowiem odmienne.

W rezultacie podejmujemy decyzję o definiowaniu tych funkcji w klasach pochodnych – `trojkat` i `kwadrat`. Oznacza to, że można narysować trójkąt, można narysować kwadrat, ale nie można narysować `figury`.

Szkoda. W końcu w życiu codziennym mówimy: „narysujmy tę figurę" myśląc czasem o trójkącie, a czasem o kwadracie.

Zauważ co powiedzieliśmy: narysujmy **tę figurę**. *Tę*– oznacza to, że na coś pokazujemy, czyli mamy wskaźnik. Wskaźnik jest do figury – to jest oczywiste. Z drugiej strony pokazujemy właśnie tym wskaźnikiem na kwadrat albo trójkąt.

Panie i Panowie – jest to klasyczna sytuacja kiedy nasuwa się użycie funkcji wirtualnej. Wskaźnikiem do klasy abstrakcyjnej `figura` pokazujemy obiekt klasy pochodnej (`kwadrat` lub `trojkat`) i żądamy wykonania funkcji właściwej dla klasy pochodnej.

Oto jak powinny wyglądać nasze klasy:

```
class figura {
protected:
     int pozycja_x ,
      pozycja_y ,
      kolor ;
public:
      void przesun(int delta_x, int delta_y){
          pozycja_x += delta_x ;
          pozycja_y += delta_y ;
     }
     void virtual narysuj() = 0;
} ;
//////////////////////////////////////////////////////////
class kwadrat : public figura {
public:
     void narysuj() {
          // instrukcje rysowania kwadratu
     }
} ;
//////////////////////////////////////////////////////////
class trojkat : public figura {
```

```
public:
     void narysuj() {
          // instrukcje rysowania trojkąta
     }
} ;
```

Oto fragment programu, w którym to wykorzystujemy

```
kwadrat k ;                    // definicje obiektów
trojkat t ;

figura *wskfig ;               // definicja wskaźnika do figury
wskfig = &k ;                  // ustawnienie wskaźnika na kwadrat k

wskfig -> narysuj() ;          // wywołanie funkcji wirtualnej
```

Ostatnia linijka oznacza wywołanie funkcji zdefiniowanej w klasie `kwadrat`. Mimo, że `wskfig` jest wskaźnikiem do klasy `figura`, to działa mechanizm funkcji wirtualnej. W trakcie wykonania programu nasz komputer zorientuje się na obiekt jakiej klasy pokazuje właśnie wskaźnik – i uruchomi funkcję właściwą klasie tego obiektu.

Podsumujmy tu, jak doszliśmy do wniosku, że funkcja ma być wirtualną

- 1) W naszej hierarchii wystąpiło zachowanie czyli funkcja, które było cechą kilku klas. To był więc powód, by funkcja znalazła się w klasie abstrakcyjnej.
- 2) Z drugiej strony jednak, praktyczna realizacja tej funkcji jest odmienna dla każdej z klas. Dlatego w klasie abstrakcyjnej postawiliśmy przy tej funkcji słowo `virtual`.

Skoro klasa jest abstrakcyjna to znaczy, że nigdy nie będziemy wykonywać tej realizacji funkcji `narysuj`, która jest w klasie `figura`. Dlatego, że rysuje się kwadraty, trójkąty, może kiedyś w przyszłości kółka, ale nie rysujemy obiektów klasy figura – klasa `figura` żadnych zdefiniowanych obiektów nie ma.

Ponieważ ta realizacja funkcji jest niepotrzebna – możemy ją zdefiniować w klasie abstrakcyjnej tak

```
void virtual narysuj() = 0 ;
```

O takiej funkcji mówimy, że jest **czysto wirtualna**. (ang. pure virtual). Oznacza to, że tej wersji funkcji wirtualnej nie ma się nigdy wykonywać. Może być wykonywana ta z klasy pochodnej `kwadrat`, lub `trojkat`, ale nigdy ta. Ma to oczywiście sens, bo przecież klasa `figura` nie zamierzała mieć żadnych obiektów – a tylko na rzecz jej obiektów ta wersja funkcji wirtualnej mogłaby być wywołana.

Zdefiniowanie tej funkcji wirtualnej jako czysto wirtualnej ma jeszcze jeden odwrotny skutek: oznacza to, że klasa naprawdę nie może mieć żadnego obiektu. Do tej pory ostatecznie mogliśmy sobie zdefiniować obiekt klasy figura – mimo, że nic sensownego nie oznaczał. Jednak gdy klasa ma w sobie choćby

jedną funkcję czysto wirtualną – wówczas nie da się zdefiniować żadnego obiektu tej klasy. Klasa jest naprawdę abstrakcyjna. Po prostu abstrakcyjna i już.

Zasada, że nie może być żadnego obiektu takiej klasy dotyczy naprawdę każdej sytuacji

Zatem:

✧ Kompilator zaprotestuje nie tylko wtedy, gdybyśmy chcieli zdefiniować obiekt tej klasy

```
figura f ;                    // !!!
```

Także w sytuacjach, kiedy tworzyłyby się obiekty chwilowe tej klasy.

✧ Nie możemy więc nigdzie zdefiniować funkcji, która odbierałaby obiekt takiej klasy przez wartość,

```
void funkcja(figura x) ;
```

Wówczas bowiem kompilator tworzy zwykle na stosie obiekt (automatyczny) tej klasy. Skoro jest zabronione tworzenie jakichkolwiek obiektów tej klasy, to kompilator widząc taką funkcję zaprotestuje. (Ale przesłanie przez adres jest możliwe, bo wskaźnik do takiej klasy istnieć może).

✧ Z tego samego powodu funkcja nie może zwracać przez wartość obiektu klasy abstrakcyjnej.

✧ Klasa abstrakcyjna nie może też być typem w jawnej konwersji. I tutaj powstawałby przecież obiekt tej klasy – co jest zabronione.

Jeśli jeszcze pamiętasz nasz przykład z instrumentami, to w wypadku gdybyśmy funkcję wirtualną `wydaj_dzwiek` *z klasy* `instrument` *zdefiniowali (zamienili) jako czysto wirtualną, wówczas niemożliwe byłoby zdefiniowanie w programie obiektu klasy* `instrument`. *I słusznie — przecież instrument to jakaś abstrakcja. Konkretnie mamy do czynienia z waltornią, skrzypcami, grzechotką czy gwizdkiem.*

Natomiast nasza funkcja `muzyk` – pracująca na referencji obiektu klasy `instrument` – jest nadal poprawna. Referencja do obiektu to przecież nie sam obiekt.

Jeśli decydujemy, że jakaś funkcja ma być wirtualna — to mamy trzy ewentualności:

a) zdeklarować funkcję jako wirtualną (taką zwykłą, nie będącą: czysto wirtualną). Musimy wtedy dostarczyć także definicję tej funkcji dla tej klasy (czyli jej ciało)

```
class K1 {
     // ...

     void virtual funkcja
     {
          // ... ciało funkcji
```

```
        }
    } ;
```

b) zdeklarować funkcję jako czysto wirtualną. Definicja funkcji w tej klasie jest wtedy nieobowiązkowa, więc jej nie podajemy,

```
class K2 {
    // ...
    void virtual funkcja() = 0 ;
} ;
```

c) zdeklarować funkcję jako czysto wirtualną i mimo wszystko podać jej definicję dla tej klasy †)

```
class K3 {
    // ...

    void virtual funkcja() = 0
    {
        // ... ciało funkcji
    }
} ;
```

Jakie mamy korzyści z użycia tych sposobów?

❖ Wypadek a)

(czyli funkcja w klasie abstrakcyjnej jest wirtualna i jest też tam jej definicja) – daje nam to, że jeśli w jakiejś klasie pochodnej nie zdefiniujemy swojej wersji funkcji wirtualnej, to uruchomiona zostanie ta z klasy podstawowej.

Czy ma to sens? – pomyślałeś – na przykład jeśli w nowo zdefiniowanej klasie `dwunastokat` nie zdefiniowałem swojej wersji funkcji `narysuj` – to funkcja z klasy podstawowej nie potrafi go przecież narysować !

Rzeczywiście, ale zostanie uruchomiona po to, by chociaż napisać na ekranie „nie wiem jak to zrobić", albo zapiszczeć ostrzegawczo, albo zrobić coś, co przynajmniej odlegle zasymuluje prawdziwe działanie – na przykład narysuje na ekranie krzyżyk. Ostatecznie może też nic nie robić – jej ciało definiujemy wtedy jako puste `{ }`.

❖ Wypadek b)

(w klasie abstrakcyjnej jest funkcja **czysto** wirtualna i nie ma jej definicji) daje nam to, że jeśli nie zdefiniujemy w klasie pochodnej swojej wersji funkcji wirtualnej, to wówczas przez domniemanie zostanie ona **odziedziczona do klasy pochodnej jako czysto wirtualna.** ††)

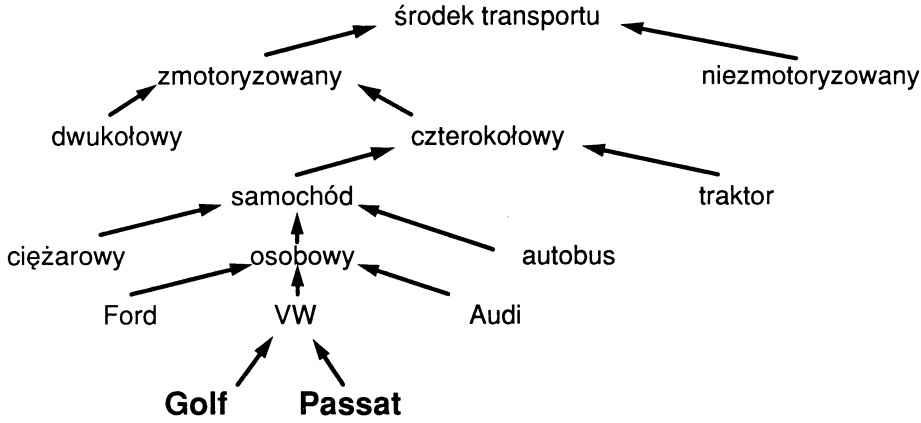
Rozumiesz co to oznacza? Klasa pochodna staje się wówczas automatycznie abstrakcyjną. Jeśli nie było to naszym celem, a tylko o zdefiniowaniu w klasie pochodnej zapomnieliśmy – to kompilator nam o tym zaraz przypomni, bo

†) Uwaga miłośnicy Borland C++: Z niewiadomych mi powodów kompilator Borland C++ wersja 3.1 nie zgadza się na nasz wariant c)

††) to zmiana w stosunku do wcześniejszych wersji języka.

zaprotestuje, gdy zobaczy w programie liczne definicje obiektów tej klasy (obecnie abstrakcyjnej).

Jeśli funkcji wirtualnej w klasie pochodnej nie zdefiniowaliśmy celowo – to znaczy, że rzeczywiście chcemy, by i ta klasa była abstrakcyjna. Jest to bardzo częste zjawisko. Zwykle w hierarchii jest wiele pięter klas abstrakcyjnych, a dopiero te ostatnie klasy na dole reprezentują konkretne obiekty.



❖ Wypadek c)
to w skrócie mówiąc przypadek b) wyposażony dodatkowo w funkcję, której nigdy nie można użyć. No, może prawie nigdy.

Dlaczego „nigdy" ?

Funkcja wirtualna z klasy podstawowej ma szansę być tylko wtedy użyta, gdy wywołujemy ją na rzecz obiektu klasy podstawowej. W naszym wypadku klasa podstawowa nie ma obiektów. Za pomocą wskaźników i referencji też się nie da, bo nasz wspaniały mechanizm wirtualności zawsze uruchomi nam funkcję właściwej klasy pochodnej.

Dlaczego „prawie" nigdy ?

Dlatego, że są sytuacje, w których mechanizm wirtualności nie działa. Mówiliśmy już o tym (str. 566). Jedną z tych sytuacji jest użycie operatora zakresu

```
wskaznik->K3::funkcja() ;
```

W wyniku tej instrukcji ruszy do pracy funkcja czysto wirtualna z klasy podstawowej `K3` – normalnie niemożliwa do uruchomienia.

Inną taką sytuacją jest wywołanie funkcji wirtualnej z wnętrza konstruktora klasy podstawowej.

Zastanowić by się można co by było, gdybyśmy tak właśnie zrobili (wywołali z użyciem operatora zakresu), a tu nagle okazało się, że jednak nie ma w klasie podstawowej definicji funkcji czysto wirtualnej – czyli że naprawdę mamy do czynienia z wypadkiem b)?

Bjarne mówi, że wtedy efekt takiego wywołania jest niezdefiniowany. Jeśli tak mówi, to znaczy, że możesz się spodziewać najgorszego.

20.9 Destruktor? to najlepiej wirtualny!

Dobrze jest przyjąć taką zasadę:

> Jeśli klasa deklaruje jedną ze swoich funkcji jako `virtual` wówczas jej destruktor deklarujemy także jako `virtual`

Zanim powiem jaka z tego korzyść pragnę Cię uspokoić: wirtualność destruktora jest możliwa mimo, że w klasie pochodnej ma on przecież inną nazwę niż w klasie podstawowej. (Nazwa destruktora to nazwa jego klasy poprzedzona znakiem ~ (wężyk)).

Jest to oczywiście wyjątek, bo zwykłe funkcje mogą być wirtualne tylko wtedy, gdy mają identyczne nazwy i argumenty.

Argumenty – to w wypadku destruktora sprawa prosta – każdy destruktor *musi* mieć przecież pustą listę argumentów. Natomiast co do nazwy... no cóż, udawajmy, że po prostu nazwa destruktora brzmi `destruktor` – wtedy nie będziemy się buntowali, że zasada została pogwałcona.

Najciekawsze jest chyba jednak to, po co jest nam destruktor wirtualny

Łatwo to zrozumieć: skoro w klasie deklarujemy jakąś funkcję wirtualną, to znaczy, że zamierzamy na obiekty klas pochodnych od tej klasy mówić czasami jak na obiekty klasy podstawowej.

Przykładowo: skoro w klasie instrument zdefiniowaliśmy funkcję wirtualną, to znaczy, że na obiekty klas pochodnych (gitary, trąby, fortepiany) zamierzamy czasem mówić przezwiskiem (wskaźnikiem) instrument. Mimo, że chcemy, by wykonały akcję charakterystyczną nie dla instrumentu, ale dla siebie.

Mówimy: „Zagraj na tym instrumencie" – chcąc usłyszeć upojne dźwięki fortepianu, a nie: nieokreślony brzdęk.

> Skoro tak, to można się spodziewać także powiedzenia : „zniszcz ten `instrument`" – powodującego uruchomienie destruktora. Ważne jest, że wtedy – jeśli tym instrumentem był `fortepian` powinien ruszyć do pracy destruktor `fortepianu`.

Tę pozornie trudną sprawę rozwiązuje jedno słowo `virtual` postawione w klasie `instrument` przy deklaracji destruktora. To wszystko.

Dzięki temu również destruktory będą uruchamiane inteligentnie.

O tym, jak ważnie jest uruchomienie destruktora właściwego obiektowi - nie muszę nikogo przekonywać. Wystarczy pomyśleć sobie, że np. konstruktor klasy pochodnej `fortepian` dodatkowo rezerwował jakieś miejsce w zapasie pamięci. Destruktor klasy `fortepian` powinien ten obszar pamięci zwolnić. W wypadku destruktora niewirtualnego przy powiedzeniu – „proszę zniszczyć ten instrument" wykonałby się destruktor klasy `instrument`, a destruktor klasy `fortepian` nie. Miejsce nie zostałoby zwolnione.

Jeśli w jakiejś klasie destruktor zdeklarowany jest jako `virtual`, to odtąd destruktory wszystkich klas wywodzących się od tej klasy – aż do ostatniego pokolenia – będą także `virtual`.

Zatem jeśli w którymś pokoleniu będziemy kasowali instrument klasy (bardzo) pochodnej – flet poprzeczny „piccolo", wówczas – mimo, iż wskaźnik (referencja) jest do klasy instrument – uruchamiany będzie destruktor właściwy klasie flet poprzeczny „piccolo".

Skoro wirtualny destruktor jest tak przydatny, to dlaczego każdy destruktor nie jest automatycznie wirtualny?

Powód jest prosty – i znamy go już. Za wirtualność się płaci wielkością obiektu i czasem wykonania funkcji. Tymczasem nie zawsze jest nam potrzebny ten mądry mechanizm uruchamiania destruktorów. Stosowany jest więc tylko na życzenie programisty. Życzenie to wyrażane jest jednym jedynym słówkiem `virtual` umieszczonym przed deklaracją destruktora w klasie podstawowej.

Zobaczmy przykład.

```
#include <iostream.h>
#include <string.h>
//////////////////////////////////////////////////////////
class instrum {
public :
     void virtual wydaj_dzwiek() {
          cout << "cisza" ;
     }
     //————————wirtualny destruktor
     virtual ~instrum()                                  // ❶
     {
          cout << "Destruktor instrumentu \n" ;
     }
} ;
//////////////////////////////////////////////////////////
class skrzypce : public instrum {                        // ❷
     char *nazwa ;
public :
     //— konstruktor————————————————————————————
     skrzypce(char *firma)
     {
          nazwa = new char[strlen(firma) + 1] ;          // ❹
          strcpy(nazwa, firma) ;
     }
     //— destruktor (wirtualny) ————————————————
     ~skrzypce()
     {
          cout << "Destruktor skrzypiec + " ;
          delete nazwa ;                                 // ❺
     }
     //————————————————————————————————————————
     void wydaj_dzwiek()
     {
          cout << "tirli-tirli ("
               << nazwa << ")\n" ;
     }
} ;
```

```
//////////////////////////////////////////////////////////////
class gwizdek :public instrum {                          // ❸
public :
     void wydaj_dzwiek()     {
          cout << "fiu-fiu \n" ;
     }
} ;
//////////////////////////////////////////////////////////////
class gitara : public instrum {
     char *nazwa ;
public :
     // — konstruktor ————————————————————————
     gitara(char *firma)
     {
          nazwa = new char[strlen(firma) + 1] ;          // ❹
          strcpy(nazwa, firma) ;
     }
     //— destruktor (wirtualny) ————————————————
     ~gitara()
     {
          cout << "Destruktor gitary + " ;
          delete nazwa ;                                 // ❺
     }
     //————————————————————————————————————
     void wydaj_dzwiek()
     {
          cout << "brzdek-brzdek ("
               << nazwa << ")\n" ;
     }
} ;
/*************************************************************/
main()
{
     cout << "Definiujemy w zapasie pamieci\n"
          "trzy instrumenty orkiestry\n " ;

     instrum *pierwszy = new skrzypce("Stradivarius") ;
     instrum *drugi = new gitara("Ramirez") ;
     instrum *trzeci = new gwizdek ;                     // ❻

     cout << "Gramy polimorficznie ! \n" ;

     pierwszy->wydaj_dzwiek() ;                          // ❼
     drugi ->wydaj_dzwiek() ;
     trzeci ->wydaj_dzwiek() ;

     cout << "\nKoncert sie skonczyl, "
              "likwidujemy instrumenty\n\n" ;

     delete pierwszy ;                                   // ❽
     cout << "******************\n" ;
     delete drugi ;
     cout << "******************\n" ;
     delete trzeci ;
}
```

W rezultacie na ekranie zobaczymy

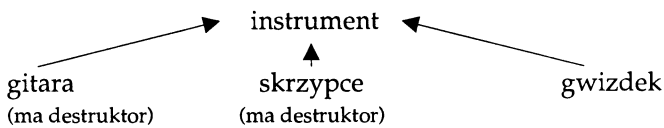
```
Definiujemy w zapasie pamieci
trzy instrumenty orkiestry
 Gramy polimorficznie !
tirli-tirli (Stradivarius)
brzdek-brzdek (Ramirez)
fiu-fiu

Koncert sie skonczyl, likwidujemy instrumenty

Destruktor skrzypiec + Destruktor instrumentu
******************
Destruktor gitary + Destruktor instrumentu
******************
Destruktor instrumentu
```

Omówmy ciekawsze punkty programu

W klasie `instrum` widzimy funkcję wirtualną `wydaj_dzwiek` (znamy to!). Skoro mamy choć jedną funkcję wirtualną, to stosujemy naszą harcerską zasadę, by destruktor był także wirtualny. Tak też robimy w ❶.

Co prawda wcale nie zamierzaliśmy definiować destruktora w tej klasie `instrum`, ale robimy to dla dobra ewentualnych przyszłych pokoleń. Destruktor w tej klasie jest pusty, bo uznajemy, że nie ma co sprzątać przed likwidacją obiektu tej klasy. Jedynie, aby w naszym przykładzie wiedzieć kiedy ten destruktor pracuje – umieszczamy sobie w jego ciele instrukcję wypisującą na ekran.

❷ Klasę `instrum` dziedziczą publicznie klasy `gitara`, `skrzypce` i `gwizdek`. Oto graf:



Klasa gwizdek ❸ jest najprymitywniejsza, bo nie ma konstruktora ani destruktora.

❹ Klasy `gitara` i `skrzypce` są już bardziej wyszukane. Mają konstruktory, które rezerwują w zapasie pamięci miejsce na schowanie stringu– określającego nazwę instrumentu.

Skoro to robią, to powinny mieć destruktor, który odda ten obszar pamięci instrukcją `delete`. To właśnie widzimy w destruktorach. ❺

Destruktory są wirtualne, choć nie stoi przy nich słowo `virtual`. Wystarczy jednak jeśli raz pojawiło się w destruktorze klasy podstawowej `instrum`.

❻ W funkcji `main` widzimy kreację trzech instrumentów. Robimy to nie w zwykły sposób, ale za pomocą instrukcji `new`. Do tych obiektów klasy `skrzypce`, `gitara`, `gwizdek` mamy dostęp jedynie za pomocą wskaźników typu: `instrum *`

❼ O to nam właśnie chodzi, bowiem będziemy bawić się w destrukcję obiektów klasy pochodnej za pomocą wskaźników klasy podstawowej. Najpierw jednak gramy na tych instrumentach krótki koncert.

❽ Kasowanie obiektów udowadnia nam, że mimo, iż pokazujemy na obiekty wskaźnikami do klasy podstawowej – ruszają do pracy właściwe destruktory. Czyli w wypadku obiektu klasy `skrzypce` najpierw destruktor klasy pochodnej skrzypce, a potem podstawowej `instrum`. Dzięki temu zwalnianie rezerwacji znajdujące się w destruktorze skrzypiec dochodzi do skutku.

Zobacz jak wyglądałby odnośny fragment na ekranie, gdybyśmy zlikwidowali słówko `virtual` stojące przy deklaracji destruktora

```
Koncert sie skonczyl, likwidujemy instrumenty

Destruktor instrumentu
******************
Destruktor instrumentu
******************
Destruktor instrumentu
```

A więc destruktory klas pochodnych nie byłyby uruchomione.

Klasa `gwizdek` nie ma destruktora, więc jej jest wszystko jedno czy destruktor w klasie podstawowej jest wirtualny czy nie. Ale tylko jej. Innym klasom to może bardzo przeszkadzać.

Podsumujmy więc co jest istotą tego przykładu.

> Dzięki temu, że w klasie podstawowej (zawierającej jakieś funkcje wirtualne) zdefiniowaliśmy destruktor jako wirtualny – możliwe jest poprawne kasowanie nawet takich obiektów, na które pokazujemy wskaźnikiem do klasy podstawowej.

Na koniec w ramach dygresji zagadka:

> *Co by było, gdyby destruktor w klasie podstawowej `instrum` był zdefiniowany jako czysto wirtualny? Jak by to zmieniło proces likwidacji obiektu klasy `gwizdek`? Gwizdek swojego destruktora nie miał, uruchamiał tylko ten od instrumentu. Co by uruchamiał teraz ?*
> *Odpowiedź brzmi: Nawet się nie zastanawiaj co by `gwizdek` uruchamiał. Obiektu klasy `gwizdek` w ogóle by nie było. Pamiętasz zapewne, że funkcja czysto wirtualna dziedziczy się jako czysto wirtualna. Skoro `gwizdek` w swojej klasie nie ma własnej wersji destruktora, to ta cecha czystości wirtualnej przeszła na klasę `gwizdek` i ona przez to stała się klasą abstrakcyjną. Czyli już w linijce definicji obiektu klasy `gwizdek` kompilator zaprotestuje.*[†)]

†) Przypominam, że tak będzie jeśli masz nowy kompilator. Stara wersja języka zachowuje się inaczej: wymaga w takiej sytuacji od programisty by w klasie pochodnej zdefiniował funkcję wirtualną.

20.10 Co prawda konstruktor nie może być wirtualny, ale...

Tytuł tego paragrafu sprawił, że zaczynasz czegoś żałować. Wierz mi, nie ma powodu.

Kiedy byłby potrzebny konstruktor wirtualny? Wtedy gdybyśmy powiedzieli tak:

> Chcę by powstał obiekt klasy - nie wiem jeszcze jakiej. Teraz, w trakcie pisania programu wiem tylko, że klasa ta będzie klasą pochodną od wiadomej mi już klasy (np. instrument).
>
> Teraz nie wiem jednak jeszcze jak daleki potomek instrumentu to będzie. To, jakiej dokładnie klasy ma być ten obiekt (czyli jaki dokładnie będzie to instrument), okaże się dopiero w trakcie wykonania programu.

Chcielibyśmy zatem by możliwa była taka instrukcja

`new` *wirtualny_konstruktor_instrumentu* ;

Oczywiście to niemożliwe. Już chyba widzisz dlaczego. Aby mógł zadziałać polimorfizm (czyli wywołanie odpowiedniej funkcji wirtualnej) musi być wskaźnik lub referencja do obiektu, którego klasę się rozpozna. Tymczasem konstruktora nie wywołuje się dla obiektu, bo on jeszcze nie istnieje. Nie ma według czego rozpoznawać.

Zresztą zobacz – tak wywoływaliśmy zwykłe funkcje składowe

obiekt . funkcja_składowa() ;
wskaźnik -> funkcja_składowa() ;
referencja . funkcja_składowa() ;

tymczasem konstruktor wywołuje się bez tego zapisu z kropką . lub –>
Piszemy po prostu

konstruktor();

Czyli nasz kompilator nie może się tu zastanowić na co w momencie wykonania programu pokazuje wskaźnik lub co jest przezywane daną referencją.

Nie może być więc konstruktora wirtualnego. Niby to rozczarowanie – ale w takich chwilach rozczarowania trzeba pomyśleć o tym, co w zasadzie chcieliśmy osiągnąć i czy czasem rozumując nie poszliśmy zupełnie błędnym tropem.

Zatem udawajmy, że nic się nie stało i pomyślmy od początku. Co chcemy osiągnąć:

Chcielibyśmy, by można było powiedzieć tak: mam w programie obiekty różnych klas. Tu na przykład klasa instrumenty. Zaraz wyprodukujemy jakiś konkretny instrument. Na razie nie wiadomo jakiej klasy pochodnej konkretnie. Okaże się to dopiero w ostatniej chwili. W trakcie wykonania programu przychodzi jednak taki moment, kiedy już wiadomo. Oto monolog wewnętrzny komputera:

> *Człowiek obsługujący program odpowiada mi, że to ma być instrument klasy ... Na czym polega taka odpowiedź? Ponieważ ja, komputer nie*

rozumiem ludzkiego głosu, więc ta odpowiedź polegać może np. na pokazaniu mi wskaźnikiem jakiegoś istniejącego obiektu danej klasy.
Patrząc na obiekt, który mi wskazano, zorientuję się według niego (wg typu obiektu) i stworzę obiekt tego samego typu. (Tu nastąpi polimorfizm, czyli przy tej orientacji wykaże się wirtualnością czyli inteligencją).

W opowieści tej mimo, że mowa o wirtualności, nie musi istnieć konstruktor wirtualny.

W zasadzie mógłbym już pokazać rozwiązanie, ale zatrzymam się jeszcze. Otóż nie tylko, że rozwiążemy ten problem, lecz nawet dostarczymy

dwa warianty rozwiązania

- Nowotworzony obiekt może być tworzony tak, jakby działał konstruktor domniemany - czyli by nowy obiekt miał standardowe wartości, jakie mają noworodki tej klasy.
- Nowy obiekt może też powstać tak, jakby był zrobiony konstruktorem kopiującym. Czyli będzie miał wartości takie, jakie miał pokazany w ostatniej chwili obiekt przykładowy. Co prawda do tej pory interesował nas tylko jego typ (czy fortepian, czy trąbka, czy bęben) ale co nam szkodzi zainteresować się jego bliższymi danymi (skoro już fortepian, to jakiego koloru, ile waży, i kto jest producentem). Noworodka wyposażymy w te same cechy.

Tajemnica rozwiązania problemu konstruktora wirtualnego polega na tym, że nie trzeba tego robić za pomocą konstruktora. Zrobimy to za pomocą wirtualnych funkcji składowych.

Oczywiście nadać im można dowolne nazwy. Ja zastosuję takie:

- `nowy_dziewiczy` - funkcja zastępująca wirtualny konstruktor domniemany,
- `nowy_wzorowany` - funkcja zastępująca wirtualny konstruktor kopiujący,

Oto program

```
#include <iostream.h>
//////////////////////////////////////////////////////////
class strunowy{                                          // ❶
public :
     int liczba_lat ;
     // ...
     strunowy() : liczba_lat(0)    // kostruktor domniemany
     { }
     //--------------------
     virtual strunowy * nowy_dziewiczy() = 0 ;
     virtual strunowy * nowy_wzorowany() = 0 ;
     //---------------------
     virtual void jestem() = 0 ;
};
//////////////////////////////////////////////////////////
class skrzypce : public strunowy {                       // ❷
     // ...
     virtual strunowy * nowy_dziewiczy( )                // ❸
     {
```

```
            return new skrzypce() ;     // wywołaj konstr. domniemany
        }
        //---------------------
        virtual strunowy * nowy_wzorowany()
        {
            return new skrzypce(*this);// wywołaj konstr. kopiujący
        }
        //---------------------
    public:
        void jestem()
        {
            cout << "Jestm klasy skrzypce, mam lat = "
                << liczba_lat << endl ;
        }
    };
    ////////////////////////////////////////////////////////
    class wiolonczela : public strunowy {
        // ...
        virtual strunowy * nowy_dziewiczy()
        {
         return new wiolonczela() ;      // wywołaj konstr. domniemany
        }
        //---------------------
        virtual strunowy * nowy_wzorowany()
        {
         return new wiolonczela(*this);     // wywołaj konstr. kopiujący
        }
        //---------------------
    public:
        void jestem(){
            cout << "Jestm klasy wiolonczela, mam lat = "
                << liczba_lat << endl ;
        }
    };
    ////////////////////////////////////////////////////////
    main()                                                  // ❹
    {
        skrzypce      skrz ;
        wiolonczela   wiol ;

        wiol.liczba_lat = 157 ;       // niech będzie tak stara

        skrz.jestem();
        wiol.jestem();
        cout << "Teraz bedziemy wirtualnie konstruowac "
                                "dodatkowe obiekty\n" ;
        strunowy * wskaznik ;                               // ❺
        wskaznik = &skrz ;                                  // ❻

        strunowy * wsk1 ;                                   // ❼
        wsk1 = wskaznik->nowy_dziewiczy();                  // ❽

        // wskaźnik pokazywał na skrzypce więc przekonajmy się czy to
        // powstały naprawdę skrzypce.
        wsk1->jestem();                                     // ❾
        //----------------------
```

```
    wskaznik = &wiol ;                                    // ❿

    strunowy * wsk2 = wskaznik->nowy_dziewiczy(); // ❶❶
    wsk2->jestem();

    // następna tak kreowana wiolonczela niech będzie
    // dokładną kopią tej starej

    strunowy * wsk3 = wskaznik->nowy_wzorowany(); // ❶❷
    wsk3->jestem();

    delete wsk1 ;                                         //❶❸
    delete wsk2 ;
    delete wsk3 ;
}
/*****************************************************/
```

Po wykonaniu programu na ekranie zobaczymy

```
Jestm klasy skrzypce, mam lat = 0
Jestm klasy wiolonczela, mam lat = 157
Teraz bedziemy wirtualnie konstruowac dodatkowe obiekty
Jestm klasy skrzypce, mam lat = 0
Jestm klasy wiolonczela, mam lat = 0
Jestm klasy wiolonczela, mam lat = 157
```

Jak to działa?

❶ W programie widzimy klasę `strunowy` - inaczej mówiąc„instrument strunowy". Jest ona klasą abstrakcyjną, bo widzimy w niej funkcje czysto wirtualne. W naszym przykładzie nie było to konieczne, ale dzięki temu deklaracja klasy jest czytelniejsza - bo krótsza. (Nie muszę definiować tych funkcji).
W klasie jest składnik `liczba_lat`, w którym będzie zapisane ile to lat ma dany instrument.

Jest też konstruktor domniemany. W zasadzie tylko po to, by nam do nowych instrumentów (`strunowych`) wpisywał wiek: 0 lat.

Składnikami są też dwie funkcje wirtualne, które zastąpią nam niemożliwe do realizacji wirtualne konstruktory. Spójrz na typ rezultatu tych funkcji. Jest tam `strunowy *`, czyli funkcje zwracać mają adres nowonarodzonego instrumentu strunowego.

Funkcja `nowy_dziewiczy` będzie zajmowała się wirtualną produkcją nowych instrumentów, których składniki będą inicjalizowane wartościami domniemanymi.

Funkcja `nowy_wzorowany` będzie zajmowała się wirtualną produkcją nowych instrumentów, a ich składniki będą inicjalizowane wartościami zobaczonymi u wzorca.

Dodatkowo widzimy też funkcję wirtualną `jestem`, za pomocą której instrument może powiedzieć coś o sobie.

❷ Klasa `skrzypce` jest klasą pochodną od instrumentu strunowego. Widzimy w niej realizacje funkcji wirtualnych.

Funkcja `nowy_dziewiczy` po prostu stwarza operatorem `new` obiekt klasy `skrzypce`. Robi to za pomocą konstruktora domniemanego klasy skrzypce.

Co prawda w tej klasie nie definiowaliśmy konstruktora domniemanego, ale pamiętasz chyba, że jeśli nie ma żadnego konstruktora, to kompilator automatycznie wygeneruje nam taki konstruktor. To nie ma nic wspólnego z omawianym zagadnieniem. Mogliśmy równie dobrze taki konstruktor zdefiniować - nie zrobiłem tego jednak by było prościej.

Funkcja `nowy_wzorowany` także stwarza operatorem `new` obiekt klasy `skrzypce`. Z tą różnicą, że robi to za pomocą konstruktora kopiującego klasy skrzypce.

Również i tego konstruktora nie definiowaliśmy ale pamiętasz chyba, że kompilator zawsze stara się taki konstruktor wygenerować nam automatycznie. Tu się na ten godzimy. (Jeśli by nam nie odpowiadał, to możemy również zdefiniować lepszą wersję).

Zauważ, że nasze funkcje wirtualne po prostu wywołują konstruktory, które same wirtualnymi być nie mogły.

W klasie widzimy też funkcję wirtualną `jestem` - i z jej ciała odczytujemy, co będzie nam pokazywała na ekranie.

Poniżej widzimy deklarację następnej klasy. To inna klasa pochodna od abstrakcyjnego instrumentu strunowego. Jest to `wiolonczela`. Jak łatwo zobaczyć – jest zbudowana podobnie jak skrzypce. Różnica jest oczywiście ta, że jej funkcje wirtualne produkują **jej** typ obiektu.

❹ W funkcji `main` dokonuje się całe misterium produkcji wirtualnej. Najpierw widzimy definicje obiektów różnych klas. Te obiekty w programie służyć nam mogą do różnych celów, jednak w tym przykładzie użyjemy ich tylko do tego, by powiedzieć: „Proszę stworzyć instrument strunowy *takiego* typu."

Skoro *takiego* to znaczy, że musimy mieć wskaźnik do pokazywania na instrumenty strunowe. Jego definicję widzisz w ❺. W momencie jego definicji (i w ogóle w czasie kompilacji) nie wiadomo jeszcze na jaki konkretny instrument będzie on pokazywał. Jest to wskaźnik do instrumentów `strunowych`, ale to nie problem - wiemy przecież, że taki wskaźnik nadaje się również do pokazywania na typy pochodne od instrumentów strunowych. Czyli będzie mógł ewentualnie pokazać na wiolonczelę lub skrzypce.

❻ Już w trakcie wykonywania możemy zapytać operatora o numer buta i datę urodzenia. Na podstawie tej informacji decydujemy jaki obiekt stworzymy mu (wirtualnie) na urodziny. Kiedy już zadecydowaliśmy, odnajdujemy dowolny obiekt takiego typu i ustawiamy na niego wskaźnik.

Ten żart z numerem buta to po to, byś uwierzył że w linijce ❽ naprawdę w czasie pisania programu nie wiadomo jakiego typu stworzymy tu obiekt. To się okaże dopiero w trakcie wykonania programu. Utworzony obiekt będzie nie takiego typu, jak wskaźnik (czyli instrument `strunowy`), ale takiego typu jak obiekt, na który ten wskaźnik właśnie pokazuje.

U nas wskaźnik pokazuje właśnie na obiekt klasy `skrzypce` (tak wynikło z numeru buta) więc polimorficznie uruchomiona zostanie funkcja wirtualna z klasy `skrzypce`. ❸

Skoro w jej ciele jest wywołanie konstruktora klasy `skrzypce`, więc taki obiekt właśnie się narodzi. Dzieje się to za sprawą operatora `new`, a jak wiadomo, tak urodzone obiekty nie mają nazw. Operator `new` daje nam adres nowego obiektu,

więc my powinniśmy sobie zdefiniować wskaźnik do takiego obiektu i do niego ten otrzymany adres wpisać.

To dlatego o linijkę wyżej, czyli w ❼, widzimy definicję wskaźnika, a w ❽ po prostu wpisujemy do tego wskaźnika adres nowego obiektu.

Wskaźnik jest znowu kolejnym wskaźnikiem do instrumentów `strunowych`, ale to znowu nie przeszkadza - może on pokazywać na właśnie stworzone skrzypce.

❾ Dla niedowiarków obiekt przedstawia się - okazuje się, że naprawdę powstały `skrzypce`. (Ta funkcja, to jakby powtórzenie starej znajomej funkcji `wydaj_dzwiek`). Powstał więc obiekt klasy `skrzypce`, a nie klasy instrument `strunowy`.

(Zresztą skoro klasa `strunowy` ma funkcje czysto wirtualne to znaczy, że jest klasą naprawdę abstrakcyjną. Utworzenie obiektu klasy instrument strunowy byłoby po prostu niemożliwe!)

❿ Oto przykład tworzenia innego obiektu tą techniką. Tu ustawiamy nasz wskaźnik na jakiejś `wiolonczeli`.

❶❶ Oto moment stworzenia nowej `wiolonczeli`. Ta linijka wygląda trudniej dlatego, że jest równocześnie definicją wskaźnika `wsk2` i wywołaniem naszej funkcji wirtualnej. Innymi słowy to połączenie linijek ❼ i ❽.

❶❷ Jeśli chcielibyśmy teraz stworzyć następną wiolonczelę, ale będącą prawdziwą kopią już istniejącej, to wystarczy, że wywołamy naszą funkcję `nowy_wzorowany`. Ona zastąpi nam konstruktor kopiujący.

Patrząc na ekran zobaczysz dowód, że ostatnio stworzona wiolonczela ma naprawdę tyle samo lat, co przesłana jej na wzór.

❶❸ Pod koniec programu, oprócz zwykłych dwóch obiektów, istnieją więc dodatkowo jeszcze te trzy, które wirtualnie kreowaliśmy. Możemy je skasować operatorem `delete`.

Wniosek:

Konstruktory nie mogą być wirtualne, bo nie są przecież funkcjami wywoływanymi na rzecz obiektów. To nic nie szkodzi. Tę samą pracę mogą dla nas wykonać napisane prosto a umiejętnie - wirtualne funkcje składowe.

20.11 Finis coronat opus

Funkcje wirtualne, którymi zajmowaliśmy się w tym rozdziale, są jakby ukoronowaniem języka C++. Jeśli nawet nie wszystko z tego rozdziału od razu zrozumiałeś, chciałbym byś zapamiętał najważniejsze: jest w języku C++ mechanizm, który bardzo ułatwi Ci pisanie takich programów, o których już w momencie pisania wiesz, że w przyszłości będą musiały być ciągle modyfikowane.

Funkcje wirtualne to doskonałe narzędzie. Domyślam się jednak, że teraz byś jeszcze nie potrafił powiedzieć, w którym miejscu Twojego programu mogłyby Ci się przydać. Takim praktycznym rozważaniom do czego użyć tego, czego się nauczyliśmy, poświęcimy rozdział ostatni.

Tymczasem zakończyliśmy omawianie języka C++. Następny rozdział omawia już bibliotekę funkcji wejścia/wyjścia – co nie jest już częścią definicji języka. To po prostu instruktaż jak posługiwać się klasami, które ktoś dla naszej wygody napisał.

21 Operacje Wejścia / Wyjścia

Dla odprężenia, porozmawiamy teraz o operacjach wejścia i wyjścia, czyli o tym, jak program porozumiewa się ze światem zewnętrznym. Ze światem zewnętrznym – to znaczy z użytkownikiem siedzącym przed ekranem i klawiaturą, a także z pamięcią zewnętrzną – czyli np. z dyskiem magnetycznym – (po to, by zapisać na nim lub z niego odczytać jakieś dane).

Operacje wejścia/wyjścia nie są zdefiniowane w samym języku C++. Umożliwiają je biblioteki. Takie biblioteki standardowo dołączane są przez producenta danego kompilatora.

Biblioteki standardowe zajmować się mogą wieloma zagadnieniami, nas jednak interesują tutaj te, odpowiedzialne za operacje we/wy.

Jest kilka bibliotek obsługujących te operacje:

- 1) **Biblioteka stdio** (standard input/output), która istnieje po to, by programiści klasycznego C przyzwyczajeni do niej, mogli używać jej także w C++. Biblioteką tą nie będziemy się zajmować.
  Jeśli znasz język C, to znasz dobrze tę bibliotekę i potrafisz się nią posługiwać.
  Jeśli nie znasz klasycznego C i tej biblioteki, to lepiej się jej nie ucz. Używanie tej biblioteki w C++ jest, co prawda, dopuszczalne, ale nie należy do dobrego stylu programowania. (Z drugiej strony jednak może Ci się zdarzyć, że będziesz modyfikował cudzy program, który używa `stdio`. Nie ma wtedy wyjścia – będziesz musiał wziąć do ręki opis tej biblioteki).
- 2) **Biblioteka stream** – jest zbiorem klas zapewniającym operacje we/wy. Tą biblioteką zupełnie nie będziemy się zajmować dlatego, że jest jakby starą wersją biblioteki, którą za chwilę przedstawię w punkcie 3).
  Biblioteka ta dostarczana jest mimo wszystko przez producentów kompilatorów, jednak AT&T zapowiada, że wersja języka C 2.1 jest ostatnią,

która jest jeszcze wyposażona w tę bibliotekę `stream`. Następne wersje mieć jej nie będą.

❖ 3) **Biblioteka iostream** Ta biblioteka jest zalecana do stosowania w programowaniu w C++. Ona właśnie będzie przedmiotem tego rozdziału.

O ile zamierzam dotrzymać obietnicy i nic nie mówić o bibliotece z punktu 2, o tyle nie uda mi się przemilczeć biblioteki `stdio` (punkt 1). To dlatego, że domyślam się, iż duża część czytelników może mieć doświadczenia z klasycznego C i nasuwać się będą uwagi co jest lepsze, a co gorsze i dlaczego.

Zanim przystąpimy do omawiania `iostream` jedna ważna uwaga: rozdział ten nie jest dokładnym opisem jakiejś szczególnej implementacji biblioteki `iostream`. Służy raczej temu, by pokazać możliwości tej biblioteki, a także istotę posługiwania się nią. Jeśli chcesz naprawdę dokładnie poznać bibliotekę `iostream` musisz przeczytać w dokumentacji swojego kompilatora jej opis. Wierzę jednak, że po lekturze tego rozdziału *Symfonii* zrozumienie tamtego opisu nie sprawi Ci kłopotu.

21.1 Biblioteka iostream

Nazwa jest skrótem od: input output stream – ang. strumień wejściowy / wyjściowy.[†)]

Przy omawianiu tej biblioteki spotkamy się z trzema zagadnieniami:

- Wyprowadzanie i wprowadzanie informacji ze **standardowych urządzeń we/wy, takich jak klawiatura i ekran.**
- Podobne **operacje na plikach** danych znajdujących się na nośnikach zewnętrznych – dyskach, taśmach magnetycznych itd.
  Są to np. takie sytuacje, gdy program ma wczytać informację przygotowaną w jakimś pliku dyskowym, lub gdy program ma rezultat swojej pracy zapisać na dysku w postaci pliku.
- Trzecie zagadnienie nie dotyczy wcale komunikacji ze światem zewnętrznym. Jest to sytuacja, gdy w obrębie programu chcemy wyprowadzić informację nie na ekran, nie na plik dyskowy, ale wpisać ją do jakiegoś obszaru pamięci. Na przykład zamiast wypisać na ekran liczbę 3.1416 chcemy to wpisać do jakiejś tablicy typu `char` tak, że poszczególne jej elementy będą takie: 3, kropka, 1,4,1,6, NULL. Ta możliwość jest czasami bardzo przydatna.
  Zagadnienie to nazywa się **formatowaniem wewnętrznym** (ang. incore formatting) i obejmuje także sytuację odwrotną: umożliwić wczytywanie i analizę informacji z takiej tablicy.

†) (iostream – czytamy: „aj-oł strim")

Aby skorzystać z biblioteki `iostream` należy dyrektywą `include` włączyć do programu pliki nagłówkowe zawierające odpowiednie deklaracje. Deklaracje takie znajdują się w kilku plikach nagłówkowych. Niezależnie od tego, z której części biblioteki zamierzamy korzystać – musimy włączyć plik nagłówkowy `iostream.h` Dodatkowo możemy też włączyć plik `fstream.h` - jeśli zamierzamy przeprowadzać z programu operacje na plikach. Jeśli zaś zamierzamy dokonywać operacji we/wy na tablicach znakowych (formatowanie wewnętrzne), wówczas musimy do programu włączyć deklaracje znajdujące się w pliku nagłówkowym `strstream.h`

Oto zestawienie:

iostream.h – jakiekolwiek korzystanie z tej biblioteki

fstream.h – operacje we/wy na plikach zewnętrznych

strstream.h – operacje we/wy na tablicach (formatowanie wewnętrzne)

Dygresja dla miłośników Borland C++.

Niektóre realizacje bibliotek (np. Borland C++) są tak skonstruowane, że np. korzystając z operacji na plikach zewnętrznych wystarczy włączyć jedynie plik nagłówkowy `fstream.h`, gdyż ten i tak sam włącza niezbędny mu plik nagłówkowy `iostream.h`

Jeśli jednak o tym nie wiemy i sami także zastosujemy dyrektywę

```
#include <iostream.h>
```

to nie jest to błędem, plik będzie włączony i tak jednokrotnie.

21.2 Strumień

Wprowadzanie i wyprowadzanie informacji można potraktować jako strumień bajtów płynący od źródła do ujścia.

Na przykład: jeśli chcemy do zmiennej `x` wczytać z klawiatury jakąś liczbę, wówczas strumień (bajtów) płynie od urządzenia zewnętrznego zwanego klawiaturą do tego miejsca w pamięci operacyjnej, gdzie mieści się zmienna `x`.

Skoro poruszamy się po obszarze języka C++ – nic dziwnego, że strumienie są zrealizowane na zasadzie klas.

Wczytywanie lub wypisywanie informacji może się odbywać na dwóch poziomach:

Poziom niższy

- jest najbardziej elementarny. Polega na tym jedynie, że określone bajty informacji przesyłane są od źródła do ujścia – a nie jest istotne znaczenie tych bajtów. Nie są interpretowane w żaden sposób. Odbywa się tylko ich przepływ przez strumień. Za tego typu komunikację odpowiada klasa biblioteczna `streambuf`. Ten poziom stosujemy, gdy nasz program ma przeprowadzać dużo operacji we/wy na bajtach.

Poziom wyższy

- polega na przesyłaniu informacji przez strumień łącznie z interpretowaniem jej – czyli jak to często mówimy – formatowaniem. Ogólnie polega to na tym, że jeśli w jakiejś komórce pamięci zapisana jest liczba `float` 274.81 – to nie wystarczy treść (binarną) tej komórki (czyli cztery bajty) przesłać na ekran. Trzeba wcześniej sformatować tę informację tak, by najpierw zamienić ją na sekwencję znaków: najpierw 2, potem 7, potem 4, potem kropka, potem 8, potem 1. To dopiero popłynie strumieniem na ekran.

Przy wczytywaniu odwrotnie. Jeśli chcemy z klawiatury przyjąć liczbę, a ktoś na niej wystukał znaki `0 x 1 a`, wówczas te znaki są sprowadzane strumieniem z klawiatury muszą zostać zinterpretowane jako liczba w zapisie szesnastkowym (hexadecymalnym). Dopiero rezultat tej interpretacji (czyli jakaś liczba binarna) umieszczana jest w danej komórce pamięci.

Za pracę na tym poziomie odpowiadają klasy:

- `istream` -(input stream) strumień wejściowy (wczytujący),
- `ostream` -(output stream) strumień wyjściowy (wypisujący),
- `iostream` - klasa, która jest pochodną obu powyższych klas (wielokrotne dziedziczenie). Klasa ta umożliwia nam definiowanie swoich strumieni mogących i wczytywać i wypisywać dane.

Tymi właśnie klasami oraz ich pochodnymi posługiwać będziemy się najczęściej.

Aby posłużyć się strumieniem należy:

a) najpierw zdefiniować w pamięci ośrodek dowodzenia strumieniem,

b) wskazać mu to, jakim urządzeniem zewnętrznym ma się zajmować,

c) przeprowadzać konkretne wczytania (wypisywania) informacji – dowolną ilość razy,

d) zlikwidować strumień (razem z jego ośrodkiem dowodzenia) wtedy, gdy uznamy, że strumień nie będzie nam już więcej potrzebny.

Pierwsze Twoje wrażenie jest zapewne takie: tyle pracy po to, by z klawiatury wczytać jedną głupią liczbę x ?

Nie. Jedną głupią liczbę x wczytuje się z klawiatury po prostu instrukcją

```
cin >> x ;
```

Dlaczego to takie proste – mówimy w następnym paragrafie.

21.3 Strumienie predefiniowane

Dla kompilatora jest oczywiste, że jeśli piszesz program, to będziesz zapewne chciał z tego programu wypisać coś na ekran lub wczytać coś z klawiatury. Dlatego kompilator predefiniuje kilka strumieni. Predefiniuje - czyli praca opisana w poprzednim paragrafie w punktach a), b) oraz d) zostanie zrobiona

za Ciebie. Innymi słowy: strumień zostanie założony i otwarty, możesz go używać. Przy zakończeniu programu strumień automatycznie zostanie zamknięty.

Są cztery predefiniowane strumienie:

```
cout      cin      cerr      clog
```

Aby skorzystać z tych strumieni trzeba w programie zamieścić dyrektywę

```
#include <iostream.h>
```

włączającą w trakcie kompilacji odpowiednie deklaracje. Strumienie te są po prostu egzemplarzami obiektów jakichś klas. Na przykład strumień `cout` jest egzemplarzem obiektu klasy `ostream`.

- `cout`— jest powiązany ze standardowym urządzeniem wyjścia (zwykle ekran).
- `cin`— jest powiązany ze standardowym urządzeniem wejścia (zwykle klawiatura).
- `cerr`— jest powiązany ze standardowym urządzeniem, na które chce się wypisywać komunikaty o błędach (zwykle także ekran). Strumień ten jest niebuforowany.
- `clog`— jak wyżej, z tym, że ten strumień jest buforowany. †)

To, że strumień jest niebuforowany oznacza, że jak tylko zażądamy wypisania komunikatu o błędzie – zrobione to zostanie natychmiast.

Buforowanie – oznacza tutaj, że może być robiona pewna optymalizacja, polegająca np. na tym, że dopiero gdy zbierze się kilka komunikatów – wtedy zostaną one wpisane do dziennika pokładowego hurtem. Oszczędza się w ten sposób czas.

21.4 Operatory >> i <<

Przy posługiwaniu się strumieniami bardzo wygodne jest posługiwanie się operatorami << oraz >>. Jak pamiętamy, operatory te w stosunku do typów wbudowanych odpowiadają za przesunięcie bitów w danej komórce pamięci o zadaną liczbę pozycji w lewo lub prawo.

W wypadku strumieni we/wy konieczne było posłużenie się jakimiś operatorami, które by wykonywały funkcje wpisywania (wstawiania, wpuszczania) informacji do strumienia oraz odczytywania (wyjmowania, wyławiania) tej informacji ze strumienia.

Wybór padł na operator >> oraz << między innymi dlatego, że oba wizualnie sugerują ruch. W obrębie klasy strumień wyjściowy (`ostream`) operator << został tak przeładowany, że odpowiada za wysłanie informacji do strumienia. Zapis

†) log pochodzi od ang. log-book = dziennik pokładowy.

```
cout << x ;                    // ruch: od x do cout (czyli ekranu)
```

powoduje zatem, że liczba x zostaje wpuszczona (wstawiona) do strumienia. Strumień ten kończy się na ekranie. (To jego *ujście*).

Operator ten użyty w stosunku do strumieni nazywa się często operatorem *insert* – wstawiania (albo *put to*). Ja powiedziałbym: wysyłania, gdyż informacja wysłana zostaje do strumienia.

Odwrotny operator >> odpowiedzialny za wczytanie informacji

```
cin >> y ;
```

nazywany jest *extract operator* – czyli operatorem ekstrakcji, albo operatorem *get from* (czyli w naszym wypadku pobierz z... , wyłów z ... , wyjmij z ...)

Ponieważ wielokrotnie posługiwaliśmy się już tymi operatorami dlatego wystarczy przypomnieć sposoby posługiwania się nimi.

```
#include <iostream.h>
/*****************************************************/
main()
{
int i;
float f ;
char tekst [80] ;

     cout << "Podaj liczbe int : " ;
     cin >> i ;
     cout << "Podales : " << i << endl ;

     cout << "Podaj liczbe float : " ;
     cin >> f ;
     cout << "Podales : " << f << endl ;

     cout << "Napisz wyraz : " ;
     cin >> tekst ;
     cout << "Napisales ***" << tekst << "***\n" ;
}
```

Posługiwanie się zwrotami: „wstawianie do strumienia" i „wyjmowanie ze strumienia" może być dla Ciebie Czytelniku początkowo mało obrazowe. Dlatego stosujmy też równolegle inne zwroty określające to samo – zrozumiałe nawet dla kogoś, kto nie wie nic o istnieniu strumieni:
– Jeśli wstawiamy informację do strumienia to po to, by popłynęła ona gdzieś – może na ekran – i tam została wypisana. Dlatego też mówić będziemy często **wypisywanie informacji**.
– Natomiast jeśli wyjmujemy coś ze strumienia (płynącego choćby od klawiatury), to jakbyśmy wczytywali informację do programu z klawiatury. Dlatego nazywać będziemy to także **wczytywanie informacji**.

21.5 Domniemania w pracy strumieni predefiniowanych

W wypadku **wypisywania** na ekran (**wstawiania** do strumienia, który płynie na ekran) mamy do czynienia z następującymi domniemaniami:

✧ Typy wbudowane, które służą do przechowywania liczb całkowitych (np. `int`, `long`, `unsigned short`, itd.) są wypisywane w systemie dziesiątkowym.

✧ Typy `char` i `unsigned char` – wypisywane są jako pojedyncze znaki ASCII

✧ Liczby typu `float` i `double` wypisywane są z dokładnością do 6 miejsc po kropce dziesiętnej. Zbędnych zer nie wypisuje się – czyli otrzymamy 1.04 zamiast 1.040000

✧ Wskaźniki (z wyjątkiem `char*` i `unsigned char*`) wypisywane są jako heksadecymalne (szesnastkowe).

✧ Żądanie wypisania wskaźników `char*` (i `unsigned char*`) rozumiane jest jako żądanie wypisania stringu, na który ten wskaźnik pokazuje.
Czyli

```
char string[] = "Napis" ;
char *wsk = string ;
cout << wsk ;
```

spowoduje wypisanie nie tyle adresu, który ten wskaźnik przechowuje, ale stringu znajdującego się pod takim adresem (a kończącego się znakiem NULL).
Gdyby nas jednak mimo wszystko interesował ten adres, to wystarczy posłużyć się rzutowaniem – na przykład na typ `void*`

```
cout << (void*) wsk ;
```

❦

W wypadku **wczytywania (wyjmowania ze strumienia** płynącego z klawiatury), przyjmuje się przez domniemanie że:

✦ Wszystkie wczytywane typy mogą poprzedzać białe znaki (spacje, tabulatory itd.), które są ignorowane.
Zatem: czy napiszemy na klawiaturze

```
11
```

czy też

[spacja][spacja][tabulator]`11`

to nie ma różnicy. Dotyczy to także wczytywania do tablicy znakowej stringu lub pojedynczego znaku. Białe znaki poprzedzające pierwszą „literę" będą zignorowane.

✦ Gdy żądamy wczytania do typów reprezentujących liczby całkowite – znaki przychodzące z klawiatury interpretowane są według konwencji zapisu stałych dosłownych w C++ (str. 35). Oznacza to, że znaki

- `11` zostaną zinterpretowane jako liczba dziesiątkowa (11)

- `011` zostaną zinterpretowane jako liczba ósemkowa (czyli dziesiątkowo = 9)
- `0x11` zostaną zinterpretowane jako liczba szesnastkowa (czyli dziesiątkowo = 17)

✦ Jeśli chcemy wystukując na klawiaturze umieścić przed liczbą znak (+ lub –), to między nim a liczbą nie może być żadnej spacji.

Czyli poprawne jest

```
-3.31
```

a błędne

```
-   3.31
```

✦ Wczytywanie liczby całkowitej zostaje zakończone, gdy napotkany zostaje znak nie będący cyfrą.

```
712p
```

albo

```
712_
```

(znakiem _ oznaczamy tu spację)

✦ Liczbę zmiennoprzecinkową wczytuje się podobnie, z tym, że w wypadku notacji wykładniczej (tzw. scientific notation) może wystąpić po raz drugi znak (+ lub –) jako znak wykładnika, a także odpowiednia litera określająca wykładnik.

Np.

```
2.3e-15
-1.4e2
+71.3e-2
```

Wewnątrz nie może być spacji. Zatem błędne są np. takie zapisy

```
2.3 e -15
1-.4e 2
+71.3 e - 2
```

✦ Jeśli wczytujemy informację dla tablicy znakowej (string) np. za pomocą następujących instrukcji

```
char tablica[80] ;
char t[80];
char *wsk = t
     cin >> tablica ;
     cin >> wsk ;
```

to nie należy zapominać, że wczytywanie zacznie się po zignorowaniu spacji poprzedzających tekst, skończy z napotkaniem pierwszego białego znaku. Innymi słowy ze zdania

```
____To jest tekst
```
(znakami _ oznaczamy tu spację)

zostanie do tablicy wczytany tylko pierwszy wyraz

```
To
```

Trzeba także pamiętać, że przy takim wczytywaniu nie jest sprawdzane zapełnienie tablicy. Zatem jeśli w danym fragmencie programu

```
char tab[6] ;
cin >> tab ;
```

na klawiaturze wystukamy wyraz o długości 100 znaków, to pierwsze znaki rzeczywiście znajdą się w tablicy `tab`, ale następne nie mieszczące się tam, będą niszczyły (zacierały) jakieś fragmenty pamięci. O tym, jak bezpiecznie wczytywać tym sposobem, porozmawiamy później (patrz: funkcja `width` oraz manipulator `setw`).

Wspomniane domniemania zapisane są wewnątrz obiektu klasy strumień, w danych składowych – tak zwanych flagach odpowiadających za format (format flags). Jeśli domniemane ustawienie tych flag nam nie odpowiada, to możemy je zmienić. Wkrótce się tego nauczymy.

Spacje oddzielające

To już nie domniemanie, to ogólna zasada. Gdy wczytujemy z klawiatury dwie liczby instrukcją

```
float x, y ;
          cin >> x >> y ;
```

i chcemy podać wartości 3.14 oraz 20.1 to nie możemy na klawiaturze wystukać

```
3.1420.1
```

Konieczna jest dla większości typów przynajmniej jedna spacja oddzielająca — pokazująca, gdzie jedna liczba się kończy, a druga zaczyna. Poprawnie powinno zatem być

```
3.14 20.1
```

Po pierwszej liczbie powinien nastąpić jakiś znak kończący jej wczytywanie. Natomiast gdyby drugą liczbą było –20.1 to moglibyśmy napisać tak

```
3.14-20.1
```

gdyż wtedy da się rozpoznać gdzie się kończy jedna liczba, a zaczyna druga.

21.6 Uwaga na priorytet

Z rozdziału o przeładowaniu operatorów pamiętasz zapewne, że co prawda operator można przeładować w dowolny sposób, ale paru rzeczy zmienić się nie da. Przykładowo nie da się zmienić jego priorytetu.

Operator * (mnożenia) będzie zawsze wykonywany przed operatorem + (dodawania). Niezależnie od tego, co robią naprawdę.

Zasada ta obowiązuje także w przypadku operatorów >> i <<. Z tabeli priorytetów (str. 72) widzimy, że priorytet operatorów >> i << jest dość niski, a więc wyrażenie

```
int a = 5 ,b = 7 ;
          cout << a + b << endl ;
```

rzeczywiście wydrukuje sumę, bez potrzeby ujmowania operacji dodawania w nawiasy

```
          cout << (a + b) << endl ;
```

Są jednak operatory o priorytecie niższym niż >> i << . Musimy wtedy posługiwać się nawiasami. Dla przykładu weźmy operator && W wyrażeniu

```
int a = 1 , b = 0 ;
          cout << (a && b) << endl ;
```

wypisane zostanie 0 – co jest rezultatem iloczynu logicznego zmiennych a i b. Gdybyśmy jednak zapomnieli o nawiasach i napisali po prostu tak:

```
          cout << a && b << endl ;
```

to kompilator wiedząc, że priorytet << jest wyższy niż priorytet && zinterpretuje to jako

```
          (cout << a) && (b << endl) ;
```

i zaprotestuje.

Pół biedy jeśli kompilator rzeczywiście zasygnalizuje błąd. Gdybyśmy jednak w naszym ostatnim przykładzie nie pragnęli wypisania końcowego znaku '\n' i napisali po prostu

```
          cout << a && b ;
```

wówczas zostanie to zrozumiane jako

```
          (cout << a ) && b ;
```

czyli, na skutek wyrażenia w nawiasie na ekran zostanie wypisana jedynka - po czym rezultat tego wyrażenia – a jest to (jak nam z poprzednich rozdziałów wiadomo) referencja do strumienia cout – zostanie użyta do wyrażenia

```
          (cout) && b ;
```

Nie jest to żaden błąd składni. Operacja logiczna && zostanie wykonana, a jej wynik pójdzie w próżnię. Najważniejsze jednak, że wypisujemy nie to, o co nam chodziło a kompilator nie sygnalizował błędu. Trzeba zatem w wypadkach wątpliwych pamiętać o umieszczeniu nawiasów.

21.7 Operatory << oraz >> definiowane przez użytkownika

Jedną z najważniejszych wad biblioteki `stdio` (znanej z klasycznego C) jest to, że o ile potrafi ona obchodzić się z typami wbudowanymi, o tyle jest bezradna wobec nowych typów zdefiniowanych przez użytkownika (wobec klas).

Tymczasem biblioteka strumieni C++ `iostream` rozwiązuje ten problem z niesamowitą łatwością. Nazywa się to szumnie przeładowaniem operatora << oraz >>, ale realizacja tego jest bardzo prosta. Mówiliśmy już o tym w rozdziale o przeładowaniu operatorów (str. 486). Tam pokazałem przeładowanie tych operatorów na użytek klasy `wektor` zastrzegając równocześnie, że do tego jeszcze wrócimy, gdyż nie jest to zrobione zbyt elegancko.

Klasa `wekt`, którą posługujemy się w tym przykładzie jest jeszcze prostsza niż tamta.

```
#include <iostream.h>
//////////////////////////////////////////////////////////
class wekt {
public :
     float x, y, z ;
} ;
/*******************************************************/
// globalne funkcje operatorowe
// realizujące przeładowania << oraz >> dla klasy wekt
/*******************************************************/
ostream& operator<<(ostream& strumien_wyj, wekt w)    //❶
{
     strumien_wyj << w.x << " " << w.y << " " << w.z ;
     return strumien_wyj ;
}
/*******************************************************/
istream& operator>>(istream& strumien_wej, wekt &w)   //❷
{
     strumien_wej >> w.x >> w.y >> w.z ;
     return strumien_wej ;
}
/*******************************************************/
main()
{
wekt a, b ;

     cout << "Podaj wspolrzedne wektora a : " ;
     cin >> a ;
     cout << "Podaj wspolrzedne wektora b : " ;
     cin >> b ;

     cout << "Wektor a ma wspolrzedne [" << a << "]"
          << endl ;
     cout << "Wektor b ma wspolrzedne [" << b << "]" ;
}
```

Po wykonaniu tego programu na ekranie pojawi się

(tłustym drukiem są zaznaczone odpowiedzi otrzymane z klawiatury).

```
Podaj wspolrzedne wektora a : 1 2 3
Podaj wspolrzedne wektora b : 7 8 64.32e+10
Wektor a ma wspolrzedne [1 2 3]
Wektor b ma wspolrzedne [7 8 6.432e+11]
```

Komentarz

❶ Przyjrzyjmy się najpierw realizacji operatora wstawiania <<. (Przypominam: wstawiania do strumienia płynącego na ekran).
Najpierw deklaracja operatora << . Typ rezultatu to oczywiście referencja strumienia, na rzecz którego operator wywołano. Zachowanie tej konwencji umożliwia wygodne kaskadowe łączenie operacji

```
cout << a << x << b ;
```

i wówczas w jednej instrukcji możemy na ekranie wypisywać typy wbudowane i typy zdefiniowane przez siebie.

Pierwszym argumentem formalnym funkcji `operator<<` jest referencja do strumienia klasy `ostream`. Nie zawsze bowiem strumieniem tym musi być obiekt o nazwie `cout`: jeśli tę funkcję wywołamy na rzecz strumienia płynącego do pliku dyskowego, to wówczas liczby zostaną zapisane na dysku.

Operator jest więc bardzo uniwersalny – służy do wstawiania informacji do jakiegokolwiek strumienia klasy `ostream`. (Nawet więcej: także do ewentualnych strumieni klas pochodnych od klasy `ostream` - pliki dyskowe obsługuje właśnie klasa pochodna od klasy `ostream`).

W naszym przykładzie referencja do obiektu klasy `ostream` brzmi: `strumien_wyj`, ale równie dobrze mogłaby brzmieć „zdzicho". Prawdę mówiąc zwykle dla skrócenia zapisu nazywa się ją jedną literą. Na razie jednak, dla oswojenia, lepiej napisać coś, co nam uprzytamnia z czym mamy do czynienia.

Drugim argumentem jest **obiekt** klasy `wekt`, który chcemy właśnie wypisać (na ekran lub dysk). Ten argument jest przysłany przez wartość, bo obiekt tej klasy jest mały, a poza tym nie zamierzamy niczego w obiekcie modyfikować. Tylko przeczytamy to, co ma być wypisane na ekran.

Gdyby jednak obiekt klasy `wekt` był bardzo duży mogłoby się opłacać przysłanie do go funkcji nie przez wartość, ale przez referencję (najlepiej referencję obiektu `const`). Oszczędza się wówczas długiego kopiowania. Przesłana zostaje wtedy referencja (przezwisko) czyli coś w rodzaju adresu – a nie cały wielki obiekt.

Ciało funkcji operatorowej to już zwykły wypis na ekran. Ponieważ składniki

```
w.x, w.y, w.z
```

są typu `float`, więc w wyrażeniu

```
strumien_wyj << wek.x
```

zostaje uruchomiona ta wersja operatora << , która pracuje z liczbami `float`. Tu już nie ma mowy o „naszym" przeładowanym operatorze. „Nasz" działa tylko wtedy, gdy po lewej stronie znaku << stoi obiekt klasy `ostream`, a po prawej obiekt klasy `wekt`

[*ostream*] << [*wekt*] ;

❷ Bardzo podobnie wygląda realizacja operatora >> wyjmowania ze strumienia (wyjmowania ze strumienia płynącego z np. klawiatury).
Typ rezultatu to tradycyjnie referencja do strumienia wejściowego, na którym pracujemy.

Pierwszy argument – to właśnie referencja do tego strumienia klasy `istream`, gdyż nie zawsze chodzi nam o konkretny obiekt o nazwie `cin` – będący strumieniem płynącym z klawiatury. Podobnie jak poprzednio – możemy także chcieć czytać coś z pliku dyskowego – wówczas do tej funkcji operatorowej przyślemy referencję do takiego konkretnego obiektu klasy `istream` (lub pochodnej).

Drugi argument to oczywiście obiekt klasy `wekt`. W wypadku tego operatora >> drugi argument musi być przysłany w sposób umożliwiający nam modyfikację oryginalnego obiektu. (Skoro chcemy do tego obiektu coś wpisywać, to jest to przecież modyfikacja). Odpada więc przesłanie przez wartość. Przesyłamy przez referencję.

Nasza funkcja operatorowa wywołana zostanie w tych miejscach w programie, gdzie po lewej stronie znaku >> stoi strumień klasy `istream` (lub pochodnej), a po prawej stronie obiekt klasy `wekt`.

[*istream*] >> [*wekt*]

W ciele naszego operatora widzimy np. zapis

```
strumien_wej >> w.x
```

Ponieważ składnik `w.x` jest typu `float`, więc jest to sytuacja

[*istream*] >> [*float*]

Zadziała więc tutaj zwykła wersja operatora >> – ta przeznaczona do wczytywania liczb `float`.

W naszej klasie `wekt` składniki `x`, `y`, `z` są `public`, więc funkcja `operator<<` (oraz `operator>>`) nie potrzebuje żadnych przywilejów, by móc na nich pracować. Gdyby jednak składniki te były niepubliczne (`private` lub `protected`), wówczas nasza funkcja `operatorowa<<` (nie będąca przecież funkcją składową klasy `wekt`) nie miałaby do nich dostępu. Jak temu zaradzić?

„–Zróbmy tę funkcję składnikiem klasy `wekt`!" – pomyślałeś pewnie.

Nie, nie da się. Jeśli chcesz by pierwszym argumentem funkcji operatorowej << był obiekt klasy `ostream`, to funkcja nie może być składnikiem klasy `wekt`. Może być tylko funkcją składową klasy `ostream` albo funkcją globalną. Jeśli zrobilibyśmy ją funkcją składową klasy `wekt`, to pierwszym argumentem musiałby być obiekt klasy `wekt`. To nas nie urządza, bo przyzwyczailiśmy się do zapisu

```
cout << w ;
```

i nie zamierzamy go zmieniać na

```
w << cout ;                    // !!!
```

Krótko mówiąc był to zły pomysł. Nadal nie mamy rozwiązanego problemu jak funkcja operatorowa ma mieć dostęp do prywatnych składników klasy.

Jest jednak wyjście. Klasa może nadać funkcji operatorowej przywilej dostępu do swych składników niepublicznych. Robi to oczywiście za pomocą deklaracji przyjaźni. Oto taka zmodyfikowana definicja klasy:

```
class wekt {
     float x, y, z ;

     friend ostream & operator<<(ostream &, wekt) ;
     friend istream & operator>>(istream &, wekt &) ;
} ;
```

Poza tym w naszym programie nic się nie zmienia.

Bardzo mało ważne przypomnienie

Mówiłem kiedyś, że definicję zaprzyjaźnionej z klasą funkcji można napisać wewnątrz deklaracji klasy. Zatem nasza ostatnia klasa mogłaby wyglądać tak:

```
class wekt {
     float x, y, z ;
     //------
     friend ostream& operator<<(ostream& strumien_wyj, wekt w)
     {
          strumien_wyj << w.x << " " << w.y << " " << w.z;
          return strumien_wyj ;
     }
     //------
     friend istream& operator>>(istream& strumien_wej, wekt &w)
     {
          strumien_wej >> w.x >> w.y >> w.z ;
          return strumien_wej ;
     }
} ;
```

Co to zmienia? Przypomnę z rozdziału o przyjaźni:

1) Co prawda funkcje są nadal tylko przyjaciółmi, a nie funkcjami składowymi ale: są teraz kompilowane jako `inline`.

2) Funkcje leżą teraz w zakresie **leksykalnym** klasy `wekt`. Oznacza to, że:

   - gdyby wewnątrz klasy była w mocy instrukcja typedef - nasi przyjaciele mogliby w swoich ciałach korzystać z następstw tego faktu (czyli z tak zdefiniowanych synonimów do istniejących typów),
   - mogliby też skorzystać z ewentualnych typów wyliczeniowych `enum` zdefiniowanych we wnętrzu tej klasy.

Tak się składa, że u nas nie ma niczego takiego, ale koniecznie chciałem o tym wspomnieć teraz, kiedy już jesteś ekspertem od operatorów.

Symetria

Nasze operatory są bardzo proste, wczytują po prostu 3 liczby typu `float`. Oczywiście można by je rozbudować, np. operator << wypisywania mógłby między te liczby wstawić przecinki.

Do dobrego stylu programowania należy, by operatory były „symetryczne". To znaczy, żeby wczytywanie miało taką samą formę jak wypisywane. O ile w wypadku klawiatury i ekranu ma to mniejsze znaczenie (estetyczne, lub ewentualnie – przyzwyczajenia), o tyle w wypadku pracy z plikiem dyskowym jest to bardzo ważne. Jeśli operator wstawiania (do strumienia) zapisał coś na dysku, to operator wyjmowania (ze strumienia) powinien umieć to za chwilę przeczytać.

Konkretnie: jeśli operator wstawiania << między tymi liczbami dodawał przecinki, to operator wyjmowania czytając ten zapis powinien umieć te przecinki zinterpretować. Jeśli umie on czytać tylko bez przecinków, to przy wczytywaniu drugiej liczby (współrzędna y) nastąpi błąd: zamiast spodziewanych cyfr napotkany został jakiś niezrozumiały przecinek.

Oto przykładowa realizacja operatorów w sytuacji, gdy używamy przecinków:

```
ostream& operator<<(ostream& strumien_wyj, const wekt & w)
{
     strumien_wyj << w.x << ", " << w.y << ", " << w.z ;
     return strumien_wyj ;
}
/*****************************************************/
istream & operator>>(istream & strumien_wej , wekt &w)
{
char znak ;
     strumien_wej >> w.x >> znak >> w.y >> znak >> w.z ;
     return strumien_wej ;
}
```

Teraz instrukcja wczytywania współrzędnych wektora jest zdolna przyjąć zapis

```
3.14, 7, 512e6
```

(a także zapis

```
4.14    ,    7       ,512e6
```

bo spacje przed i po przecinku są ignorowane)

21.7.1 Operatorów wstawiania i wyjmowania ze strumienia - nie dziedziczy się

Z faktu, że przeładowanie operatora wstawiania (<<) i wyjmowania (>>) na rzecz obiektu klasy `K` nie może być zrealizowane jako funkcja składowa klasy `K`, wynika ważna konsekwencja:

> W razie, gdy tworzymy klasę pochodną od klasy `K`, operatory te nie są dziedziczone. Dziedziczy się przecież tylko składniki klasy, a operatory wstawiania i wyjmowania – funkcjami składowymi nie są.

Nawet jeśli są przyjaciółmi klasy K, to także nic nie daje, bo przyjaźni też się nie dziedziczy.

Dla nowej klasy pochodnej należy więc zdefiniować nowe operatory.

> *Ma to swój sens – skoro klasa pochodna zawiera zwykle coś więcej niż klasa podstawowa – to przecież trzeba to „coś więcej" uwzględnić przy wczytywaniu czy wypisywaniu obiektu nowej klasy.*

Definiowanie takiego nowego operatora nie musi jednak polegać na pisaniu całego operatora „od zera". Z wnętrza nowego operatora można wywołać operator z klasy podstawowej.

Oto klasa pochodna:

```
class wektor_z_opisem : public wekt {
public:
     char opis[30] ;
};
```

Jest ona, jak widać, wzbogacona o nowy składnik. Nowe operatory wstawiania i wyjmowania ze strumienia możemy zdefiniować następująco: (zakładam, że mamy zdefiniowane operatory dla klasy `wekt` tak, jak w naszym ostatnim programie – czyli te bez przecinków)

```
ostream & operator<<(ostream &strumien_wyj , wektor_z_opisem &w)
{
     strumien_wyj << (wekt&)w                          // ❶

     strumien_wyj << " " << w.opis ;                   // ❷
     return strumien_wyj ;
}
/*****************************************************/
istream & operator>>(istream & strumien_wej ,
wektor_z_opisem &w)
{
     strumien_wej >> (wekt&)w >> w.opis ;
     return strumien_wej ;
}
```

Jak widać, w ciałach obu funkcji – w obu wypadkach posłużyliśmy się wywołaniem operatorów wstawiania i wyjmowania zdefiniowanymi dla obiektów klasy podstawowej. Stosując rzutowanie ❶ – polegające na konwersji referencji obiektu klasy pochodnej na referencję obiektu klasy podstawowej – sprawiamy, że na widok tego symbolu << w tej linijce ruszy do pracy funkcja operatorowa dla klasy podstawowej `wekt`. Tym samym wstawione zostaną do strumienia te składniki, które odziedziczyliśmy od klasy `wekt`. Pozostaje nam już tylko wstawić tam jeszcze dodatkowy składnik `opis`. Jest on typu wbudowanego `char[]`, zatem w linijce ❷ pracuje już operator wstawiania dla typów wbudowanych.

Przy operatorze wyjmowania ze strumienia (wczytywania) – jest podobnie.

21.7.2 Operatory wstawiania i wyjmowania nie mogą być wirtualne. Niestety.

Z faktu, że operatory wstawiania do strumienia i wyjmowania z niego nie są składnikami klasy, wynika też konsekwencja, że nie mogą być funkcjami wirtualnymi. Pamiętamy przecież, że funkcjami wirtualnymi („funkcjami inteligentnie dziedziczonymi") mogą być tylko funkcje składowe jakiejś klasy.

Szkoda. Dobrze by było powiedzieć: „–Widzisz ten samochód? No to wypisz o nim wszystkie informacje". A na to ruszałby do pracy operator << od Mercedesa (jeśli wskaźnik pokazywałby na Mercedesa).

Mimo wszystko da się to jakoś zorganizować. Poniżej pokażę jak, ale jeśli czytasz tę książkę po raz pierwszy, to radzę nie zaprzątać sobie teraz tym głowy i przejść do następnego paragrafu o sterowaniu formatem.

Jak sobie zatem tę wirtualność zorganizować? Ogólna zasada jest taka:

definiujemy funkcję wirtualną w klasie podstawowej i pochodnej. Tę funkcję wywołujemy z operatora << (lub z operatora >>).

```
#include <iostream.h>
//////////////////////////////////////////////////////////
class samochod {
public:
     int rok_produkcji ;
     virtual void rzecznik (ostream & strum);
} ;
//////////////////////////////////////////////////////////
class mercedes : public samochod {
public:
     char *model ;
     void rzecznik(ostream & strum) ;
} ;
/*******************************************************/
// realizacja operatora << dla klasy podstawowej
/*******************************************************/
ostream & operator<<(ostream &strum , samochod &x)   // ❶
{
     x.rzecznik(strum) ;                              // ❷
     return strum ;
}
/*******************************************************/
// realizacje funkcji wirtualnych
/*******************************************************/
void samochod::rzecznik(ostream & strum)
{
     strum << rok_produkcji ;
}
/*******************************************************/
void mercedes::rzecznik(ostream & strum)
{
     samochod::rzecznik(strum);
     strum << " " << model ;
```

```
}
/*****************************************************/
main()
{
int i ;
samochod a, b ;

          a.rok_produkcji = 1990 ;
          b.rok_produkcji = 1992 ;

mercedes m ;

          m.rok_produkcji = 1991;
          m.model = "sportowy" ;

     cout << a << endl ;
     cout << b << endl ;
     cout << m << endl ;
}
```

Po wykonaniu tego programu na ekranie zobaczymy

```
1990
1992
1991 sportowy
```

Kilka uwag

W programie widzimy zdefiniowany operator wstawiania do strumienia << dla klasy podstawowej ❶. Dla klasy pochodnej tego operatora nie ma. Co ciekawe: operator ten nie zajmuje się wypisaniem na ekran. Chwyt polega na tym, że on zamiast wypisać coś na ekran, wywołuje funkcję `rzecznik`, która jest funkcją składową wirtualną. To ona będzie wypisywała. Rzecznik wywoływany jest dla referencji obiektu, a więc mechanizm wirtualności tu się właśnie objawi.
Konkretnie: wewnątrz funkcji operatorowej << w miejscu ❷ uruchomiony zostanie właściwy rzecznik.
Albo `samochod::rzecznik` albo `mercedes::rzecznik` –zależnie od tego, czy pod przezwiskiem `x` rozpoznany obiekt klasy `samochod` czy klasy `mercedes`.

21.8 Sterowanie formatem

Jak powiedzieliśmy, przy operacjach na strumieniu używane są pewne domniemania dotyczące formatu informacji wstawianej lub wyjmowanej ze strumienia. Jeśli te domniemania nam nie odpowiadają, to możemy je zmienić i od tej pory dany strumień będzie wczytywał lub wypisywał według nowych zasad (według nowego formatu).

21.9 Flagi stanu formatowania

Bieżące zasady formatowania zapisane są w ośrodku dowodzenia konkretnego strumienia – w tak zwanych flagach stanu formatowania (format state flags). Flagi te są zwykle poszczególnymi bitami w słowie `long`.

Przykład -

> *przez domniemanie typy całkowite wypisywane są w systemie dziesiątkowym. To domniemanie wynika właśnie z ustawienia jednej z flag. Jeśli nam to nie odpowiada, możemy zmieniając flagę stanu formatowania sprawić, że liczby będą wypisywane szesnastkowo (heksadecymalnie).*

Ponieważ flagi stanu formatowania są potrzebne strumieniom klasy `istream`, a także strumieniom klasy `ostream`, a także jeszcze paru innym, dlatego sprawy te zebrano i umieszczono w jednej klasie, która jest podstawową dla tych pozostałych. Dzięki takiemu dziedziczeniu autorzy biblioteki oszczędzili sobie wiele pracy.†)

Klasa, w której te flagi stanu formatowania umieszczono, jest nazwana `ios` (można sobie tę nazwę rozwinąć jako: Input Output State). W klasie tej zdefiniowany jest też (publiczny!) typ wyliczeniowy, który ułatwia nam pracę na tych flagach.

```
// ...
public:
enum {
     skipws     = 0x0001,      // ignoruj białe znaki
     left       = 0x0002,      //              lewe
     right      = 0x0004,      // justowanie prawe
     internal   = 0x0008,      //              "wewnętrzne"
     dec        = 0x0010,      //          decymalna
     oct        = 0x0020,      // konwersja oktalna
     hex        = 0x0040,      //          hexadecymalna
     showbase   = 0x0080,      // pokaż podstawę konwersji
     showpoint  = 0x0100,      // pokaż kropke dziesiętną
     uppercase  = 0x0200,      // wielkie litery (w liczbach)
     showpos    = 0x0400,      // znak + w liczbach dodatnich
     scientific= 0x0800,       // notacja     : "naukowa"
     fixed      = 0x1000,      //             : "zwykła"
     unitbuf    = 0x2000,      // buforowanie ....
     stdio      = 0x4000       // .....współpraca z stdio
} ;
```

Skoro flagi te są zdefiniowane w zakresie klasy `ios` i są tam publiczne, to można ich używać spoza tego zakresu – poprzedzając je kwalifikatorem zakresu, np.:

```
ios::left
ios::scientific
```

†) Jest to dobry przykład na wielokrotne użycie raz zdefiniowanego kodu (reusability).

Zanim przystąpię do omówienia tych flag, chciałbym ostrzec byś się nie przerażał. Mimo, iż początkowo wydawać się będzie, że to ogromna ilość informacji do zapamiętania – to jednak wkrótce poznamy wygodne narzędzia do sprawnego ustawiania stanu formatowania. Cierpliwości.

21.9.1 Znaczenie poszczególnych flag sterowania formatem

`skipws` – (skip white spaces: przeskakuj białe znaki)

Ustawienie tej flagi jest sugestią by w procesie wyjmowania ze strumienia – ignorowane były ewentualne białe znaki (spacje, tabulatory, znaki nowej linii itp.), które poprzedzają oczekiwane znaki.

Przykładowo: jeśli oczekujemy, że strumieniem (choćby z klawiatury) przypłyną do nas jakieś znaki reprezentujące liczbę – to niezależnie od tego, czy na klawiaturze zostaną wystukane znaki

```
247
```

czy też

[spacja][tabulator][spacja]`247`

to efekt będzie ten sam, bo białe znaki przed liczbą zostaną w trakcie wyjmowania ze strumienia zignorowane – i otrzymamy po prostu liczbę.

Jeśli flaga ta nie jest ustawiona, wówczas napotkanie białego znaku w momencie, gdy spodziewana jest liczba – uznane zostaje za błąd. Przez domniemanie ta flaga jest ustawiona (białe znaki są więc przeskakiwane).

`left`
`right`
`internal`

Te trzy flagi nazywane są **polem justowania** `ios::adjustment`

Justowanie polega tu na tym, że jeśli np. liczba składająca się z 2 cyfr ma zostać wypisana za pomocą 10 znaków, to może ona być wypisana tak:

```
-42________        left
________-42        right
-________42        internal
```

Czyli może być dosunięta do lewej krawędzi tego obszaru (`left`) , albo do prawej krawędzi tego obszaru (`right`) albo wypisana tak, że ewentualny znak dosunięty jest do lewej krawędzi tego obszaru, a liczba do prawej. Wewnątrz (`internal`) są znaki wypełniające - najczęściej spacje.

Jest chyba oczywiste, że w danej chwili może być ustawiona tylko jedna z tych flag na polu justowania. Albo decydujemy się na jeden sposób wypisania, albo na drugi, albo na trzeci.

Przez domniemanie ustawiona jest flaga `right`.

`dec`
`oct`
`hex`

Te trzy flagi nazywane są wspólnie **polem podstawy konwersji `ios::basefield`**. Ponieważ „pole podstawy" kojarzy się trochę z geometrią, może zręczniej byłoby powiedzieć: „polem odpowiedzialnym za podstawę konwersji". Decyduje ono w jakim systemie (zapisie) wczytywane lub wypisywane są liczby całkowite.

Podstawą konwersji liczb całkowitych może być liczba:

`10` - konwersja dziesiątkowa (decymalna)

`8` - konwersja ósemkowa (oktalna)

`16` - konwersja szesnastkowa (heksadecymalna)

W danej chwili tylko jedna z tych flag może być ustawiona. Jeśli nie jest ustawiona żadna, wówczas

- wypisywanie liczb odbywa się w notacji dziesiątkowej,
- wczytywanie liczb odbywa się według obowiązujących w C++ reguł dla notacji stałych dosłownych.

Mówiliśmy o tym w rozdziale o typach (str. 35), tutaj przypomnę tylko, że są to zasady w stylu: jeśli liczbę poprzedza zero – to znaczy, że będzie to liczba w systemie ósemkowym.

`showbase` – pokaż podstawę†)

Jest to flaga, której ustawienie jest jakby żądaniem by liczby wypisywane były tak, żeby łatwo można było rozpoznać w jakim są systemie.

Zatem przed liczbą szesnastkową staną znaki `0x`, przed liczbą ósemkową `0`, przed liczbą dziesiątkową nic.

Poniżej pokazuję wypis różnych liczb przy ustawionej i skasowanej fladze `showbase`

```
            flaga ios::showbase
            ustawiona           nie ustawiona

hex         0xa4c               a4c
oct         077                 77
dec         32                  32
```

Przez domniemanie flaga ta jest nie ustawiona.

`showpos` (show positive: pokaż dodatnie)

ustawienie tej flagi powoduje, że przy wypisywaniu dodatnich liczb dziesiątkowych zostaną one poprzedzone znakiem `+ (plus)`.

†) (czytaj: „szoł bejs")

```
flaga ios::showpos
ustawiona          nie ustawiona

+107.2             107.2
```

Przez domniemanie flaga ta jest nie ustawiona

`uppercase` (wielkie litery)

W zapisie niektórych typów liczb występują litery oznaczające np. podstawę konwersji: 'x' lub wykładnik: 'e' w notacji naukowej. Ustawienie tej flagi sprawia, że te litery są wypisywane jako wielkie: `X` lub `E`.

```
flaga ios::uppercase
ustawiona          nie ustawiona

0X1A               0x1a
33E-5              33e-5
```

Przez domniemanie flaga ta jest nie ustawiona.

`showpoint` (pokaż kropkę dziesiętną)

Przy wypisywaniu liczb zmiennoprzecinkowych flaga ta powoduje, że wypisywane są nawet nieznaczące zera i kropka dziesiętna. Jeśli na przykład obowiązuje dokładność wypisywania do 6 miejsca po kropce dziesiętnej to liczby wypisywane są w taki sposób

```
flaga ios::showpoint
nie ustawiona      ustawiona

7.14               7.140000
4                  4.000000
```

Przez domniemanie flaga ta jest nie ustawiona.

`scientific` (notacja „naukowa")
`fixed` (notacja zwykła)

flagi te nazywane są **polem odpowiedzialnym za notację liczby zmiennoprzecinkowych `ios::floatfield`**

Ustawienie flagi `scientific`†) sprawia, że liczby będą wypisywane w tak zwanej notacji naukowej (czyli wykładniczej).

Ustawienie flagi `fixed` sprawia, że liczby będą wypisywane w zwykłej notacji. W notacji zwykłej czyli „dziesiętnej"—nie: dziesiątkowej ale: dziesiętnej.

> *Najmłodszym czytelnikom nieśmiało przypominam, że liczby rzeczywiste możemy zapisywać w postaci ułamkowej (½) dziesiętnej (0.25) lub wykładniczej (2.5e-1).*

Aby jednak uniknąć ewentualnych konfuzji – mówmy: „notacja zwykła".

Oto ta sama liczba w obu zapisach:

---

†) ang.: naukowy, (czytaj: „saientyfik")

```
scientific          fixed
1.7e3               1700
```

Jeśli żadna z flag tego pola `ios::floatfield` nie jest ustawiona, wówczas zastosowany przy wypisywaniu liczby sposób zależeć będzie od samej liczby. Mianowicie: jeśli wykładnik będzie mniejszy niż -4 lub większy od obowiązującej dokładności (domniemanie = 6), to użyta zostanie notacja naukowa.

```
0.02      0.002     0.0002,   2e-5,     ...

2000,     20000,    200000,   2000000,  2e7, 2e8, ...
```

`unitbuf`

ustawienie tej flagi jest rezygnacją z tak zwanego buforowania strumienia. Strumień niebuforowany nie jest tak efektywny, jak buforowany. Dlatego przez domniemanie flaga ta jest ustawiona.

`stdio`

ustawienie tej flagi jest żądaniem, by po każdym wstawieniu czegoś do strumienia `stdio` i `stderr` – strumień popłynął od razu i informacja bez zwłoki pojawiła się na ekranie.

Są to zagadnienia współpracy starej biblioteki `stdio`, z nową - o tym będziemy jeszcze osobno rozmawiali.

Przyjrzyj się bliżej naszemu typowi wyliczeniowemu, a szczególnie wartościom liczbowym przypisanym poszczególnym flagom. Jeśli masz odrobinę wprawy w posługiwaniu się liczbami w zapisie szesnastkowym- to zapewne od razu odczytasz, że w słowie stanu formatowania (jak wiemy – typu `long`) na przykład flaga `skipws` to najmniej znaczący bit, a z kolei bit piąty reprezentuje flagę `dec`.

Jednak taka wiedza nie jest Ci zupełnie potrzebna. Są bowiem wygodne narzędzia do ustawiania flag formatowania, bez myślenia o ich pozycjach.

21.10 Sposoby zmiany trybu (reguł) formatowania

Wspomniałem już, że stanem formatowania zajmuje się klasa `ios`, którą wiele klas później dziedziczy. Dziedziczą ją też klasy `ostream` oraz `istream` — a właśnie obiektami tych klas są znane nam od dawna strumienie `cout` oraz `cin`. Jeśli chcemy zmienić format wypisywania informacji na ekranie przez strumień `cout`, lub format wczytywania informacji z klawiatury strumieniem `cin` – to musimy poznać, jakimi dysponujemy narzędziami.

Oto jak w przybliżeniu wygląda klasa `ios`:

```
class ios {
     long flagi_stanu_formatowania ;
public:
                    // ----------------- funkcje składowe
```

```
        long setf(long , long);
        long setf(long) ;
        long unsetf(long) ;
                    // ---------------- wygodniejsze
        int  width(int);
        int  precision(int) ;
                    // ............... i wiele dalszych
} ;
```

Jak widać, w klasie są też jakieś funkcje składowe. Za ich pomocą możemy, mniej lub bardziej wygodnie, posługiwać się flagami.

Zanim przystąpimy do omówienia tych funkcji muszę zastrzec, że jest kilka sposobów zrobienia tego samego. Po kolei więc omówimy sposoby modyfikacji flag stanu formatowania za pomocą:

a) bardzo elementarnych funkcji składowych klasy `ios`. Funkcje te, to `setf`, `unsetf`. Osobiście bardzo nie lubię tego sposobu, bo wymaga on bym pamiętał nazwy wszystkich omówionych w poprzednim paragrafie flag.

b) funkcji składowych z klasy `ios`, ale takich, których nazwy same przypominają to, co robią. (Taka funkcja nie jest uniwersalna, służy tylko do modyfikacji jednej określonej flagi, ale w gruncie rzeczy sposób ten wydaje mi się wygodniejszy od sposobu a),

c) manipulatorów – to bardzo ciekawa rzecz. Zamiast wywoływać funkcję składową dla danego strumienia, wpuszczamy do niego — jak zatrutą wodę — specjalne kody, które strumień zinterpretuje jako życzenie zmiany sposobu formatowania. Ten sposób wydaje mi się najwygodniejszy (choć nie całkiem uniwersalny).

W poprzednim paragrafie mówiliśmy o klasie `ios` i jej funkcjach składowych, które trzeba uruchomić. Jak uruchomić funkcję składową z klasy `ios`, o której prawie nic nie wiemy?

Wiemy jednak najważniejsze – klasa ta jest klasą podstawową dla innych klas strumieni, którymi się posługujemy. Pamiętasz zapewne z rozdziałów o dziedziczeniu, że w klasie pochodnej dostęp do publicznych składników klasy podstawowej jest taki sam, jakby były one składnikami klasy pochodnej. Oznacza to, że funkcje te możemy wywołać tak samo, jakby były funkcjami składowymi klasy pochodnej.

Oto jedna z naszych klas pochodnych: `ostream`. Jest to klasa opisująca strumienie wyjściowe. Jednym z obiektów tej klasy jest `cout`. Jego definicja w programie wyglądać by mogła tak:

```
#include <iostream.h>
ostream cout ;                    // def obiektu klasy ostream
```

Nie umieszczamy jednak takiej definicji w programie, gdyż zrobił to za nas kompilator. Mówiliśmy już, iż kompilator domyślając się, że programista będzie chciał coś wypisywać na ekranie – predefiniuje kilka strumieni. Czyli definiuje je za nas, by były od razy zdefiniowane. Zatem `cout` jest konkretnym obiektem klasy `ostream`. Wróćmy do naszego problemu – jak wywołać funkcję składową z klasy `ios`. Ależ to bardzo proste! Do tego służy zapis

obiekt.funkcja()

czyli u nas

`cout.`*funkcja()*

Wiemy już jak uruchomić funkcję. Teraz porozmawiamy o tym, jakie mamy funkcje i co one mogą dla nas zrobić.

21.10.1 Zmiana sposobu formatowania funkcjami `setf`, `unsetf`

Jeśli chcemy zmienić jakąś flagę formatowania, wówczas możemy to zrobić posługując się na przykład takimi funkcjami składowymi klasy `ios`:

```
long setf(long ktore_flagi) ;
long unsetf(long ktore_flagi);
```

Pierwsza z tych funkcji umożliwia ustawienie w słowie stanu tych flag, które przysłaliśmy we wzorze jako argument. Funkcja, jako rezultat, zwraca dotychczasową wartość słowa stanu formatowania.

Np. instrukcja

```
cin.setf(ios::skipws) ;
```

ustawi flagę `skipws` odpowiadającą za ignorowanie białych znaków przez strumień wejściowy `cin`. Inne flagi pozostaną bez zmian. (Strumień jest wejściowy – bo, jak pamiętamy, ta flaga odnosi się tylko do wczytywania informacji – wyjmowania ze strumienia).

Aby skasować tę flagę używamy funkcji

```
cin.unsetf(ios::skipws) ;
```

Ustawianie flag tworzących pole

Jak już wiemy, są takie flagi, które występują w grupie zwanej polem. Jeśli chodzi o konwersję liczb, to określają ją trzy flagi

```
ios::dec
ios::hex
ios::oct
```

Tę grupę flag nazywamy polem odpowiadającym za podstawę konwersji i nazywamy `ios::basefield`.

Jeśli chcemy, by na przykład strumień wyjściowy, który wypisuje w zapisie dziesiątkowym – wypisywał od tej pory liczby w zapisie szesnastkowym - to musimy nie tylko ustawić flagę `ios::hex`, ale także skasować flagę `ios::dec`.

Można to zrobić tak

```
cout.setf(ios::hex);          // ustawienie
cout.unsetf(ios::dec) ;       // skasowanie
```

Nie jest to wygodne, dlatego istnieje funkcja składowa

```
long setf(long flaga, long nazwa_pola) ;
```

Argumentami są tu: flaga, którą chcemy ustawić, oraz nazwa pola, do którego ona należy. Takie wywołanie funkcji spowoduje nie tylko ustawienie żądanej flagi, ale także wyzerowanie pozostałych na tym polu.

Oto jak wygląda wywołanie zamieniające jedną instrukcją podstawę konwersji w strumieniu `cout`

```
cout.setf(ios::hex, ios::basefield) ;
```

Poniżej zamieszczam przykład, w którym korzystamy z właśnie omówionych funkcji, a także dodatkowo z nowej funkcji `flags` będącej jeszcze jedną funkcją składową klasy `ios`.

```
#include <iostream.h>
/*****************************************************/
main()
{
float x = 1175 ;
long stare_flagi ;

     cout << x << endl;

     cout <<"Zapamietanie flag formatowania\n" ;
     stare_flagi = cout.flags();                    // ❷

     cout.setf(ios::showpoint) ;
     cout << x << endl ;

     cout.setf(ios::scientific, ios::floatfield);
     cout << x << endl ;

     cout.setf(ios::uppercase);
     cout << x << endl ;

     cout.unsetf(ios::showpoint) ;
     cout << x << endl ;

     cout <<"Powrót do starego sposobu formatowania\n" ;
     cout.flags(stare_flagi);                       // ❶
     cout << x << endl ;

}
```

❒ **Po wykonaniu tego fragmentu programu na ekranie zobaczymy**

```
1175
Zapamietanie flag formatowania
1175.000000
1.175000e+03
1.175000E+03
1.175E+03
Powrót do starego sposobu formatowania
1175
```

Nowości

❶ W programie widzimy funkcję

```
long flags(long wzor) ;
```

Jest to funkcja składowa klasy `ios`, która pozwala ustawić wszystkie flagi stanu formatowania według podanego wzoru. Funkcja ta ustawia te flagi, które mają być ustawione, a kasuje (zeruje) te, które mają być wyzerowane. Dla porównania przypomnę, że funkcja `setf` mogła jedynie ustawić wyszczególnione flagi — wyzerować nic nie mogła. Z kolei funkcja `unsetf` potrafiła jedynie wyzerować wyszczególnione flagi — ustawić niczego nie mogła.

Funkcja jako rezultat zwraca słowo `long` obrazujące dotychczasowy stan flag formatowania.

❷ Inny wariant funkcji `flags`

```
long flags(void) ;
```

Niczego nie ustawia, a tylko „dowiaduje się" o bieżące ustawienia flag. Ponieważ w naszym programie zamierzamy zmieniać dużo flag, więc zapamiętanie początkowych ustawień flag pozwoli nam potem ❶ wrócić do starych ustawień za pomocą jednej instrukcji.

21.10.2 Wygodniejsze funkcje do zmiany stanu formatowania.

Poznaliśmy więc sposoby zmian flag stanu formatowania. Zwracam uwagę, że są to *flagi* – zatem jest w nich zapisana informacja typu logicznego: tak/nie (prawda/fałsz). (Np. pokazywać kropkę dziesiętną: tak czy nie?).

Oczywiście nie wszystko da się określić tym sposobem. Nie można tak określić chociażby informacji o tym, ile miejsc po przecinku liczby zmiennoprzecinkowej ma być wypisywane na ekranie. I na to są jednak sposoby.

W klasie `ios` oprócz poznanych w poprzednim paragrafie funkcji `setf`, `unsetf`, `flags`, są jeszcze inne funkcje składowe.

```
int width(int) ;
int width();

int precision(int);
int precision();

int fill(int);
int fill();
```

Funkcja `width`

`width`[+)] – jest funkcją określającą na ilu (co najmniej) znakach należy wypisać daną liczbę. Domniemana wartość to zero, co oznacza, że liczba zostanie wypisana za pomocą tylu znaków, ile jest wymagane, by ją wypisać w całości. Jako rezultat funkcja ta zwraca dotychczasową wartość szerokości.

Dla przykładu: liczba 107 wymaga trzech znaków: 1, 0 i 7. Stosując funkcję width możemy zmienić sposób jej wypisania

```
int liczba = 107 ;
cout << "Wypis:" << liczba << "\nWypis:" ;
cout.width(7) ;
cout << liczba << endl ;
```

Na ekranie pojawi się to jako

```
Wypis:107
Wypis:    107
```

Funkcja ta przydaje się głównie wtedy, gdy chcemy wypisywać liczby w kolumnach –jakby w postaci tabeli.

```
long kwota[] = { 120, 1650000 , 5200 , 190000123};

for(int i = 0 ; i < 4 ; i++)
{
     cout << "Rachunek nr " ;
     cout.width(2);
     cout << i << " opiewa na sume :" ;
     cout.width(8) ;
     cout << kwota[i] << " DM\n" ;
}
```

Na ekranie zobaczymy

```
Rachunek nr 0 opiewa na sume :     120 DM
Rachunek nr 1 opiewa na sume : 1650000 DM
Rachunek nr 2 opiewa na sume :    5200 DM
Rachunek nr 3 opiewa na sume :190000123 DM
```

W ostatniej linijce widać, że mimo wszystko harmonia się rozburzyła. To dlatego, że określiliśmy szerokość (width) jako 8 znaków. Co najmniej 8 znaków. Jeśli liczba wymaga do swojego wypisania więcej niż te 8 znaków, wówczas zostanie wypisana na tylu znakach, ile potrzeba. W efekcie jakby rozepchała szerokość kolumny niszcząc ładny porządek. Sami jesteśmy sobie winni. Należało przewidzieć szersze kolumny – na przykład

```
cout.width(12);
```

> Nie ma sposobu określenia maksymalnej liczby znaków, na których wyprowadzane mają być liczby.

To dlatego, że gdyby liczba okazała się za długa wymagałoby to odcięcia kawałka tej liczby (!) albo (jak w FORTRAN'ie) wyprowadzenia w tym miejscu zamiast liczby – jakichś innych znaków, np. gwiazdek. Jedno i drugie jest nie do przyjęcia. Lepiej mieć choćby rozburzoną kolumnę, ale za to zawierającą prawdziwą informację.

---

††) width – ang.: szerokość (czytaj: „łydf")

Ustawienie szerokości nie odbywa się raz na zawsze. Dotyczy tylko najbliższej operacji we/wy. Potem automatycznie zaczyna obowiązywać domniemanie, czyli szerokość 0. Jeśli zatem chcemy wypisać kilka kolumn obok siebie, musimy przed wypisaniem każdej z nich wykonać funkcję `width`.

Masz rację, nie jest to zbyt wygodne. [Niebawem, poznamy wygodniejszy sposób robienia tego samego. Nazywa się on manipulatorem `setw` (od set width – ustaw szerokość)].

Istnieje także funkcja `width` wywoływana z pustą listą argumentów. Funkcja ta pozwala nam dowiedzieć się o właśnie obowiązującą szerokość.

```
int stara = cout.width() ;
```

Uwaga:

Jeśli chodzi o *wczytywanie* informacji (wyjmowanie ze strumienia) to ustawienie szerokości funkcją `width` nie ma wpływu na wczytywanie liczby. Powtarzam: liczby! Żądanie określonej szerokości objawi się natomiast przy wczytywaniu stringu. Dzięki temu możemy zapobiec przekroczeniu rozmiaru tablicy przeznaczonej do przyjęcia znaków.

```
char napis[7] ;

     cin.width(sizeof(napis) ) ;
     cin >> napis ;
```

Z klawiatury przyjętych zostanie co najwyżej tyle znaków stringu, by string ten mógł się zmieścić w tablicy `napis`.

Zagadka: To znaczy ile maksymalnie znaków zostanie przyjętych?

Odpowiedź: Wcale nie 7 tylko 6. Na końcu stringu musi być bowiem także znak NULL, który zostaje tam zawsze automatycznie dodawany.

Należy jednak uważać. Jeśli na tablicę pokazujemy wskaźnikiem

```
char *wskaznik = napis ;
```

to instrukcje

```
cin.width(sizeof(wskaznik) ) ;
cin >> napis ;
```

Wcale nie spowodują wczytania maksymalnie 6 znaków. To dlatego, że wyrażenie `sizeof(wskaznik)` *oblicza rozmiar wskaźnika, a nie tablicy, na którą on pokazuje. Jeśli w Twoim komputerze wskaźnik do* `char` *ma rozmiar 4 bajty, to wczytane zostaną maksymalnie 3 znaki, a do nich dodany znak NULL. Pozostałe elementy tablicy* `napis` *zostaną niewykorzystane.*

Funkcja `fill`

Fill znaczy po angielsku – wypełnij. Jeśli decydujemy (za pomocą funkcji `width`), że dana liczba ma być wyprowadzana na 7 miejscach, a tymczasem właśnie ma ona szerokość tylko 2 miejsc, to pozostałe 5 zostają wypełnione spacjami. Nie zawsze muszą to jednak być spacje. Za pomocą funkcji

```
char fill(char);
```

możemy zdecydować o innym znaku wypełniającym. Funkcja ta jako argument przyjmuje właśnie żądany znak wypełniający. Jako rezultat zwraca znak wypełniający, który obowiązywał dotychczas. Inna wersja tej funkcji

```
char fill() ;
```

pozwala się dowiedzieć jaki znak obowiązuje obecnie, bez zmieniania go. Przykładem zastosowania funkcji `fill` może być sytuacja, gdy na rachunku wypisujemy liczbę i chcemy mieć pewność, że nikt jej nie sfałszuje dopisując coś z przodu.

```
int saldo = 2573 ;

cout << "stan konta : " ;
cout.width(9);
cout.fill('*');

cout << saldo << "$\n" ;
```

Na ekranie zobaczymy

```
stan konta : *****2573$
```

W przeciwieństwie do poprzedniej funkcji `width` – funkcja `fill` daje efekt trwały, to znaczy w naszym wypadku gwiazdki będą obowiązującym znakiem wypełniającym – dopóki tego nie zmienimy powtarzając wywołanie funkcji `fill`.

Na zakończenie warto dodać, że ten sam efekt, co wywołanie funkcji `fill`, można uzyskać (nieco wygodniej) stosując tzw. manipulator `setfill`, który poznamy niebawem.

Funkcja `precision`

Precision – ang. dokładność. †) Jest to funkcja, która pozwala nam określić dokładność wypisywania liczb zmiennoprzecinkowych. Przez domniemanie, liczby takie wypisywane są z dokładnością do 6 miejsca po kropce dziesiętnej. Chyba, że byłyby to nieznaczące zera, których nie chcemy (flaga `ios::showpoint`), to wtedy krócej.

Oto deklaracje tych funkcji w klasie `ios`:

†) (czytaj: „presyżyn")

```
int precision(int);
int precision();
```

Pierwsza z nich to funkcja, która jako argument przyjmuje żądaną nową wartość dokładności, a jako rezultat zwraca starą, dotąd obowiązującą. Druga z tych funkcji – wywoływana z pustą listą argumentów – jako rezultat zwraca obecnie obowiązującą dokładność.

Oto użycie tych funkcji:

```
float     x = 72.435672591143 ,
          y = 10.55 ,
          z = 2 ;

     cout << "dokladnosc = "
          << (cout.precision() ) << endl ;
     cout << "x= " << x << " y= " << y
          << " z= " << z << endl ;

     cout.precision(8) ;
     cout << "Teraz dokl. = "
          << (cout.precision() ) << endl ;
     cout << "x= " << x << " y= " << y
          << " z= " << z << endl ;
```

Na ekranie pojawi się wówczas

```
dokladnosc = 0
x= 72.435669 y= 10.55 z= 2
Teraz dokl. = 8
x= 72.43566895 y= 10.55000000 z= 2
```

Jak widać, ustawienie precyzji wypisywania liczb ma efekt trwały. Wszystkie 3 liczby w ostatniej linijce zostały wypisane z ustawioną raz dokładnością. Dokładność obowiązuje do chwili, gdy jej nie zmienimy następnym wywołaniem funkcji `precision`.

Zwróć też uwagę, jak została wypisana liczba `z`. Ten sposób wynika z domniemania – nieznaczące zera, ani kropka dziesiętna, nie mają być wypisywane. Gdyby jednak nam na tym zależało, to wystarczy owo domniemanie zmienić wywołując funkcję

```
cout.setf(ios::showpoint);
```

Na ekranie wówczas zobaczylibyśmy

```
x= 72.43566895 y= 10.55000000 z= 2.00000000
```

Innym, może wygodniejszym sposobem osiągnięcia tego samego efektu, co wywołanie funkcji `precision`, jest posłużenie się manipulatorem `setprecision`. Manipulatorami zajmiemy się w następnym paragrafie.

21.11 Manipulatory

Manipulatory to specjalne wartości, które można wstawić do strumienia albo z niego wyjąć po to, by wywołać zamierzony efekt uboczny, polegający na zmianie sposobu formatowania.

Zauważ jak niewygodny jest sposób z przedstawionymi funkcjami składowymi poznanymi w poprzednich paragrafach. Jeśli chcemy na ekranie wypisać liczbę raz w zapisie dziesiątkowym, a potem w zapisie szesnastkowym, to posługując się funkcją `setf` robimy to tak:

```
int i = 30 ;

     cout << i ;
     cout.setf(ios::hex, ios::basefield) ;
     cout << ", " << i ;
```

Na ekranie pojawi się wówczas

```
     30, 1e
```

Jak widać, zrobiliśmy to tak: najpierw wstawiliśmy do strumienia `cout` zmienną `i`. Następnie przerwaliśmy wstawianie i wywołaliśmy na rzecz strumienia `cout` jego funkcję `setf`. Wreszcie znowu wstawiliśmy do strumienia zmienną `i`, która teraz została już wypisana heksadecymalnie.

Zobaczmy jak można to zrobić sprawniej, posługując się manipulatorem o nazwie `hex`

```
     cout << i << ", " << hex << i ;
```

Pierwszą rzeczą, która rzuca się w oczy, jest prostota zapisu. Mimo, że efekt na ekranie jest ten sam, to jednak zasada jest inna. Manipulator wstawia się do strumienia tak, jakbyśmy to robili chcąc wypisać na ekranie jakiś obiekt o nazwie `hex`.

> *Bardziej obrazowe porównanie jest takie, że manipulator to jest coś, co wpuszcza się do strumienia, jak zatrutą wodę. To, co wpuszczamy, nie ma wcale popłynąć na ekran. Jest to jakby komunikat dla samego strumienia. Innym porównaniem może być chociażby działanie hormonów. Jeden organ z drugim komunikuje się nie bezpośrednio, ale przez wpuszczenie do krwiobiegu śladowych ilości hormonów, których obecność w krwi tamten drugi potrafi wykryć i na nie zareagować.*

Manipulatory są wygodnym narzędziem do zmian sposobu formatowania przez strumienie we/wy. Zostały one zebrane także w klasie `ios`. Dlatego wszystkie klasy będące pochodnymi klasy `ios` mogą się nimi posługiwać.

Skoro tyle mówiłem o tzw. rozszerzalności języka C++, wypada powiedzieć, że objawia się ona i tutaj. Manipulatory zebrane w klasie `ios`, to tak zwane manipulatory predefiniowane, które zdefiniowane zostały dla nas przez autorów biblioteki. Jeśli jednak dostępne manipulatory nie zadawalają nas – łatwo możemy zdefiniować swoje własne. W następnych paragrafach poznamy dokładniej manipulatory predefiniowane – czyli te już gotowe.

21.11.1 Manipulatory bezargumentowe

Manipulatory sterujące konwersją liczb:
`hex, dec, oct`

dec - ustaw pole `basefield` na konwersję dziesiątkową,
hex - ustaw pole `basefield` na konwersję hexadecymalną,
oct - ustaw pole `basefield` na konwersję oktalną.

Manipulatory te można zastosować wobec strumienia wyjściowego lub wejściowego. Oto przykład:

```
int i = 30 ;

cout << "Dziesiatkowo " << i
     << ", szesnastkowo " << hex << i
     << ", osemkowo " << oct << i << endl;
```

Na ekranie pojawi się

```
Dziesiatkowo 30, szesnastkowo 1e, osemkowo 36
```

Jak już wiemy, ten sam efekt zamiany podstawy konwersji na heksadecymalną można wywołać posługując się kilkakrotnie funkcją składową `setf`

```
int m = 30

cout.setf(ios::hex, ios::basefield) ;
cout << m ;
```

Jednak sposób ten nie jest tak wygodny, jak ten z użyciem manipulatora. Oto przykład zastosowania manipulatorów dla strumienia wejściowego:

```
int j, k ;

     cout << "Podaj liczbe dziesiatkowa : " ;
     cin >> dec >> i ;
     cout << "Podaj liczbe heksadecymalna : " ;
     cin >> hex >> j ;
     cout << "Podaj liczbe oktalna : " ;
     cin >> oct >> k ;

     cout << "Oto podane liczby w zapisie dziesiatkowym "
          << dec << i << ", " << j << ", " << k ;
```

Jeśli na zadane pytania odpowiemy wystukując na klawiaturze liczbę 15, to ekran będzie wyglądał tak

```
Podaj liczbe dziesiatkowa : 15
Podaj liczbe heksadecymalna : 15
Podaj liczbe oktalna : 15
Oto podane liczby w zapisie dziesiatkowym 15, 21, 13
```

Gdybyśmy na pytanie o liczbę w zapisie oktalnym odpowiedzieli 99, to nastąpi błąd, gdyż cyfra 9 jest w zapisie oktalnym niedopuszczalna. Temu, jak takie

błędy się objawiają i jak z nimi postępować, poświęcony będzie osobny paragraf.

Zwracam uwagę, że przy operacjach wczytywania z klawiatury posługujemy się strumieniem `cin`, natomiast do wypisywania na ekran służy nam strumień `cout`. Zatem dwie instrukcje

```
cin >> hex ;
cout << hex ;
```

dokonują zmian stanu formatowania w zupełnie różnych strumieniach. Jedno nie ma wpływu na drugie.

Manipulator `flush`

Jeśli strumień wyjściowy jest buforowany, to dane przeznaczone do wypisywania nie są wysyłane od razu, lecz czekają w buforze (jak w poczekalni na dworcu) aż zbierze się ich większa partia. Wtedy dopiero następuje hurtowe wysłanie. W sumie pozwala to na oszczędność czasu.

Może się jednak zdarzyć, że chcemy by coś, niezależnie od stanu zapełnienia bufora zostało wypisane natychmiast. Służy do tego właśnie manipulator `flush` †). Inicjuje on akcję wypisania treści bufora. To, co czekało w buforze - niezależnie od stopnia zapełnienia – zaczyna płynąć strumieniem. Do wypisywanego tekstu **nie jest nic dołączane**, w szczególności nie jest dołączany znak nowej linii (`'\n'`).

Manipulator ten może okazać się przydatny np. gdy pracujemy z debuggerem (programem uruchamiającym). Jeśli mamy następujący fragment programu

```
cout << "wypis linii 1 " ;
cout << "wypis linii 2 " ;
cout << "wypis linii 3 " ;
```

wówczas – gdy za pomocą debuggera będziemy krokowo (linijka po linijce) wykonywali ten program – to wcale nie znaczy, że po wykonaniu pierwszej linijki pojawi się na ekranie tekst `wypis linii 1`. Może on czekać w buforze, aż uzbiera się coś więcej. Jeśli jednak chcemy, mieć to na ekranie natychmiast, to akcję wypisania zainicjować możemy manipulatorem `flush`

```
cout << "wypis linii 1 " << flush ;
```

Na ekran popłynie cała dotychczasowa zawartość bufora – łącznie z tym stringiem.

`endl` – (end line – zakończ linię)

Powoduje on wstawienie do strumienia znaku nowej linii `'\n'` oraz wywołanie funkcji `flush` (robi ona to samo, co manipulator `flush`). Jeśli piszemy program, który rozmawia z użytkownikiem, to lepiej stosować manipulator `endl` zamiast znaku `'\n'`

---

†) flush – ang. spłukanie [np. bufora] (czytaj: „flasz")

```
cout << "Jest czwarta \n" ;
cout << "Jest czwarta " << endl ;
```

gdyż dzięki `endl` tekst znajdzie się od razu na ekranie.

`ends` (end string – zakończ string)

Manipulator ten powoduje wstawienie do strumienia znaku NULL. Manipulator ten przydaje się najczęściej wtedy, gdy wypisujemy coś nie na ekran, ale do tablicy znaków (patrz dalej: zagadnienia formatowania wewnętrznego). Wpisane tam znaki można zakończyć znakiem końca stringu.

`ws` (white spaces – [przeskocz] białe znaki)

Manipulator powoduje przeskoczenie wszystkich białych znaków oczekujących na wyjęcie ze strumienia. Zwykle stosuje się go do pominięcia spacji poprzedzających właściwy tekst.

```
char wyraz[80] ;
cin >> ws >> wyraz ;
```

21.11.2 Manipulatory parametryzowane

Inaczej mówiąc manipulatory, które mają jakieś argumenty. Znamy to. Aby określić na ilu znakach ma zostać wypisana jakaś dana, trzeba było podać tę liczbę znaków. Ta liczba to właśnie argument.

Aby skorzystać z manipulatorów predefiniowanych, które mają jakieś parametry, musimy umieścić w programie dyrektywę:

```
#include <iomanip.h>
```

Manipulator `setw(int)`

Jest to skrót od: set width – ustaw szerokość. Manipulator ten robi to samo, co wywołanie funkcji składowej `width`, czyli – jak pamiętamy - za jego pomocą możemy sprawić, że np. liczba dwucyfrowa zostanie wypisana na ekranie za pomocą 10 znaków. Te dodatkowe 8, będą to znaki wypełniające (por. `fill`) umieszczone przed lub za liczbą (por. flagi `left`, `right`, `internal`).

Porównaj dwa odpowiadające sobie zapisy:

```
int m = 437 ;

     cout.width(5) ;
     cout << m ;
     cout.width(9) ;
     cout << m ;
     cout.width(7) ;
     cout << m ;
```

oraz

```
cout << setw(5) << m << setw(9) << m << setw(7) << m ;
```

Jak pamiętamy, określenie szerokości nie wpływa na wczytywanie liczb (strumień wejściowy), ale za to ma wpływ na wczytywanie stringów. Określenie

szerokości określa maksymalną liczbę znaków, które można przyjąć w celu umieszczenia ich w tablicy.

```
char slowo[6] ;
cin >> setw( sizeof(slowo) ) >> slowo ;
```

Spowoduje to wczytanie do tablicy `slowo` co najwyżej 5 znaków i dopełnienie tego znakiem NULL.

Podobnie jak i funkcja `width`, tak i użycie manipulatora `setw` ma efekt jednorazowy – to znaczy dotyczy tylko najbliższej operacji we/wy na danym strumieniu. Po niej – szerokość wraca na domniemaną wartość 0 – czyli: „tyle, ile koniecznie trzeba dla wypisania liczby".

Trzeba o tym pamiętać szczególnie wtedy, gdy zdefiniowaliśmy swój własny operator << lub >>. Jeśli nasz operator ma np. wypisać 3 liczby, to poprzedzenie go manipulatorem `setw` wywoła efekt tylko na pierwszej z nich.

Niedawno mówiliśmy o przeładowaniu operatora >> dla klasy `wekt`. Następujące użycie operatora:

```
wekt a ;
     // ...
     cout << setw(15) << a ;
```

spowoduje, że tylko współrzędna x zostanie wypisana za pomocą 15 znaków. Oczywiście jest wyjście. Trzeba w obrębie definicji operatora „dowiedzieć się" o właśnie obowiązującą szerokość i powtórzyć ją przed następnymi wypisywanymi współrzędnymi.

```
ostream & operator<<(ostream &strumien_wyj , wekt &w)
{
int szer = strumien_wyj.width();              // jaka obowiązuje ?

     strumien_wyj << w.x << " "
                  << setw(szer)               // odświeżam
                << w.y << " "
                  << setw(szer)               // odświeżam
                << w.z ;
     return strumien_wyj ;
}
```

Teraz instrukcją

```
cout << setw(15) << a ;
```

wszystkie trzy współrzędne będą poprawnie wypisane – każda za pomocą 15 znaków.

Manipulator `setfill(int)`

Robi on to samo, co funkcja składowa `fill`, czyli pozwala nam zdecydować jakimi znakami mają być dopełnione liczby, które poleciliśmy wypisać za pomocą większej liczby znaków niż potrzeba. Przed chwilą życzyliśmy sobie

by współrzędne wektora były wypisane na 15 znakach. Jeśli akurat jakaś liczba jest równa 0, to wymaga tylko jednego znaku `0`. Pozostałe 14 znaków to tzw. znaki wypełniające, które możemy tym manipulatorem wybrać. Przez domniemanie znakiem wypełniającym jest spacja.

Oto jak posługujemy się tym manipulatorem:

```
int kwota = 3200 ;

cout << "Stan konta : " << setfill('*')
     << setw(9) << kwota << "$" << endl ;
```

To samo bez użycia manipulatora wyglądałoby tak:

```
cout << "Stan konta : " ;
cout.fill('*') ;
cout.width(9);
cout << kwota << "$" << endl ;
```

W obu wypadkach wypisany zostanie tekst

```
Stan konta : *****3200$
```

Jak już wspominałem, ten manipulator może się przydać, gdy piszemy np. program księgujący i musi on drukować rachunki, czeki etc. Puste miejsca przed kwotą dla zmniejszenia ryzyka fałszerstwa zadrukowane są gwiazdkami.

Manipulator `setprecision(int)`

Robi on to samo, co funkcja `precision` – czyli pozwala zmieniać obowiązującą bieżąco dokładność wypisywania liczb zmiennoprzecinkowych. Argumentem tego manipulatora jest liczba `int` określająca żądaną dokładność. Domniemana wartość dokładności to 6 miejsc po kropce dziesiętnej.

Oto przykład zastosowania:

```
double x = 99.123456789 ;

cout << x << " "
     << setprecision(2) << x << " "
     << setprecision(8) << x << endl ;
```

To samo da się oczywiście zrobić za pomocą funkcji `precision`, ale zapis jest bardziej skomplikowany.

```
cout << x << " " ;
cout.precision(2) ;
cout << x << " ";
cout.precision(8) ;
cout << x << endl ;
```

W obu wypadkach spowoduje to wypisanie

```
99.123457 99.12 99.12345679
```

Manipulatory `setiosflags(long)` oraz `resetiosflags(long)`

Odpowiadają funkcjom składowym `setf` i `unsetf` . Ich działanie polega więc na ustawieniu `(setiosflags)` – względnie skasowaniu `(resetiosflags)` podanych flag stanu formatowania.

Oto przykład, w którym ustawiamy flagę `uppercase`:

```
int liczba = 242 ;

cout << hex << liczba << ", "
     << setiosflags(ios::uppercase) << liczba << ", "
     << resetiosflags(ios::uppercase) << liczba << endl ;
```

To samo wykonane za pomocą wywołań funkcji jest wyraźnie dłuższe:

```
cout.setf(ios::hex, ios::basefield);
cout << liczba << ", " ;
cout.setf(ios::uppercase) ;
cout << liczba << ", " ;
cout.unsetf(ios::uppercase) ;
cout << liczba << endl ;
```

❐ **W obu wypadkach spowoduje to pojawienie się na ekranie**

```
f2, F2, f2
```

Przypominam, że funkcja `resetiosflags` użyta do skasowania określonej flagi kasuje tylko ją, inne zostają nietknięte.

Manipulatory `setiosflags` oraz `resetiosflags` mogą służyć do ustawienia (względnie kasowania) równocześnie kilku flag. Robimy to za pomocą sumy bitowej kilku flag. W żargonie mówi się na to: „OR-owanie" – jako że służy do tego operator bitowy *or* `'|'`

```
cout << setiosflags(ios::uppercase | ios::showpos) ;
```

Tak zastosowany manipulator równocześnie ustawia flagę `uppercase` i flagę `showpos`. Inne pozostają bez zmian. Natomiast takie zastosowanie manipulatora

```
cout << resetiosflags(ios::showbase | ios::unitbuf);
```

powoduje skasowanie flag `showbase` i `unitbuf`. Inne pozostają bez zmian.

Manipulator `setbase(int)`

Nie wszystkie implementacje biblioteki zawierają realizację tego manipulatora†). Służy on do określania podstawy konwersji liczb – czyli tego samego, co użycie opisanych już manipulatorów `hex`, `dec`, `oct`.

Jak łatwo się domyślić, poniższe zapisy odpowiadają sobie

†) Borland C++ tak!

```
cout << dec << hex << oct ;
cout << setbase(10) << setbase(16) << setbase(8) ;
```

Dodatkowo jednak możliwe jest użycie manipulatora z argumentem równym 0

```
cout << setbase(0) ;
```

Powoduje on powrót do domniemanego sposobu konwersji, czyli takiego, gdzie żadna z flag na polu `ios::basefield` nie jest ustawiona. Jak już mówiliśmy oznacza to:

- wobec strumienia wyjściowego (np. `cout`) – ustawienie konwersji na dziesiątkową,
- wobec strumienia wejściowego (np. `cin`) – konwersja będzie przeprowadzana na podstawie wyglądu wczytywanej właśnie liczby ;
  czyli liczba będzie w zasadzie uznana za dziesiątkową chyba, że przed liczbą wystąpią znaki `0x...` – co sugeruje, że zapis jest heksadecymalny, a jeśli liczbę rozpocznie `0` – wtedy sugeruje to że zapis jest oktalny.

21.11.3 Definiowanie swoich manipulatorów

Ten paragraf proponuję zdecydowanie opuścić przy pierwszym czytaniu. Mówić tu będziemy o tym, jak można pisać swoje własne manipulatory. Nie jest to trudne, ale na pewno nie będziesz tego chciał robić natychmiast. Proponuję przeskoczyć do następnego paragrafu – o operacjach nieformatowanych.

Są tu dwa zagadnienia: manipulatory bez argumentów (łatwiejsze) i z argumentem (trudniejsze).

Najprostszym sposobem jest posłużenie się dyrektywą preprocesora. Jeśli np. chcemy zdefiniować manipulator, który wstawi przecinek między dwie wypisywane liczby `a` i `b`, to możemy zrobić tak

```
#define sep ", "                    // sep - jak separator
```

wówczas zapis

```
cout << a << sep << b ;
```

odpowiada to zapisowi

```
cout << a << ", " << b ;
```

Nie jest to jednak pewny sposób. W programie może być już gdzieś użyty symbol `sep`, jako nazwa obiektu lub jakiejś funkcji. Nastąpi konflikt.

Najpewniej zdefiniować manipulator jako funkcję. Wówczas nawet jeśli już w programie istnieje jakoś globalna funkcja o nazwie `sep` - to nastąpi przeładowanie tej nazwy. Właściwa funkcja będzie rozpoznawana na podstawie zgodności argumentów.

Oto realizacja takiej funkcji:

```
ostream & sep(ostream & strum)
{
     strum << ", " ;
     return strum ;
}
```

Zwracam uwagę, że nie jest to przeładowanie żadnego operatora. To zwykła funkcja globalna o nazwie `sep`. Oczywiście, skoro znajduje się ona między operatorami, to musi się odpowiednio zachować.

Dlatego funkcja ta jako argument przyjmuje referencję do strumienia. Jako rezultat funkcja ta zwraca referencję do tegoż strumienia. Jeśli będziesz pisał swój manipulator, to musisz dochować tej konwencji. Dzięki temu manipulator może wystąpić w „kaskadzie" – w połączeniu z typami wbudowanymi.

```
cout << a << sep << b ;
```

Nie dziw się, że nazwa funkcji `sep` została tu użyta bez nawiasów i argumentów. Do strumienia wpuszczana jest *tylko nazwa* funkcji (czyli jakby jej adres). Strumień sam sobie uruchomi tę funkcję i w dodatku jako argument wyśle jej swoją referencję.

Pomyślałeś zapewne: „Tyle zachodu po to, by zaoszczędzić na pisaniu dwóch cudzysłowów, przecinka i spacji?"

Rzeczywiście masz rację. Na usprawiedliwienie dodam, że moim zdaniem łatwiej na klawiaturze wystukać `sep` niż `", "` gdyż to wymaga dwukrotnego przyciśnięcia klawisza shift – a to absorbuje trochę uwagi. Z lenistwa można więc posługiwać się tym manipulatorem.

Sposobem tym można też zdefiniować manipulator, który wykona dla nas więcej pracy. Załóżmy, że w programie mamy często wypisywać dane takiego typu:

- temperaturę – z dokładnością do 1 miejsca po kropce dziesiętnej i z zaznaczeniem znaku minus oraz plus,
- wysokość – w liczbach całkowitych, bez znaku plus (bo ujemna być nie może),
- gęstość – w tzw. notacji naukowej.

Wymaga to, jak widać, częstego stosowania różnych kombinacji znanych nam manipulatorów. Można jednak te manipulatory – wymagane do wypisania jednego typu danej – zapakować do manipulatora zdefiniowanego przez siebie.

Oto przykładowa realizacja takich manipulatorów i sposób ich wykorzystania. Zastrzegam, że w tym przykładzie manipulatorami posługujemy się tylko jednokrotnie, więc trudno docenić jak bardzo ułatwiają nam pracę.

```
#include <iostream.h>
#include <iomanip.h>

ostream & tem(ostream & strum)
{
     strum << setprecision(1) << setiosflags(ios::showpos)
```

```
            << setiosflags(ios::fixed)
            << resetiosflags(ios::scientific) ;
      return strum ;
}
/*****************************************************/
ostream & wys(ostream & strum)
{
      strum << resetiosflags(ios::showpos) ;
      return strum ;
}
/*****************************************************/
ostream & ges(ostream & strum)
{
      strum << resetiosflags(ios::fixed)
            << setiosflags(ios::scientific)
            << setprecision(4) ;
      return strum ;
}
/*****************************************************/
main()
{
float t = 20.47,
      g = 0.09273 ;
int w = 3000 ;

cout << " temperatura= " << tem << t
     << " wysokosc = "<< wys << w
     << " gestosc = "<< ges << g
     << endl ;
}
```

Wykonanie tego programu da na ekranie następujący efekt

```
temperatura= +20.5 wysokosc = 3000 gestosc = 9.273e-02
```

Korzyść staje się naprawdę widoczna, gdy liczby tego typu musimy wielokrotnie wypisywać w różnych miejscach programu, a jeszcze lepiej wtedy, gdy pewnego dnia zmienimy zdanie co do sposobu wypisywania danych jednego typu (np. gęstości). Wówczas zmienić należy w programie tylko ciało manipulatora `ges`.

Czy można zdefiniować manipulatory, które będą przyjmowały jakiś argument - tak, jak to robi np. manipulator `setw`?

```
cout << setw(7) << liczba ;
```

Można, niektóre realizacje biblioteki pozwalają na to. Nie jest to jednak rozwiązane w elegancki sposób. Manipulatory takie powinny być zrealizowane jako tzw. **szablony** (ang. templates). Szablony to specjalny rodzaj narzędzia. Do wersji 2.1 języka C++ zostały one włączone jako eksperyment, więc nie będziemy się nimi zajmować.

Manipulatory parametryzowane mogą być definiowane przez użytkownika bez użycia tych szablonów – za pomocą grupy specjalnych makrodefinicji. Te makrodefinicje mają symulować szablony. Powtarzam: nie jest to eleganckie rozwiązanie. Jeśli chcesz się tym zająć – odsyłam Cię do opisu Twojej biblioteki wejścia/wyjścia. Najpierw sprawdź czy w Twojej bibliotece jest to możliwe. Rozpoznasz to po obejrzeniu pliku nagłówkowego `iomanip.h`. Powinny się w nim znajdować np. deklaracje `IOMANIPdeclare(int)` oraz `IOMANIPdeclare(long)`

21.12 Nieformatowane operacje wejścia/wyjścia

Różnice między operacjami formatowanymi a nieformatowanymi

Operatory << >> wstawiania i wyjmowania ze strumienia przeprowadzają operacje we/wy na poziomie wyższym, to znaczy z obecnością formatowania. Formatowanie takie jest sprawą prostą tylko w wypadku przesyłania stringu — znaki do przesłania są już wtedy przecież gotowe i nie trzeba ich interpretować. Jednakże w wypadku liczb wymagane jest formatowanie.

Na przykład odczytuje się wartość liczby `int` czy `float` zapisaną binarnie w danych komórkach pamięci, a następnie trzeba tę wartość przedstawić za pomocą grupy jakichś cyfr, które mają się pojawić na ekranie.

Podobnie jest z wczytywaniem liczb. Nie sądzisz chyba, że wystukana na klawiaturze liczba

```
-3.4e+02
```

jest w ten sposób przechowywana w komórkach pamięci operacyjnej. Tą interpretacją (formatowaniem) zajmowały się operatory wczytywania >> i wypisywania <<. Na ich pracę mieliśmy wpływ zmieniając flagi stanu formatowania.

Są jednak sytuacje, gdy nie jest nam potrzebne żadne formatowanie. Np. chcemy przyjmować znaki płynące z klawiatury niezależnie od tego jaką informację niosą, niezależnie od tego, czy są to spacje, czy znaki alfanumeryczne. Poziom formatowania jest nam niepotrzebny.

Operacje nieformatowane – prezentacja funkcji wyjmujących ze strumienia

Dla operacji nieformatowanych wejścia/wyjścia mamy do dyspozycji zestaw funkcji składowych znajdujących się w klasach `istream` oraz `ostream`.

W klasie `istream` są funkcje odpowiadające za nieformatowane *wczytywanie* informacji. Inaczej mówiąc: za wyjmowanie informacji ze strumienia, bez interpretacji jej. Oto deklaracje tych funkcji:

```
istream&  get(char &) ;                          ❶
int       get() ;

istream& get(char *, int, int = '\n') ;          ❷
istream& get(istream &, int) ;
int getline(char *, int, int = '\n') ;
```

```
istream & read(char *, int) ;                ❸
istream & ignore(int , int);                 ❹
```

Funkcjom tym poświęcimy osobne paragrafy. W skrócie można powiedzieć, że:

❶ Pierwsze dwie służą do wczytania jednego znaku (bajtu),

❷ Następne trzy pozwalają wczytać zadaną liczbę znaków, chyba że napotkany zostanie specjalny znak uznawany przez nas za ogranicznik,

❸ Funkcja `read` służy do wczytania określonej liczby bajtów – bez względu na ich treść,

❹ Funkcja `ignore` wczytuje (wyjmuje ze strumienia) grupę bajtów i ignoruje je — czyli nie umieszcza ich nigdzie. Zostają one od razu zapomniane.

W klasie `iostream` mamy jeszcze dalsze pożyteczne funkcje:

```
int gcount() ;
int peek() ;
istream & putback(char);
```

`gcount` – Pozwala dowiedzieć się ile znaków zostało wczytanych za pomocą ostatniej operacji nieformatowanego wczytywania (wyjmowania ze strumienia).

`peek` – Służy do podglądnięcia jaki to bajt czeka w strumieniu na wyjęcie go. Funkcja ta tylko podgląda, nie wyjmując niczego.

`putback` – Pozwala rozmyślić się i zwrócić do strumienia właśnie wczytany bajt. Bajt ten wraca do strumienia i może być jeszcze raz wyjęty najbliższą operacją wyjmowania z tego strumienia.

Funkcje służące do nieformatowanego wkładania do strumienia

W klasie `ostream` są funkcję składowe odpowiadające za nieformatowane wypisywanie informacji.

```
ostream put(char) ;
ostream write(char *, int) ;
```

Pierwsza z nich wkłada do strumienia jeden bajt, druga zaś może włożyć większą liczbę bajtów. Także i tym funkcjom poświęcimy dodatkowe paragrafy.

W klasie `iostream`, która jak wspominaliśmy jest pochodną od klasy `istream` oraz `ostream`, odziedziczone są wszystkie powyższe funkcje – te wyjmujące i te wstawiające do strumienia.

Na koniec jeszcze jedno zastrzeżenie:

W powyżej deklarowanych funkcjach wskaźnik do bufora, z którego pochodzą bajty kierowane do strumienia, określany jest jako `char*` . Pamiętać należy, że równie dobrze możemy mieć wskaźnik typu `unsigned char*` . Nie dlatego, żeby było to wszystko jedno. Wręcz przeciwnie, są to zupełnie różne typy. Po prostu dlatego, że istnieje drugi komplet takich funkcji zdeklarowany na okoliczność wskaźników `unsigned char*` . Dla oszczędności miejsca nie zamie-

szczam deklaracji tego drugiego kompletu. Jednak pamiętać należy, że np. funkcja

```
ostream & write (char*, int);
```

ma brata bliźniaka

```
ostream & write (unsigned char*, int);
```

W ten sposób zdublowane są zarówno funkcje wstawiające do strumienia, jak i wyjmujące z niego.

21.13 Omówienie funkcji wyjmujących ze strumienia

21.13.1 Funkcje do pracy ze znakami i stringami

Funkcja `istream & get(char & znak) ;`

Funkcja ta wyjmuje ze strumienia jeden bajt i umieszcza go w podanej zmiennej `znak`. Na przykład takie użycie tej funkcji:

```
char c ;
     cin.getc(c) ;
```

spowoduje wczytanie jednego bajtu ze strumienia `cin` płynącego z klawiatury i umieszczenie go w zmiennej `c`. Dla porównania taka operacja:

```
     cin >> c ;
```

nie potrafi wczytać znaku spacji. Do `c` zostanie wczytany dopiero pierwszy nie-biały znak. Tymczasem funkcja `get` potrafi wczytać wszystko, nawet biały znak.

Jako rezultat funkcja `get` zwraca referencję do strumienia, na którym pracowała. Pozwala to na wygodne kaskadowe użycie tej funkcji

```
cin.get(a).get(b).get(c) ;
```

Powyższa linijka spowoduje wyjęcie ze strumienia `cin` trzech bajtów i umieszczenie ich w `a`, `b` oraz `c`.

Jeśli oczekujemy funkcją `get` na znak z klawiatury, a okaże się, że jest to znak końca pliku danych EOF†) –to wówczas rezultat zwracany przez funkcję będzie NULL (a nie referencja do strumienia, na którym pracowała).

To, czym jest w danym komputerze znak końca pliku EOF – zdefiniowane jest w pliku nagłówkowym `iostream.h`

Niezależnie od rezultatu NULL, ów znak EOF zostanie wpisany do oczekującej na niego zmiennej `char`.

†) end of file – ang. koniec pliku, (czytaj: „end of fajl")

Dzięki tej zasadzie funkcję tę można umieścić w pętli `while`, która będzie wykonywała się dopóki nie napotkany zostanie znak końca pliku.

```
char znak ;
     cout << "Wciskaj rozne klawisze (^Z - koniec)\n" ;
     while( (cin.get(znak)) != NULL )
     {
          cout << "Wcisnales " << znak
               << ", kod : " << (int)znak << endl ;
     }
     cout << "Pozegnalny znak mial kod " << (int)znak << endl;
```

Funkcja `int get(void) ;`

Podobnie jak poprzednia funkcja, także i ta służy do wczytania jednego znaku. Różnica jest ta, że wczytany znak jest rezultatem zwracanym przez funkcję

```
int z ;
 z = cin.get() ;
```

Zauważ, że funkcja zwraca typ `int`, a nie typ `char`. To dlatego, żeby funkcja mogła zwrócić jako rezultat EOF jeśli ze strumienia został wyjęty znak końca danych.

Oto przykład podobny do poprzedniego:

```
char znak ;
     cout << "Wciskaj rozne klawisze (^Z - koniec) "
          << endl ;
     while( (znak = cin.get()) != EOF )
     {
          cout << "Wcisnales " << znak
               << ", kod : " << (int)znak << endl ;
     }
     cout << "Pozegnalny znak mial kod " << (int) znak << endl;
```

Jeśli mamy wczytywać dużo bajtów, to można tego oczywiście dokonać za pomocą funkcji `get` umieszczonej w pętli, jednakże lepiej zrobić to funkcjami przeznaczonymi do wczytywania wielu znaków.

Funkcja `istream & get(char* gdzie, int dlugosc, char ogran='\n');`

Tą wersją funkcji `get` posługujemy się, gdy chodzi nam o wczytanie z strumienia nie jednego bajtu, ale większej ilości.

Wyjaśnijmy sobie najpierw jej argumenty:

`char*` – to adres tablicy, do której należy wpisywać wyjmowane ze strumienia znaki.

`int` – to liczba decydująca ile (maksymalnie) znaków można wczytać. Zwykle w tym miejscu stawia się liczbę określającą rozmiar tablicy,

w której umieszczane są wyjmowane ze strumienia znaki. Ponieważ string umieszczony w tablicy musi mieć na końcu znak NULL – dlatego, jeśli ten argument będzie miał wartość 7, to wyjętych ze strumienia będzie maksymalnie 6 znaków, bo na siódmej pozycji dopisany zostanie znak NULL.

`char` – ten trzeci argument to ogranicznik, który określa kiedy przerwać wczytywanie. Domniemana wartość tego argumentu jest `'\n'`. Jeśli podczas wyjmowania ze strumienia napotkany zostanie ten znak, to wyjmowanie zostanie przerwane – nawet jeśli nie osiągnęliśmy jeszcze ilości określonej poprzednim argumentem.

Niezależnie od tego, czy przerwanie wczytywania nastąpi po osiągnięciu maksymalnej liczby znaków, czy też po napotkaniu ogranicznika – do wczytanych znaków dopisywany jest znak NULL kończący poprawnie string.

Rezultat zwracany przez funkcję to referencja do strumienia, z którym pracujemy. Pamiętamy, że to umożliwia kaskadowe łączenie takich funkcji.

Jeśli oczekujemy funkcją `get` na znak z klawiatury, a okaże się, że jest to znak końca pliku danych EOF – to wówczas rezultat zwracany przez funkcję będzie: NULL, (zamiast referencji do strumienia, na którym pracowała).

Oto ilustracja działania tej funkcji:

```
#include <iostream.h>
/*****************************************************/
main()
{
char imie[7],
     nazwisko[20],
     c ;

     cout << "Podaj swoje imie : " ;
     cin.get(imie, 7);                                  // ❶
     cout << "Podales -->" << imie                      // ❷
          << "\nteraz nazwisko : " ;
     cin.get(nazwisko, 20, 's' );                       // ❸
     cout << "\nNazwisko brzmi -->"
          << nazwisko << endl ;                         // ❹
     cout << "przyjecie jeszcze jednego znaku : " ;
     cin.get(c) ;                                       // ❺
     cout << "\nKolejny wczytany znak -->" << c << endl ;
}
```

❒ **Przykładowe odpowiedzi są poniżej zaznaczone tłustym drukiem**

```
Podaj swoje imie : Konstanty Galczynski
Podales -->Konsta
teraz nazwisko :
Nazwisko brzmi -->nty Galczyn
przyjecie jeszcze jednego znaku :
Kolejny wczytany znak -->s
```

Komentarz

❶ Tutaj program oczekuje tylko na (maksymalnie) 6 znaków alfanumerycznych. Mimo tego, wpisujemy tutaj całość (imię i nazwisko), co oczywiście przekracza żądaną liczbę.

❷ Co prawda my wszystkie znaki wystukaliśmy od razu na klawiaturze, jednak funkcja `get` wyjęła z tego strumienia tylko 6. Na dowód tego wypisujemy na ekran zawartość tablicy `imie`. Reszta znaków jest nadal w strumieniu i oczekuje na wyjęcie (strumień jest jakby zatamowany).

❸ Następna funkcja `get` wyjmie znowu część z tych znaków. Z jej argumentów widzimy, że wyjmie co najwyżej 19 znaków – chyba, że napotka ogranicznik: znak `'s'`. Wtedy zakończy wyjmowanie, a znak `s` i ewentualne dalsze czekać będą na „następnych chętnych".

❹ Na dowód tego, co zostało wyjęte i wpisane do tablicy `nazwisko`, wypisujemy to na ekran.

❺ Reszta znaków nadal czeka w strumieniu. Tutaj widzimy właśnie „następnego chętnego". Ta funkcja `get` bierze jednak tylko jeden znak – pierwszym do wzięcia jest właśnie znak, który spowodował zatrzymanie wyjmowania funkcją ❸ czyli znak `'s'`. Skoro na tym kończy się program, to pozostałe w strumieniu znaki `'k' 'i'` nie zostają z niego wyjęte.

Tego przykładu nie należy traktować jako prezentację wad funkcji `get`. Pokazuje on raczej jak można skorzystać z jej dodatkowych możliwości.

Omówiona funkcja `get` zabezpiecza nas przed przepełnieniem tablicy, do której kierujemy wyjmowane ze strumienia znaki. Z kolei możliwość zdefiniowania ogranicznika jest bardzo przydatna. Nie musi to być klawisz Enter – możemy też kończyć wyjmowanie ze strumienia po napotkaniu kodu klawisza Escape i odpowiednio na taką sytuację reagować.

Zapamiętaj:

> Jeśli wczytujemy funkcją `get`, to napotkany ogranicznik nie jest wyjmowany ze strumienia.

(Następna omawiana funkcja `getline` – różni się tym od tej, że ogranicznik jest wyjmowany ze strumienia i wyrzucany).

Funkcja `istream & getline(char * gdzie, int ile, char ogran = '\n');`

W sytuacji, gdy chodzi nam o to, by ogranicznik był także wyjęty ze strumienia posługujemy się funkcją getline[†)].

Rezultat zwracany przez funkcję to referencja do strumienia, z którym pracujemy. Pamiętamy, że to umożliwia kaskadowe łączenie takich funkcji.

†) get line – ang: weź linię (czytaj: „get lajn")

Argumenty są dokładnie takie same, jak przy poprzedniej funkcji `get` - czyli:

`char*` – to adres miejsca w pamięci, gdzie należy lokować wyjmowane ze strumienia znaki.

`int` – to liczba decydująca ile (maksymalnie) znaków można wczytać. Zwykle w tym miejscu stawia się liczbę określającą rozmiar tablicy, w której umieszczane są wyjmowane ze strumienia znaki. Ponieważ string umieszczony w tablicy musi mieć na końcu znak NULL – dlatego jeśli w ten argument będzie miał wartość 7, to wyjętych ze strumienia będzie maksymalnie 6 znaków, bo na siódmej pozycji dopisany zostanie znak NULL.

`char` – ten trzeci argument to ogranicznik, który określa kiedy przerwać wczytywanie. Domniemana wartość tego argumentu jest `'\n'`. Jeśli podczas wyjmowania ze strumienia napotkany zostanie ten znak, to wyjmowanie zostanie przerwane – nawet jeśli nie osiągnęliśmy jeszcze ilości określonej poprzednim argumentem.

Niezależnie od tego, czy przerwanie wczytywania nastąpi po osiągnięciu maksymalnej liczby znaków, czy też po napotkaniu ogranicznika – do wczytanych znaków dopisywany jest znak NULL kończący poprawnie string.

> W wyniku działania tej funkcji ewentualny ogranicznik przerywa wczytywanie znaków, jest on jednak także wyjmowany ze strumienia, ale nie dołączany do właśnie wczytanych znaków. Po prostu wyjmujemy go, ale po to, by go wyrzucić.

Oto przykład:

```
#include <iostream.h>
/*****************************************************/
main()
{
char kuferek[10] = { "xxxxxxxxx" };

     cout << "Napisz max 9 znakow : " << endl ;
     cin.getline(kuferek, sizeof(kuferek), 's' ) ;

     cout << "Oto zawartosc elementow kuferka :\n" ;
     for(int i = 0 ; i < sizeof(kuferek) ; i ++)
     {
           cout  << "kuferek[" << i << "]= "
                 << kuferek[i] << endl ;
     }
     cout << "Nastepnie wyjety znak : "
          << (char)(cin.get()) ;
}
```

Na ekranie zobaczymy poniższy tekst

```
Napisz max 9 znakow : a bcsdefgh
Oto zawartosc elementow kuferka :
kuferek[0]= a
kuferek[1]=
kuferek[2]= b
kuferek[3]= c
```

```
kuferek[4]=
kuferek[5]= x
kuferek[6]= x
kuferek[7]= x
kuferek[8]= x
kuferek[9]=
Nastepnie wyjety znak : d
```

Komentarz

Zauważ, że w przeciwieństwie do poprzedniego przykładu ogranicznik (znak `'s'`) nie pojawił się już w następnej funkcji wyjmującej ze strumienia.

Już sama nazwa funkcji sugeruje, do czego może się ona przydać getline - weź linijkę tekstu. Jest to więc kolejny sposób na wczytanie stringu składającego się z kilku wyrazów.

21.13.2 Wczytywanie binarne – funkcja `read`

Kolejną funkcją składowa klasy `istream` jest funkcja

```
istream & read(char* gdzie, int ile) ;
```

Z deklaracji tej funkcji widzimy, że jako rezultat zwraca ona referencję do strumienia na którym pracuje. Jak wiemy, cecha ta umożliwia kaskadowe łączenie wywołań tej funkcji.

Argumenty:

`char*` – miejsce w pamięci, gdzie należy lokować wyjmowane ze strumienia bajty,

`int` – liczba bajtów, które należy ze strumienia wyjąć.

Oto sposób zastosowania. Załóżmy, że chcemy ze strumienia płynącego z klawiatury wyjąć dokładnie 6 bajtów i umieścić je w tablicy `dane`. Realizujemy to tak:

```
char dane[10] ;
     cin.read(dane, 6) ;
```

Funkcja ta pracuje rzeczywiście na bajtach – nie interesuje ją co znaczą te bajty. W szczególności – funkcji tej nie interesuje czy wczytujemy dane binarne czy string. Niezależnie od tego czy ze strumienia wyjmowane będą znaki alfanumeryczne, czy też znaki nowej linii, czy końca pliku – będą one umieszczane we wskazanym miejscu pamięci. Tą funkcją wczytujemy na przykład z pliku dyskowego dane binarne.

Jeśli jej użyjemy w stosunku do strumienia wejściowego `cin`, to pamiętać należy, iż do wystukiwanego na klawiaturze stringu nie zostanie na końcu dopisany automatycznie znak NULL.

Gdyby nam o to chodziło – powinniśmy po prostu użyć funkcji `get(char*, int)`.

Prawdziwą domeną zastosowania tej funkcji są więc binarne operacje wejścia/wyjścia. Takie operacje rzadko przeprowadza się z klawiaturą. Najczęściej pracuje się w ten sposób z plikami dyskowymi. Funkcja `read` pozwala nam wczytać z dysku parę tysięcy bajtów będących np. rysunkiem lub danymi pochodzących z eksperymentu. Takie dane mogą zawierać nawet takie bajty, które akurat odpowiadają znakowi NULL. Nie ma to wpływ na przebieg przesłania.

W czasie przesyłania bajtów strumieniem może nastąpić błąd. Na przykład zażądaliśmy wczytania 1000 bajtów, a w pliku jest ich tylko 800. O tym, ile bajtów zostało rzeczywiście wczytanych, możemy przekonać się wykonując funkcję `gcount` (patrz dalej).

Oto przykład:

```
char napis[10] ;
 cout<< "Wczytam tylko 4 bajty, napisz cos : " ;
 cin.read(napis, 4) ;
```

W rezultacie w tablicy znajdą się 4 bajty, które przypłynęły strumieniem z klawiatury. Do nich nie zostanie dołączony znak NULL. O pracy tej funkcji z plikami porozmawiamy dalej.

Funkcja `read` jest bardzo sprawnym narzędziem do binarnych operacji wczytywania. Jeśli jednak program ma wykonywać wyjątkowo dużo takich operacji, a zależy nam na czasie – to należy rozważyć posłużenie się klasami pochodnymi od klasy `streambuf`. Co do szczegółów odsyłam do opisu implementacji biblioteki kompilatora.

21.13.3 Funkcja `ignore`

Można powiedzieć obrazowo, że jest to taka funkcja do wyjmowania ze strumienia, która to, co przeczyta, od razu wyrzuca na śmietnik. Czasem może się to przydać do przeskoczenia jakichś niechcianych bajtów, które musimy ze strumienia usunąć, by móc potem czytać następne.

Oto jej deklaracja:

```
istream & ignore(int ile = 1, int ogran = EOF) ;
```

Starym zwyczajem rezultat zwracany przez funkcję to referencja do strumienia wejściowego, na którym funkcja pracuje.

- Pierwszy argument `int` to liczba bajtów, które należy zignorować. Zignorowana będzie taka liczba bajtów – chyba, że wcześniej skończy się plik. (Domniemana wartość tego argumentu = 1 bajt).
- Drugi argument `int` jest to ogranicznik. Jeśli w trakcie wczytywania i ignorowania kolejnych bajtów napotkany zostanie

bajt odpowiadający temu ogranicznikowi wówczas akcja wyjmowania ze strumienia zostanie przerwana.

Zauważ, że ten ogranicznik jest typu `int` a nie, jak poprzednio, typu `char`. To po to, byśmy jako ogranicznik mogli użyć znak EOF (takie jest zresztą domniemanie).

Jeśli ogranicznikiem jest EOF, to ignorowanie nie będzie przerwane z powodu żadnego ogranicznika. To dlatego, że w pliku wczytywanym są bajty, a żaden z nich nie równa się dwubajtowemu EOF. Jeśli np. w Twoim komputerze EOF jest kodowane jako -1 to jego wartość heksadecymalna wynosi `0xffff`, podczas, gdy wyjmowane ze strumienia bajty mogą być co najwyżej z zakresu liczb `0x0 0xff`

Użycie EOF jako ogranicznika oznacza, że nie życzymy sobie żadnego ogranicznika.

Przykład:

```
#include <iostream.h>
/*****************************************************/
main()
{
char kuferek[10] ;
char skrytka[10] ;

     cout << "Napisz okolo 10 znakow : " ;
     cin.getline(kuferek, 4).ignore(2).getline(skrytka, 10);
     cout << "\W kuferku jest :"<< kuferek
          << ", w skrytce jest :" << skrytka
          << "\na dwa znaki zignorowalem " << endl ;
}
```

Oto przykładowy wygląd ekranu:

```
Napisz okolo 10 znakow : abcdefghijkl
W kuferku jest :abc, w skrytce jest :fghijkl
a dwa znaki zignorowalem
```

21.13.4 Pożyteczne funkcje pomocnicze

Funkcja `gcount`

O tym, ile konkretnie znaków zostało wyjętych ze strumienia za pomocą ostatniej funkcji wczytywania nieformatowanego, informuje nas funkcja

```
int gcount() ;
```

Przykład:

```
#include <iostream.h>
/*****************************************************/
main()
{
char tablica[200] ;
```

```
        cout << "Napisz jakies zdanie : " ;
        cin.get(tablica, sizeof(tablica) , 'x' );        // ❶

        cout << "Ze strumienia wyjeta zostalo :"          //
             << ( cin.gcount() )
             << " znakow, \noto zdanie :"
             << tablica << endl ;
}
```

Oto ekran:

```
Napisz jakies zdanie : abcxdef
Ze strumienia wyjeta zostalo :3 znakow,
oto zdanie :abc
```

Komentarz:

❶ Zagadka: gdybyśmy tu zamiast funkcji `get`, zastosowali funkcję `getline`, to mimo podania tego samego tekstu funkcja `gcount` odpowiedziałaby liczbą 4 (zamiast 3). Dlaczego?
Odpowiedź: funkcja `gcount` mówi ile znaków zostało wyjętych ze strumienia. Jak pamiętamy, w przeciwieństwie do funkcji `get`, funkcja `getline` po napotkaniu ogranicznika wyjmuje go ze strumienia (i wyrzuca). Liczba 4 zamiast 3 jest właśnie dowodem na to.

W naszym przykładzie korzystaliśmy ze strumienia płynącego z klawiatury. Funkcja `gcount` tak naprawdę przydaje się szczególnie przy binarnym odczytywaniu pliku funkcją `read`.

Funkcja `peek`

Inną pożyteczną funkcją pomocniczą jest

```
int peek();
```

Jest to funkcja, która pozwala nam zajrzeć do strumienia i zobaczyć co też tam czeka na wyjęcie najbliższą instrukcją wyjmującą. Podglądnąć można tylko jeden bajt. Mimo, że poznajemy go – bajt nie jest ze strumienia wczytywany. Bajt ten jest właśnie wartością (rezultatem) tej funkcji. Rezultat ma typ `int` dlatego, by można było zwrócić EOF w wypadku, gdy dotrzemy do końca pliku.

Funkcję tę używamy zwykle aby odpowiednio zareagować na to, co czeka jeszcze w strumieniu. W zależności od tego, co tam zobaczyliśmy możemy uruchomić odpowiednią akcję wczytywania.

```
#include <iostream.h>
#include <ctype.h>                                        // ❸
/*****************************************************/
main()
{
char nazwa[200] ;
```

```
float x ;
int zwiastun ;
     cout << "napisz liczbe lub nazwe : " ;
     zwiastun = cin.peek() ;                          // ❶

     if(isdigit(zwiastun) ){                          // ❷
          cin >> x ;                                  // ❹
          cout << "Byla to liczba : " << x << endl ;
     }else {
          cin >> nazwa ;                              // ❺
          cout << "Byla to nazwa : " << nazwa << endl ;
     }
}
```

Oto dwa warianty ekranu:

Tak będzie on wyglądał w wypadku napisania nazwy

```
Napisz liczbe lub nazwe : Berlin
Byla to nazwa : Berlin
```

A tak po napisaniu liczby

```
Napisz liczbe lub nazwe : 12.73
Byla to liczba : 12.73
```

Komentarz

❶ Tutaj podglądamy co jest pierwszym oczekującym na wyjęcie ze strumienia znakiem.

❷ Posługujemy się funkcją biblioteczną `isdigit` („is digit?" – znaczy po angielsku: „czy to jest cyfra?"). Aby skorzystać z tej funkcji konieczne jest włączenie pliku nagłówkowego ❸. Funkcja ta odpowiada nam czy jej argument jest cyfrą czy nie.
Wiedząc już czy mamy wyjąć liczbę czy nazwę, stosujemy odpowiednią operację wczytywania formatowanego:

❹ – Formatowane wyjęcie ze strumienia liczby

❺ – Formatowane wyjęcie ze strumienia stringu

Zauważ, że takie podglądanie było tutaj konieczne. Nie można po prostu wczytać pierwszego znaku, bo „zjedlibyśmy" kawałek zaczynającej się właśnie liczby lub nazwy.

Funkcja `putback`

Nawiązując do przedniego przykładu: jeśli mimo wszystko zamiast tylko podglądnąć – wyjęlibyśmy znak ze strumienia, to mamy ostatnią szansę go zwrócić. Służy do tego funkcja `putback` (ang. włóż z powrotem)

```
istream & putback(char) ;
```

Pomyślałeś pewnie: „–Co to oznacza? Klawiatura nie może przecież odszczekać tego, co na niej przed chwilą napisaliśmy!"

Rzeczywiście. Znak zwrócony wraca tylko do samego ujścia strumienia i tam czeka aż go ktoś nie wczyta.

To tak, jakbyśmy pracując w biurze krzyknęli „następny!" i do pokoju wszedł oczekujący w kolejce przed drzwiami interesant. Możemy mu jednak powiedzieć: „–Przepraszam pana, ale widzę, że nie jestem jeszcze gotowy pana przyjąć, proszę jeszcze zaczekać przed drzwiami." Interesant wychodzi i staje przy samych drzwiach jako pierwszy do obsłużenia.

Funkcja `putback` daje szansę oddania tylko jednego znaku. Gdybyśmy chcieli zrobić oszustwo: wczytać znak, podmienić go, a następnie tę podmienioną wartość oddać – wówczas efekt jest niezdefiniowany.

Oto przykładowe użycie:

```
char c, z ;
     cin >> c ;
     cout << "Przeczytany c="<< c << endl
     cin.putback(c) ;                    // wstawienie z powrotem
     cin >> z ;                          // wyjęcie po raz drugi
     cout << "Przeczytany z="<< z << endl
```

21.13.5 Funkcje wstawiające do strumienia

Poznamy teraz dwie funkcje składowe klasy `ostream` (strumień wyjściowy) odpowiedzialne za nieformatowane wypisywanie informacji.

Funkcja `ostream & put(char)`

Służy do wstawienia do strumienia jednego znaku. Przykładowo następujące instrukcje:

```
char napis[] = "Listopad" ;
int i = 0 ;
     do {
          cout.put(napis[i]).put('-') ;
     }while(napis[++i]);
```

spowodują, że na ekranie pojawi się

```
L-i-s-t-o-p-a-d-
```

Jeśli wstawiamy do strumienia, który płynie do pliku dyskowego, to czasem operacja ta może się nie udać – bo na przykład dysk się całkowicie zapełnił. W takich sytuacjach funkcja jako rezultat – zamiast zwracać referencję do strumienia, na którym pracuje – zwraca NULL.

Funkcja `write`

Funkcja `write` jest także funkcją składową klasy `ostream`. Służy ona do wstawienia do strumienia żądanej liczby bajtów. Inaczej mówiąc funkcja ta służy nam do wypisania czegoś na ekranie, albo do zapisania w pliku dyskowym. Oto deklaracja tej funkcji:

```
ostream & write(const char *skad_pisac, int ile);
```

Rezultatem zwracanym przez funkcję jest referencja strumienia, na którym ona pracuje.

- Pierwszy argument to wskaźnik do tablicy `char`, z której mają być pobierane bajty w celu wstawiania ich do strumienia. Przy wskaźniku widzimy słowo `const`, które oznacza zapewnienie, iż funkcja `write` zobowiązuje się niczego w tej tablicy nie zmieniać. Tylko sobie odczyta bajty, które ma wypisać.
- Drugi argument to liczba `int` określająca ile bajtów (począwszy od miejsca pokazywanego wskaźnikiem) należy do strumienia wstawić.

Przykładowo instrukcje:

```
char napis[] = "Rembrandt" ;
int i = 0 ;

          cout.write(napis+3, 4);
```

Spowodują wypisanie na ekranie tekstu

```
bran
```

Jest to, jak widać, sprytny sposób na wypisywanie fragmentu stringu. Prawdziwe zastosowanie tej funkcji, to wypisywanie do pliku dyskowego wielu tysięcy bajtów reprezentujących jakieś dane binarne – np. dane z układu pomiarowego, czy dane z urządzenia skanującego rysunek.

Podobnie jak przy funkcji `read`, także i tutaj dodam, że jeśli program ma wykonywać wyjątkowo wiele operacji pisania binarnego – dobrze jest rozważyć posłużenie się klasą `streambuf` i jej pochodnymi. Szczegółów musisz szukać w opisie implementacji biblioteki we/wy swojego kompilatora.

21.14 Operacje we/wy na plikach

Jeśli chcemy zapisywać coś do plików (np. dyskowych) lub czytać z nich, to mamy do dyspozycji klasy, które nam takie operacje zapewniają:

`ofstream` – (output file stream) – zapis do plików,

`ifstream` – (input file stream) – odczytywanie z plików,

`fstream` – (file stream) – oba powyższe.

Aby móc posłużyć się tymi klasami należy do programu włączyć plik nagłówkowy tego fragmentu biblioteki

```
#include <fstream.h>
```

Klasa `ofstream` jest pochodną klasy `ostream`, a klasa `ifstream` jest pochodną klasy `istream`. Natomiast klasa `fstream` jest pochodną od klasy `iostream`.

W zasadzie o tych faktach nie musimy pamiętać – ważny jest wniosek wynikający z fenomenu dziedziczenia:

Skoro klasy te są pochodnymi klas `istream` oraz `ostream`, to znaczy, że dziedziczą wszystkie cechy i zachowania swych klas podstawowych. Oznacza to w praktyce, że wszystko, co powiedzieliśmy do tej pory o strumieniach `iostream`, obowiązuje także i tutaj. Konkretnie: możemy także używać manipulatorów, operatorów >> <<, funkcji `get`, `put`, `read`, `write` itd.

Podstawowa różnica w pracy z tymi strumieniami polega na tym, że tutaj już strumienie nie są predefiniowane – czyli zdefiniowane za nas przez kompilator. Musimy sami sobie te strumienie zdefiniować – to oczywiste, bo przecież musimy wyraźnie określić do (od) jakiego pliku dyskowego strumień ma płynąć.

Zatem, aby czytać z pliku (lub pisać doń) należy:

- 1) Zdefiniować strumień czyli wykreować obiekt klasy `ifstream`, `ofstream`, lub `fstream`.
- 2) Określić strumieniowi z jakim konkretnie plikiem ma się komunikować i otworzyć ten plik.
- 3) Przeprowadzać żądane operacje we/wy.
- 4) Zlikwidować strumień, gdy uznamy, że praca z plikiem jest zakończona.

Jakie to proste zobaczmy na przykładzie – poniższy krótki program otwiera na dysku plik o nazwie "ksiezyc.tmp" i do tego pliku wpisywane jest słowo `misja`, po czym plik ten jest zamykany.

```
#include <iostream.h>
#include <fstream.h>
/*****************************************************/
 main()
{
ofstream osrodek ;                                  // etap ❶

     osrodek.open("ksiezyc.tmp") ;                  // etap ❷
     osrodek << "misja" ;                            // etap ❸
     osrodek.close();                               // etap ❹
}
```

W tekście programu zaznaczone są wspomniane wcześniej etapy pracy ze strumieniem:

❶ Definicja egzemplarza obiektu klasy `ofstream`. Obiekt ten nazywa się u nas „ośrodek" (niby od: ośrodek kontroli lotów).

❷ Poinformowanie ośrodka czym ma się zajmować – czyli do jakiego pliku ma płynąć strumień. Jest to po prostu wywołanie funkcji `open` będącej funkcją składową klasy `ofstream`.

❸ Dowolna liczba operacji wstawiania do strumienia `osrodek` (czyli pisania do pliku).

❹ Zamknięcie strumienia. Ponieważ strumień jest buforowany, więc następuje tu także ewentualne wypróżnienie bufora i zapisanie do pliku oczekujących tam jeszcze danych.

Etapy ❶ i ❷ zwykle łączy się w jeden instrukcją

```
ofstream osrodek("ksiezyc.tmp");
```

Jest to definicja obiektu połączona z wywołaniem konstruktora. Konstruktor ten zajmuje się od razu otwarciem podanego pliku. Oto jak wtedy wygląda odnośny fragment:

```
ofstream osrodek("ksiezyc.tmp") ;          // etap ❶ i ❷
          osrodek << "misja" ;             // etap ❸
          osrodek.close();                 // etap ❹
```

21.14.1 Otwieranie i zamykanie strumienia

Otwarcia strumienia możemy dokonać albo funkcją składową `open` †) albo za pomocą konstruktora. W obu wypadkach argumenty są takie same, zatem przyjrzyjmy się funkcji składowej `open`.

Funkcja open

```
void open(char* nazwa, int tryb = ***, int prot = filebuf::openprot);
```

Jeśli chodzi o **trzeci argument**, to reguluje on takie sprawy, jak to, kto – oprócz nas (właścicieli) – ma jeszcze pozwolenie z tego pliku korzystać.

Są to sprawy charakterystyczne dla konkretnego typu komputera i jego systemu operacyjnego – dlatego tym argumentem nie będziemy się zajmować. Korzystać będziemy z domniemanej wartości tego argumentu.

Pierwszy argument, to oczywiście nazwa pliku podana w postaci stringu.

Drugi argument określa tryb pracy z danym plikiem. Gwiazdki, jako argument domniemany, to oczywiście niemożliwe. To oszustwo musiałem zrobić dlatego, że domniemanie zależy od tego, czy mówimy o funkcji `open` z klasy `istream`, z klasy `ostream` czy z klasy `iostream`. Domniemania te są następujące:

w klasie	`ifstream`	`ios::in`
w klasie	`ofstream`	`ios::out`
w klasie	`fstream`	nie ma domniemania

Określeń trybów może być kilka, mogą być zastosowane pojedynczo lub po kilka równocześnie.

Oto możliwe określenia trybów otwarcia:

`in`	– (input) Otwórz plik do czytania
`out`	– (output) Otwórz plik do pisania
`ate`	– (at end) Otwórz i ustaw się na końcu zawartości
`app`	– (append) Otwórz do dopisywania

†) (czytaj: „ołpen")

`trunc` - (truncate) Otwórz, a jeśli plik istnieje – skasuj starą treść

`nocreate` - (no create) Otwórz jeśli plik już istnieje

`noreplace` - (no replace) Otwórz jeśli plik nie istnieje

`binary` - (binary) Tryb ma być binarny (domniemany jest tekstowy)

Określenia te zdefiniowane są jako publiczny typ wyliczeniowy `(enum)` w klasie `ios`. Klasa ta jest klasą podstawową także i dla klas odpowiadających za operacje na plikach. Ponieważ tryby otwarcia zwykle określamy będąc poza zakresem klasy strumień – dlatego też nazwy tych trybów poprzedzone są kwalifikatorem zakresu `ios::`

Przyjrzyjmy się dokładniej tym trybom

Przeważnie nie są one wykluczające się. Określają konkretne cechy jakie ma mieć nasz strumień.

ios::in

Informuje strumień, że chcemy otworzyć plik po to, by z niego czytać. Możliwe jest to tylko, gdy strumień dopuszcza operacje czytania (wyjmowania ze strumienia) – czyli jest to strumień klasy `ifstream` lub `fstream`.

ios::out

Informuje strumień, że chcemy otworzyć plik po to, by do niego zapisywać. Możliwe jest to tylko wtedy, gdy strumień dopuszcza operacje pisania (wstawiania do strumienia) – czyli jest to strumień klasy `ofstream`, `fstream`.

Uwaga:

> Otwarcie pliku w trybie `ios::out` bez dodatkowo trybu `ios::app` albo `ios::ate` zakłada także milcząco tryb `ios::trunc` – czyli stara treść pliku właśnie otwartego zostaje odrzucona, zaczynamy na pustym pliku.

Oba powyższe tryby `ios::in` oraz `ios::out` można zastosować równocześnie jeśli strumień jest klasy `fstream` – oznacza to, że z pliku będziemy czytali i do niego zapisywali. Np.

```
fstream strumyk("notatnik.txt", ios::in | ios::out);
```

Jak widać, takie połączenie dwóch trybów wyrażamy za pomocą operatora bitowej sumy '|'. (Ponieważ jest to operator OR dlatego często mówi się, że tryby mogą być „OR-owane").

ios::ate

(at end) – „na koniec" – określenie to oznacza, że chcemy, by po otwarciu pliku specjalny wskaźnik określający, w którym miejscu pliku właśnie pracujemy – ustawił się wstępnie na końcu pliku. Nie oznacza to wcale, że koniecznie musimy coś tam zapisywać. Plik może być równie dobrze otwarty do czytania,

a ustawiamy się na końcu po to, by za chwilę zrobić np. cztery kroki w tył i przeczytać czwarty bajt od końca pliku.

ios::app

(append – doczepiaj) określenie to oznacza, że wszelkie operacje zapisywania do pliku będą polegały wyłącznie na dopisywaniu do jego końca. (Jeśli użyjemy określenia tego trybu to niejawnie pociąga to za sobą tryb `ios::out`).

ios::trunc

(truncate – ang. obetnij) -ten tryb wybieramy, gdy nie interesuje nas dotychczasowa zawartość otwieranego pliku. Po otwarciu go, stara treść jest odrzucana i do pliku można zapisywać tak, jakby był to świeżo wykreowany plik. Jeśli takiego pliku do tej pory nie było – to jest tworzony.

Tryb ten jest milcząco zakładany, gdy zastosowaliśmy tryb `ios::out` bez dodatkowej specyfikacji `ios::app` albo `ios::ate`.

ios::nocreate

no create[†)] – ang.: nie stwarzaj. Tryb ten zabrania otwarcia nowego pliku. Jeśli plik już istniał, to może być otwarty; jeśli natomiast nie istniał, to operacja otwarcia nie powiedzie się.

ios::noreplace

no replace[††)] – ang.: nie zastępuj. Żądanie by plik został otwarty tylko wtedy, gdy nie istnieje jeszcze plik o takiej nazwie. W przeciwnym razie operacja otwarcia nie powodzi się.

ios::binary

binary[†††)] – ang.: binarny. Ten tryb nie występuje w niektórych wersjach biblioteki. (Np. jest w Borland C++, a nie ma w Sun C++). Jest to życzenie jak mają być traktowane dane płynące strumieniem: jako dane binarne czy tekstowe. (Domniemany jest tryb tekstowy).

W zasadzie strumieniowi jest wszystko jedno czy płyną nim dane binarne czy tekstowe. Zawsze przecież chodzi o przesłanie jakichś bajtów. Niezależnie od tego w jakim trybie otworzymy plik – można użyć równie dobrze funkcji do przesłania znaku np. `get(char)` czy `put(char)` jak i funkcji do czytania/pisania binarnego `write` lub `read`.

Jest tylko jedno „ALE". Jeśli otwieramy plik w trybie tekstowym wówczas:

- w wypadku wstawiania do strumienia (pisanie do pliku): każde wystąpienie znaku `'\n'` zostaje zamienione na sekwencję dwóch znaków: `'\r'` i `'\n'`

†) (czytaj: „noł kriejt")
††) (czytaj: „noł ryplejs")
†††) (czytaj: „bajnery")

- w wypadku wyjmowania ze strumienia (czytanie pliku) sekwencja `'\r'` i `'\n'` zostaje zamieniona na znak `'\n'`

w skrócie:

czytanie: `'\r'+'\n'` ⟶ `'\n'`

pisanie: `'\n'` ⟶ `'\r' + '\n'`

W wypadku trybu binarnego nie robi się żadnych takich konwersji.

Dobrze jest o tym pamiętać. Jeśli zapomnisz i będziesz chciał zapisać w pliku kilka bajtów reprezentujących dane binarne, a strumień będzie pracował w trybie tekstowym

```
char rysunek[] = { 1,32,10, 17, 4 } ;
ofstream osrodek("tmp.tmp", ios::out) ;
          osrodek.write(rysunek, 5);
          osrodek.close();
```

to w rezultacie do pliku zostanie zapisana taka treść binarna:

```
1, 32, 13, 10, 17, 4
```

Jak widzisz pojawiła się dodatkowa trzynastka. To dlatego, że plik otwarty był (przez domniemanie) w trybie tekstowym. Strumień zobaczywszy bajt 10 — który jest identyczny jak kod ASCII znaku `'\n'` dołożył w prezencie znak `'\r'` (kod ASCII = 13). Strumień sądził, że pracujemy w trybie tekstowym, więc chciał się przysłużyć. Winni jednak jesteśmy my sami, bo powinniśmy go uprzedzić, że pracujemy na danych binarnych, a nie tekstowych.

W powyższym przykładzie zobaczyliśmy użycie funkcji `close`, która kończy pracę strumienia z danym plikiem.

Wielokrotne otwieranie strumienia

Strumień wejściowy lub wyjściowy po zakończeniu pracy z jakimś plikiem może posłużyć do pracy z innym. Nic w tym dziwnego – ośrodek kontroli lotów w Huston po zakończeniu jednej misji promu kosmicznego może obsłużyć inną.

W praktyce odbywa się to w ten sposób, że po wykonaniu wszystkich operacji we/wy z danym plikiem, plik jest zamykany, a potem otwierany jest inny.

Oto przykład:

```
ofstream strum("raz.tmp") ;
     strum << "do jednego zbioru \n" ;
     strum.close() ;

     strum.open("dwa.tmp");
     strum << "do drugiego zbioru" ;
     strum.close () ;
```

Strumień `strum` służy nam pierwotnie do pisania w pliku "raz.tmp". Następnie funkcją składową `close` zamykamy ten plik i komunikację z nim, po czym za

pomocą funkcji open otwieramy nowy plik. Tym samym sprawiamy, że strumień „popłynie" do pliku o nazwie "dwa.tmp".

Oczywiście strumienia można ponownie użyć tylko do celów, do których został przeznaczony – czyli jeśli jest strumieniem wyjściowym, to nadal musi się zajmować tego typu operacjami. Strumień wejściowy też musi pozostać wejściowym. Słowem: strumień jest nadal tej samej klasy – zmienia się tylko plik, który obsługuje.

21.15 Błędy w trakcie pracy strumienia

Nie zawsze otwarcie pliku się udaje – przykładowo: chcemy otworzyć do czytania plik, który nie istnieje. Nie zawsze też muszą się udać operacje we/wy – np. chcemy zapisać coś do pliku dyskowego, a na dysku nie ma już miejsca. Inaczej mówiąc zażądaliśmy operacji we/wy, a ona nie mogła zostać wykonana poprawnie. Problem poprawności operacji we/wy nie pojawia się dopiero przy operacjach z plikami. Nasze dotychczasowe przykłady ze strumieniami cin, cout też powinny zawierać miejsca, w których sprawdzamy poprawność operacji we/wy. Oto znany przykład: jeśli strumień cin oczekuje na wczytanie liczby, a tymczasem na klawiaturze ktoś wystuka coś, co liczbą nie jest — wówczas wczytanie liczby nie może nastąpić. W naszych programach przykładowych mogliśmy się bez sprawdzania poprawności obejść, lecz w programie, który piszemy dla publicznego użytku, należy się spodziewać i takich sytuacji, że użytkownik pomyli się. Wniosek jest prosty:

> Do dobrego stylu programowania należy kontrola poprawności właśnie wykonanych operacji na strumieniach.

W następnych paragrafach porozmawiamy o narzędziach, za pomocą których łatwo to robić.

Takie narzędzia istnieją – zebrane zostały w klasie ios. Z klasą tą spotkaliśmy się już przy omawianiu flag stanu formatowania. Jest ona klasą podstawową dla wielu klas strumieni – także dla ifstream, ofstream, oraz fstream, zatem owe narzędzia zostają do nich odziedziczone.

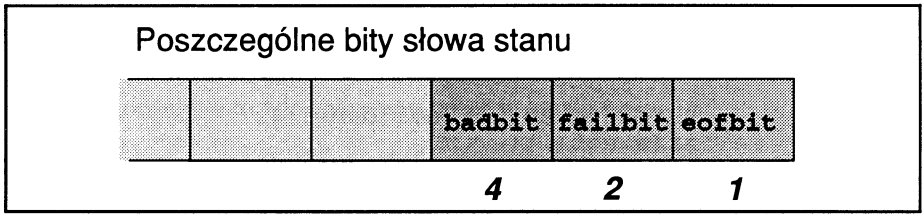
Przyjrzyjmy się bliżej tym narzędziom:

21.15.1 Flagi stanu błędu strumienia

W ośrodku dowodzenia pracą strumienia – czyli w każdym obiekcie klasy strumień – jest słowo odpowiadające za stan błędu strumienia. Znaczy to, że w sytuacji, gdy następuje jakiś błąd pracy strumienia – w słowie tym ustawiany jest bit odpowiadający za daną kategorię błędu. Te kategorie błędów określone są typem wyliczeniowym zdefiniowanym w klasie ios

```
enum io_state {
     goodbit   = 0,
     eofbit    = 1,
     failbit   = 2,
     badbit    = 4
} ;
```

Przedstawić to można tak:



Oto znaczenie poszczególnych flag:

goodbit -

widzimy, że w typie wyliczeniowym `io_state` jej wartość jest 0. Czyli tak naprawdę – nie jest to żadna flaga błędu. Mówimy, że stan `goodbit` jest wtedy, gdy wszystkie bity stanu błędów są wyzerowane;

eofbit -

ustawiana jest wtedy, gdy podczas czytania napotkany został koniec pliku (EOF);

failbit -

ustawienie tej flagi oznacza, że jakaś operacja we/wy nie powiodła się. Strumień tkwi w stanie błędu, jednak po wyzerowaniu tej flagi nadal nadaje się do pracy;

badbit -

ustawienie tej flagi oznacza, że nastąpił jakiś poważny błąd. Dalsza praca na tym strumieniu jest niemożliwa.

21.15.2 Funkcje do pracy na flagach błędu

Zwykle flagami błędu strumienia nie zajmujemy się bezpośrednio. Do pracy z nimi klasa `ios` definiuje kilka funkcji składowych informujących nas o stanie strumienia. Oto one:

int good()

- ta funkcja zwraca wartość niezerową jeśli wszystko jest w porządku, czyli żaden z bitów błędu nie jest ustawiony, np.

```
char znak ;
do{
     cin >> znak ;
     cout << znak ;
}while(cin.good() ) ;
```

Pętla będzie się wykonywała dopóki nie nastąpi błąd strumienia `cin`.

int eof()

- funkcja ta zwraca wartość niezerową jeśli jest ustawiona flaga `ios::eofbit`, czyli przy operacji wczytywania napotkany został koniec pliku, np.

```
char dane[1000];

ifstream s("wtorek.tmp" , ios::in|ios::binary);
// na razie pomijam sprawdzenie poprawności otwarcia pliku

s.read(dane, sizeof(dane) );
if(s.eof()){
     cout <<   "Napotkany EOF przed wczytaniem "
               "wszystkich danych" << endl ;
}else {
     cout <<   "Wszystko wczytane O.K. " <<endl ;
}
```

Powyższy przykład pozwala na wykrycie czy z pliku wczytała się żądana ilość danych.

int fail()

- funkcja ta zwraca niezerową wartość, gdy `failbit` lub `badbit` są ustawione. Na przykład w sytuacji

```
int liczba ;
     cin >> liczba ;
     if(cin.fail()){
          cout << "Zle podana liczba !" << endl ;
     }
```

flaga `failbit` zostaje ustawiona wtedy, gdy zamiast oczekiwanej liczby z klawiatury przypłynie strumieniem tekst.

int bad()

- zwraca wartość niezerową, gdy `badbit` jest ustawiony. Tak zdarzy się na przykład wtedy, gdy chcemy by strumień wczytywał z pliku, który nie istnieje. Np.:

```
ifstream list("wtorek.dat", ios::in);
if(list.bad()){
     cout << "Blad otwarcia pliku 'wtorek.dat' "
          << endl ;
     // reakcja na to...
}
```

21.15.3 Kilka udogodnień

Operator !

Powyżej poznaliśmy funkcje, za pomocą których można było sprawdzać poprawność właśnie dokonanych operacji we/wy. Sprawdzenie takie polegało na wywołaniu jednej z funkcji składowych strumienia. Istnieje jednak wygodniejszy sposób. Jest to także wywołanie jednej z tych funkcji, jednak w niezauważony sposób. Jak to jest zrobione? Otóż dla wygody w klasie `ios` jest także zdefiniowany operator ! (wykrzyknik), który robi to samo, co wywołanie funkcji `fail`.

Dwa poniższe zapisy są zatem równoważne

```
cin >> liczba ;

if(cin.fail() )     cout << "Niepowodzenie " << endl ;
if(!cin)            cout << "Niepowodzenie " << endl ;
```

Inne udogodnienie to zdefiniowany także w klasie `ios`

operator konwersji na typ `void*`

```
operator void*()
```

Operator ten zwraca wartość niezerową (że tak powiem: nie-NULL-ową), gdy nie jest ustawiana żadna flaga błędów. Natychmiast nasuwa się identyczne zastosowanie

```
cin >> liczba ;
if(cin)   cout << "Sukces !!' " ;
else      cout << "Niepowodzenie " ;
```

Konwencja

Pamiętamy też, że rezultatem wyrażenia

```
cin >> liczba
```

(jako całości) jest referencja do strumienia `cin`. Zatem poprawność tej operacji można sprawdzić też tak:

```
if(cin >>liczba) cout << "Sukces " ;
```

Zauważ różnicę: funkcje `fail` i `good` jako rezultat zwracają typ `int`, natomiast operator >> (lub <<) zwraca referencję do strumienia – mimo tej różnicy w obu sytuacjach można sprawdzić poprawność operacji instrukcją `if`.

Jednak tak naprawdę, to nie chodzi wcale o dublowanie operatora ! (wykrzyknik). Jest w tym głębszy sens. Zauważ:

> Wiemy już, że jeśli operacja wczytania nie powiedzie się, to wartością wyrażenia
>
> `(cin>>liczba)`
>
> nie będzie już referencja do strumienia `cin`, ale NULL

zobacz teraz, co będzie w przypadku takiej „kaskadowej" instrukcji wczytującej

```
cin >> liczba1 >> liczba2 ;
```

Inaczej można to zapisać jako

```
(cin >> liczba1 ) >> liczba2 ;
```

Jeśli operacja wczytania pierwszej liczby się nie powiedzie, to mamy przed sobą taką instrukcję

```
(NULL) >> liczba2 ;
```

Oznacza to, że jeśli pierwsza operacja się nie powiodła, to nie zostanie podjęta druga. Jest to bardzo ważne spostrzeżenie. Tak ważne, że uznane jest za konwencję.

Zachowania strumieni są zdefiniowane tak, że jeśli w danym strumieniu nastąpi błąd, to dalsze próby wstawiania lub wyjmowania z tego strumienia są ignorowane.
Ewentualne dodatkowe próby wstawiania lub wyjmowania nie zlikwidują jego stanu błędu. Nie zlikwidują, ale mogą jeszcze jakąś flagę błędu dodatkowo ustawić.

Tę konwencję powinniśmy znać – by wiedzieć czego się spodziewać w danej sytuacji. Z drugiej strony konwencji tej powinniśmy przestrzegać w sytuacjach, gdy sami piszemy swoje własne operatory wczytywania i wypisywana.

21.15.4 Bardziej wyszukane operacje na flagach błędu strumienia

W tym paragrafie mówić będziemy o tym, jak samemu decydować co jest poprawną, a co niepoprawną operacją wczytania/wypisania. Przydaje się to w wypadku operacji we/wy z obiektami typu zdefiniowanego przez użytkownika.

Sprawy te nie są trudne, ale na pewno nie będziesz się nimi zajmować od razu. Dlatego proponuję przy pierwszym czytaniu przeskoczyć ten paragraf.

Własnoręczne ustawianie flag błędu

Dotychczas do informowania się o stanie flag błędu wystarczały nam funkcje

```
good, fail, eof, bad
```

W klasie `ios` są jeszcze dodatkowe funkcje pracujące na flagach stanu błędu

```
int rdstate();
void clear(int = 0);
```

Pierwsza z nich daje nam jako rezultat słowo stanu błędu danego strumienia, natomiast druga z nich pozwala zmieniać wybrane flagi błędu. Przyjrzyjmy się im bliżej.

Funkcja `rdstate`

- (czyli read state – odczytaj stan) zwraca jako rezultat słowo `int`, w którym odpowiednie bity odpowiadają flagom stanu błędu. (Z definicji typu wyliczeniowego `io_state` wiemy jak nazywają się poszczególne flagi i które bity zajmują).

Funkcja ta właściwie nie służy temu, by nas poinformować o stanie flag. Prościej mogliśmy się tego dowiedzieć stosując funkcje `bad`, `fail` itd. Prawdziwie jej zastosowanie jest takie, że dzięki niej możemy poznać całe bieżące słowo stanu po to, by w jakiś sposób pozmieniać je – i tę zmienioną wartość umieścić z powrotem w ośrodku dowodzenia strumieniem - za pomocą funkcji `clear`.

Funkcja `clear`

wywoływana jest z jednym argumentem typu `int`, a zwraca typ `void` (nic). Argument jest domniemany i domniemanie jest 0. Działanie tej funkcji polega na tym, że jej argument zastępuje dotychczasowe słowo stanu błędu strumienia. Jak już wspomniałem wyżej, jest to moment kiedy „podrzucamy kukułcze jajo". Strumień ma od tej pory nowe słowo stanu błędu.

Celowo użyłem tak obrazowego porównania o kukułczym jaju, bo nie chcę byś miał jakiekolwiek skojarzenia z nazwą `clear`. Ta nazwa (kasuj, zeruj) jest bardzo niefortunnie dobrana. Funkcja ta, co prawda kasuje wszystkie flagi stanu błędu, gdy wywołamy ją tak

```
cin.clear(0) ;          // np. dla strumienia cin
cin.clear();
```

natomiast, gdy wywołamy ją z argumentem niezerowym, wówczas niektóre flagi zostaną skasowane, ale inne ustawione. Przykładowo instrukcja

```
cin.clear(ios::eofbit) ;
```

ustawia w strumieniu `cin` flagę `ios::eofbit`, a wszystkie inne kasuje.

Jeśli chcemy ustawić dwie flagi, to nie możemy użyć kolejnych dwóch instrukcji

```
cin.clear(ios::eofbit);
cin.clear(ios::failbit);
```

dlatego, że druga z nich zniszczy to, co ustawiła pierwsza. Poprawna forma to „OR-owanie" czyli suma bitowa

```
cin.clear(ios::eofbit | ios::failbit);
```

Powyższa instrukcja ustawia dwie żądane flagi, a wszystkie inne kasuje.

Jak ustawić jedną lub dwie flagi błędu bez wpływu na resztę ?

Załóżmy, że chodzi o flagę `ios::eofbit` w strumieniu o nazwie `wejscie`

```
int r ;
    r = wejscie.rdstate() ;
    r |= ios::eofbit ;          // czyli r = r |ios::eofbit;
    wejscie.clear(r) ;
```

To samo zwięźlej

```
wejscie.clear( wejscie.rdstate() | ios::eofbit );
```

Jak widać i tutaj funkcja `clear` nie służy naprawdę kasowaniu flag (niczego tu nie kasuje) tylko sensownemu ustawieniu nowej flagi.

Zastosowanie

Ustawianiem flag stanu błędów nie musimy się zwykle zajmować – robione jest to za nas automatyczne. Są jednak sytuacje, gdy może nam się to okazać przydatne. Pokażemy to na przykładzie. Istota tego przykładu jest taka: mamy operator wczytywania, który wczytuje pisane na klawiaturze wyrazy; tymczasem nie wszystkie podane wyrazy uznajemy za poprawne - wolno napisać

tylko takie, które zaczynają się od małej litery e. Jeśli wyraz zaczyna się od innej litery, to uznajemy to za błąd.

Oto przykład, gdzie występuje klasa, której obiekty służą do przechowywania słów. Definiujemy dla niej operator wczytywania (wyjmowania ze strumienia). Operator ten powinien wczytywać jeden wyraz pod warunkiem, że będzie on na literę e. W przeciwnym razie powinien ustawić flagę `failbit`.

```
#include <iostream.h>
#include <fstream.h>

class slowo_na_e {
public:
      char wyraz [80] ;
} ;
//////////////////////////////////////////////////////////
istream & operator >>(istream & str, slowo_na_e & w)
{
     str >> w.wyraz ;
     if(w.wyraz[0] != 'e'){
          str.clear(str.rdstate() | ios::failbit ) ; // ❶
     }
     return str ;
}
/*****************************************************/
main()
{
slowo_na_e pierwsze, drugie, trzecie ;

 while(1){
          cout << "Podaj trzy slowa na litere 'e' : " ;
          cin >> pierwsze >> drugie >> trzecie ;      //❷

     if(!cin) {
          cout << "\nZle ! Od poczatku : " ;
          cin.clear(cin.rdstate() & ~ios::failbit);  // ❸
     }
     else break ; // jeśli poprawnie
 }
}
```

W przypadku wykonania programu ekran może wyglądać na przykład tak:

```
Podaj trzy slowa na litere 'e'
era
zlosliwiec

Zle ! Od poczatku : Podaj trzy slowa na litere 'e'
era
europa
estragon
```

Komentarz

❶ Jest to zwykła realizacja przeładowania operatora >> . Ciekawe w niej jest tylko to, że badamy czy istotnie pierwszą literą przyjętego wyrazu jest `'e'`. Jeśli nie, to ustawiamy flagę `ios::failbit`.

❷ Trzy „kaskadowo" ustawione operacje wczytania. Jeśli w trakcie jednej z nich ustawiliśmy flagę `ios::failbit`, to dalsze usiłowania wyjęcia czegoś z tego strumienia nie będą podejmowane. Tak będzie dopóki strumień tkwi w stanie błędu – czyli dopóki flaga jest ustawiona.

❸ Jeśli po napisaniu ostrzeżenia chcemy użytkownikowi programu dać szansę poprawienia się, to likwidujemy stan błędu w tym strumieniu – kasujemy flagę `ios::failbit`. Innych flag błędu nie zmieniamy, zerujemy tylko tę jedną. Zauważ, że stosujemy tu iloczyn bitowy z wartością zanegowaną.
Gdybyśmy zapomnieli i nie zlikwidowali tego stanu błędu, to żadna dalsza próba wyjęcia czegoś ze strumienia `cin` nie byłaby podejmowana.

Wniosek:

> Mając możliwość zmian flag stanu błędów możemy definiować inteligentniejsze operatory << >> – takie, które zareagują na to, co my sami określamy jako poprawne lub niepoprawne.

21.15.5 Trzy plagi - czyli „gotowiec" jak radzić sobie z błędami

Ten paragraf nie wprowadza niczego nowego. Jednak postanowiłem go dodać dlatego, że sam najlepiej wiem, jak to bywa. Czasem trzeba szybko napisać parę linijek kodu pracującego z plikiem – i to tak, by obsłużyć ewentualne błędy mogące pojawić się w tej pracy. Ktoś, kto chce to zrobić szybko - nie ma ochoty jeszcze raz czytać kilku stronic tej książki, by sobie przypomnieć parę drobiazgów. Właśnie na okoliczność takich sytuacji przygotowałem „gotowca". Przykładowy program, w którym pracujemy z plikami i zdarzają się nam same nieszczęścia.

```
#include <iostream.h>
#include <fstream.h>

main()
{
     ifstream strum ;     // def strumienia do pracy z plikiem
     char nazwa_pliku[50] ;

     // --------otwieranie pliku może ustawić flagę błędu ios::badbit
     // w razie gdy plik nie istnieje. Strumień tkwi wówczas w stanie błędu.
     // Aby ponowić próbę otwarcia pliku flagę tę trzeba najpierw skasować

     for(int sukces = 0 ; !sukces ; )
     {
          cout << "Podaj nazwe pliku: " ;
          cin >> nazwa_pliku ;
          cout << endl ; // kosmetyka ekranu
```

```
                                        // próba otwarcia
        strum.open(nazwa_pliku, ios::in);                    // ❶

                                        // czy sie udało ?
        if(!strum)                      // czyli inaczej: strum.fail()
        {
             cout <<   "Blad otwarcia pliku: "
                  << nazwa_pliku << endl ;
                  // Skoro próba otwarcia nie udała się to strumień
                  // jest w stanie błędu !!!";
                  //musimy usunąć stan poważnego błędu strumienia
             strum.clear(strum.rdstate() & ~ios::badbit);

                  // strumień już jest w porządku. W kolejnym obiegu pętli
             cout << "Ponowiamy probe...\n";
        }
        else
        {                               // udało się otworzyć,
             sukces = 1 ;               // więc pętlę można zakończyć
        }
   }                                    // koniec pętli for

// ---operacje czytania mogą wywołać ustawienie flagi błędu ios::eofbit
// w wypadku, gdy dojdziemy do końca pliku i mimo to próbujemy
// czytać nadal.
// Aby dalej pracować z tym strumieniem musimy skasować tę flagę błędu
// ( a potem ewentualnie ustawić kursor czytania  w poprawne miejsce)

int numer ;
char znak ;

for(int sukces2 = 0 ; !sukces2 ; )
{
     cout << "Podaj numer bajtu ktory "
              "chcesz poznac: " ;
     cin >> numer ;
                    // pozycjonujemy kursor czytania na tym bajcie
     strum.seekg(numer);

     znak = strum.get() ;                                    // ❷
     if(strum.eof() )
     {
          cout << "Blad pozycjonowania, "
               "prawdopodobnie plik\n\t jest krotszy"
               " niz " << numer << " bajtow\n" ;
                    // strumień tkwi w stanie błędu ios::eofbit,
                    // trzeba go skasować
          strum.clear(strum.rdstate() & ~ios::eofbit);
          cout << "(Podaj mniejsza liczbe)\n" ;
                    // będzie ponowny obieg pętli
     }
     else {
          sukces2 = 1 ;
          cout << "Ten bajt to ASCII: '" << znak
               << "' hexadecymalnie " << hex
               << (int)znak << endl ;
```

```
            }
        }
        // ------------------próba formatowanego wczytania liczby w sytuacji,
        // gdy właśnie oczekuje na wczytanie coś, co liczbą nie jest, - wprowadzi
        // sturmień w stan błędu ios::failbit

        int liczba ;
                        // instrukcja wczytania liczby - rozpocznie czytać zaraz po
                        // wczytanym poprzednio bajcie
        strum >> liczba ;                                        // ❸
                                // czy się udało ?
        if(strum.fail() )
        {                       // nie !
            cout <<   "Blad failbit, bo najblizsze "
                      "bajty\n\t nie moga "
                      "byc wczytane jako liczba\n" ;

                                // Aby to coś wówczas wczytać jako string
                                // należy najpierw skasować ustawioną flagę błędu
            strum.clear(strum.rdstate() & ~ios::failbit);

                                // strumień nadaje się już do pracy, a to coś, co go
                                // zatkało, nadal czeka na wczytanie
            char slowo[80] ;
            strum >> slowo ;
            cout << "Jest to slowo: "<< slowo << endl;

        }else{
                                                // tak !
            cout << "Pomyslnie wczytana liczba: "
                 << liczba << endl;
        }
        // dalsza praca z plikem ....
}
```

Na ekranie zobaczymy na przykład taki tekst

```
Podaj nazwe pliku: nic.nic
Blad otwarcia pliku: nic.nic
Ponowiamy probe...
Podaj nazwe pliku: plagi.cpp
Podaj numer bajtu ktory chcesz poznac: 9999
Blad pozycjonowania, prawdopodobnie plik jest krotszy
niz 9999 bajtow
(Podaj mniejsza liczbe)
Podaj numer bajtu ktory chcesz poznac: 3
Ten bajt to ASCII: 'c' hexadecymalnie 63
Blad failbit, bo najblizsze bajty
     nie moga byc wczytane jako liczba
Jest to slowo: lude
```

Komentarz

Program ten pokazuje jak przykładowo radzić sobie z trzema sytuacjami, gdy następuje błąd pracy strumienia. Te trzy sytuacje oddzielone są dużym komen-

tarzem. Tekst programu obficie opatrzyłem komentarzami, dlatego teraz opiszę to lakonicznie:

❶ Otwarcie pliku. Za pierwszym razem odpowiadam, że chcę otworzyć plik nic.nic, który na moim dysku nie istnieje. Strumień wchodzi w stan błędu, ale ja ten błąd usuwam i ponawiam próbę. Za drugim razem podaję nazwę pliku plagi.cpp, która jest nazwą tekstu tego programu - a więc pliku, który na pewno istnieje.

❷ Mając otwarty plik pozycjonuję kursor czytania. (O pozycjonowaniu mówimy parę stron dalej - mam nadzieję, że korzystając z tego "gotowca" przeczytałeś już choć raz ten rozdział). Ponieważ jestem przewrotny każę pozycjonować daleko za końcem pliku. To oczywiście przy próbie wczytania - wywołuje natychmiast błąd. I tym razem się nie przejmuję, tylko kasuję flagę błędu. Za drugim razem pozycjonuję sensownie. Strumień już jest w poprawnym stanie więc próba wczytania bajtu powodzi się. Dla pewności pozycjonowałem gdzieś na początku pliku - i widzisz na ekranie, że wczytałem znak z wnętrza słowa „`include`".

❸ - Próbuję wczytać liczbę z pliku. Czytanie oczywiście odbywa się z miejsca, na którym poprzednio skończyliśmy, a tam, jak pamiętamy, nie było żadnej liczby. Wywołuje to więc błąd. Gdy flagę błędu skasujemy, możemy spróbować przeczytać te bajty, już nie jako liczbę, a jako string. Wczytanie się powodzi — widzimy na ekranie dalszą (niewczytaną jeszcze) część słowa `include`.

21.16 Przykład programu pracującego na plikach

Zobaczymy teraz program, który kopiuje treść jednego pliku do drugiego. Tytułem dygresji dodam, że nie jest to tylko program wymyślony dla ilustracji tego rozdziału. Program ten kilkakrotnie ratował mnie z kłopotów.

```
#include <iostream.h>
#include <fstream.h>
/*****************************************************/
main()
{
char plik_a[80] ;
char plik_b[80] ;

     //---------------------------------------------
     cout << "Podaj nazwe pliku wejsciowego : " ;
     cin >> plik_a ;
     ifstream czyt(plik_a) ;                          // ❶
     if(!czyt){
          cout << "Nie moge otworzyc takiego pliku ";
          return 1 ;
     }
     //---------------------------------------------
     cout << "Podaj nazwe pliku wyjsciowego : " ;
     cin >> plik_b ;

     ofstream pisz(plik_b) ;                          // ❷
     if(!pisz){
```

```
            cout << "Nie moge otworzyc takiego pliku ";
            return 1 ;
        }
        // ----- akcja przepisywania ----------------------
        char c ;
        while(czyt.get(c)){                                  // ❸
            if(!pisz.put(c) ){                               // ❹
                cout << "Blad pisania ! \n" ;
                break ;
            }
        }
        // -------- koniec, spr powód zakończenia ---------
        if(!czyt.eof() ){                                    // ❺
                cout << "Blad czytania\n" ;
        }
}
```

Komentarz

❶ Definicja obiektu klasy strumień wejściowy `ifstream`. Strumieniowi temu nadajemy nazwę `czyt`. Widzimy tu od razu wywołanie konstruktora otwierającego plik o podanej nazwie. Jest to jedyny argument konstruktora — drugi argument nie musi być obecny, bo dla strumienia klasy `ifstream` jest domniemanie, że drugi argument to `ios::in`

❷ Definicja obiektu klasy „strumień wyjściowy" `ofstream`. Strumień płynie do pliku o nazwie podanej jako pierwszy argument konstruktora. Drugi argument jest domniemany, a domniemanie to dla strumienia klasy `ofstream` brzmi `ios::out`.

❸ Przeczytanie jednego bajtu ze strumienia wejściowego. Od razu sprawdzana jest poprawność operacji czytania. Gdyby czytanie nie powiodło się, to wartością wyrażenia w nawiasie byłoby NULL i pętla `while` byłaby przerwana.

❹ Zapis przeczytanego bajtu do zbioru wyjściowego ze sprawdzeniem poprawności pracy strumienia wyjściowego `pisz`.

❺ Za jedyny poprawny sposób wyjścia z powyższej pętli `while` uznajemy ustawienie flagi `ios::eofbit`, czyli napotkanie końca czytanego pliku. Jeśli zaś powód jest inny - to ogłaszamy błąd.

Jak widać, jest to zwykły program kopiujący treść pliku. Mówiłem jednak, że w programie jest coś szczególnego, co wybawiło mnie już kilkakrotnie z kłopotów. Na pierwszy rzut oka jednak tego nie widać.

Jeśli jesteś ciekaw, to posłuchaj tej historii.

Przytaczam ją dlatego, że możliwe, iż przytrafi się także Tobie.

Jeśli posługiwałeś się kiedyś programem uruchomieniowym (debuggerem), to wiesz, że dzięki niemu możesz wykonywać program krokowo – linijka za linijką. Debugger wykonuje program krokami, a żebyśmy widzieli gdzie w danym momencie jest – używa tekstu źródłowego naszego programu.

Wyświetla na ekranie określony fragment i za pomocą strzałki umieszczonej przy danej linijce (lub podświetlenia tej linijki) oznajmia: „tu właśnie pracuję".

Otóż posługując się debuggerem zauważyłem, że czasem, debugger pokazywał błędnie – strzałka była przy innej linijce niż powinna, najczęściej przy sąsiedniej. Jeśli ta sąsiednia linijka zawierała tylko komentarz, to łatwo było ustalić, że debugger oszukuje, bo przecież nie może tam nic wykonywać (nie zawsze jednak sytuacja jest tak jasna).

Długo szukałem przyczyn takiego zachowania debuggera, wreszcie doszedłem do wniosku, że debugger jest zdezorientowany przez niedopasowanie znaków `'\r'` i `'\n'`. Znaki te kończą każdą linijkę tekstu programu. Teoretycznie zawsze na końcu linijki powinny być oba te znaki – czasem jednak w pewnych niewyjaśnionych dla mnie okolicznościach - zgubiony był jeden z nich. Nie każdy edytor pokazuje taki tekst błędnie – dlatego wszystko w edytorze wyglądało niewinnie.

Tymczasem debugger chcąc powiedzieć mi, że wykonuje właśnie linijkę 229 odliczał 229 par znaków `'\r'` i `'\n'` i w tekście źródłowym mojego programu pokazywał, że tę właśnie linijkę wykonuje. Jeśli z jakichś powodów zamiast pary `'\r'` i `'\n'` był tylko sam znak `'\n'` to debugger był zdezorientowany i gubił się w odliczaniu (o właśnie tę linijkę).

Zmusiło mnie to do napisania programu: „Korektor". Program ten wczytuje plik i robi jego kopię przy okazji naprawiając ewentualne braki. Jego tajemnica polega na tym, że oba pliki otwierane są w trybie tekstowym. Oznacza to, że wczytując plik każda sekwencja `'\r'` i `'\n'` zostaje zamieniona na jeden znak `'\n'` a w trakcie wypisywania do pliku wyjściowego wszystkie znaki `'\n'` zamienią się na pary `'\r'` i `'\n'`.

Debugger pracując z takim skorygowanym plikiem źródłowym już się nie mylił.

Tego samego efektu naprawienia nie osiągnąłbym za pomocą komendy systemu operacyjnego (np. komenda copy). To dlatego, że taka komenda kopiuje pliki binarnie. Robi się więc identyczną kopię. Razem z tkwiącym tam błędem.

21.17 Wybór miejsca czytania lub pisania w pliku

Pisanie znaków (czy ogólniej bajtów) do pliku, to nie jest wrzucanie ich do dużego worka o określonej nazwie. Jest to bowiem „plik". Określenie *plik* sugeruje, że informacja jest tam umieszczona w jakiejś kolejności.

> *Muszę wyznać ze skruchą, że sam posługuję się starszym określeniem „zbiór" – wiedząc jednak, że nie jest to dobre tłumaczenie angielskiego „file".*

Zapis do pliku następuje zawsze w jakieś określone miejsce. Pisząc tę książkę stosuję zasadę, że kolejne litery umieszczam na samym końcu tego, co napisałem do tej pory. Mam więc jakby w głowie jakiś wskaźnik, który mówi mi: tu należy postawić ewentualną następną literę.

Podobnie Ty czytelniku – czytając tę książkę masz jakby pewien wskaźnik, który mówi: „tu właśnie czytam". Jeżeli zadzwoni dzwonek do drzwi, to zaznaczasz to miejsce zakładką by wiedzieć, gdzie zacząć czytać potem. Wskaźnik ten w trakcie czytania cały czas się zmienia. Ten wskaźnik, to po prostu numer litery

w książce. Opisana sytuacja nie jest jednak jedynym sposobem czytania. Jeśli weźmiesz do ręki encyklopedię w celu wyjaśnienia słowa „Medyceusz" – to nie będziesz czytał encyklopedii od pierwszej strony z literą A aż do momentu, gdy natkniesz się na to hasło. Najpierw odszukasz sobie to miejsce, gdzie jest interesująca Cię informacja. Inaczej mówiąc określisz pozycję wskaźnika, gdzie zacząć czytać. Po przeczytaniu wyjaśnienia tego hasła może znajdziesz odsyłacz na hasło „renesans" zatem znowu pozycjonujesz wskaźnik (przewracasz strony) i zaczynasz czytać właściwą rzecz.

Nie potrzebuję chyba wyjaśniać, że tak samo jest w wypadku pisania. Nie muszę koniecznie ciągle coś dopisywać na końcu tekstu. Mogę przecież w środku tekstu w miejscu jakiegoś wyrazu napisać inny.

Strumienie pracujące na plikach są tak zdefiniowane, by umożliwiać nam identyczny sposób pracy z informacją zapisaną w pliku.

W związku z tym, strumienie mają także analogiczne wskaźniki. Jeśli strumień jest wejściowy, to ma taki wskaźnik do czytania. Ciekawe, że wskaźnik taki pokazuje nie tyle na samą literę, co na miejsce między dwoma literami – ta „z lewej" to ostatnio przeczytana, a tę „z prawej" zaraz będę czytał.

Strumień wyjściowy ma podobny wskaźnik, ale dla pisania. Strumień, który może zapewniać i pisanie i czytanie – ma oba takie wskaźniki - i są one niezależne. Można przecież w jednym miejscu pliku czytać i zapisywać coś w innym.

Wskaźnik czytania – wskaźnik `get`

Jeśli otwieramy plik do czytania, to wskaźnik ten znajduje się na początku pliku. Tak jak nową książkę zaczyna się zwykle czytać od początku. Wskaźnik ten jednak można natychmiast przesunąć w wybrane miejsce.

Wskaźnik do pisania – wskaźnik `put`

Jeśli otwieramy plik do dopisywania (tryb `ios::append`) to wskaźnik ten pokazuje na koniec pliku. Dopisywanie będzie odbywało się do końca pliku.

Wskaźniki do pokazywania wewnątrz plików mają typ

```
streampos
```

typ ten zdefiniowany jest w klasach `istream` i `ostream`. Zwykle jest to po prostu typ `long`. Wskaźnik taki określa numer znaku (bajtu), który czytasz. Na przykład – wczytałeś już 58 literę.

W stosunku do takich wskaźników wykonujemy zwykle dwa typy operacji :

- – ustalenie, gdzie wskaźnik obecnie pokazuje,
- – ustawienie go na żądaną pozycję.

21.17.1 Funkcje składowe informujące o pozycji wskaźników

Aby dowiedzieć się o bieżącą pozycję wskaźników w strumieniu, posługujemy się funkcjami składowymi:

```
streampos tellg() ;          // - f. składowa klasy ifstream
```

```
streampos tellp() ;          // - f. składowa klasy ofstream
```

funkcja `tellg` – co jest zapewne skrótem od „tell get" – powiedz [skąd] brane - czyli powiedz, gdzie pokazuje wskaźnik do czytania,

funkcja `tellp` – analogicznie: tell put – powiedz [gdzie] wstawiane – czyli powiedz, gdzie pokazuje wskaźnik pisania.

Ponieważ klasa `fstream` jest pochodną obu powyższych, dlatego obie funkcje są w niej odziedziczone. Zatem dla strumienia klasy `fstream` - czyli piszącego i czytającego – można używać obu tych funkcji.

Przykładowo, po zapisaniu czegoś w pliku za pomocą strumienia wyjściowego, możemy poznać bieżącą pozycję wskaźnika `put`:

```
ofstream strum("wiersz.tmp", ios::app);
strum << "Zapytajcie Artura, slowo daje nie klamie \n";
cout << "Pozycja wskaznika = " << (strum.tellp() ) ;
```

21.17.2 Wybrane funkcje składowe do pozycjonowania wskaźników

```
istream & seekg(streampos, seek_dir = ios::beg);
ostream & seekp(streampos, seek_dir = ios::beg);
```

Pierwsza z nich pozycjonuje wskaźnik czytania (get), druga wskaźnik pisania (put).

Argumenty tych funkcji:

Pierwszym jest typ `streampos`. Jeśli w Twoim komputerze odpowiada on typowi `long`, to deklaracje tych funkcji składowych możemy napisać prościej:

```
istream & seekg(long, seek_dir = ios::beg);
ostream & seekp(long, seek_dir = ios::beg);
```

Ten pierwszy argument określa na który bajt pliku ma pokazywać wskaźnik. Załóżmy, że np. na 34 bajt pliku.

Drugi argument to typ wyliczeniowy `seek_dir`, (zdefiniowany w klasie `ios`)

```
enum seek_dir {
     beg,                // begin -     początek
     cur,                // current -   bieżący
     end                 // end -       koniec
};
```

służący do określenia punktu odniesienia dla pozycjonowania. Dzięki niemu możemy wyrazić czy chodzi nam o ustawienie wskaźnika na np. o bajt 34 licząc od początku pliku, od końca pliku, czy też może od bieżącej pozycji wskaźnika.

Z deklaracji funkcji `seekp`, `seekg` widać, że przez domniemanie zakładają one pozycjonowanie względem początku pliku.

Oto przykład:

```
#include <iostream.h>
#include <fstream.h>
/*****************************************************/
```

```
main()
{
char tekst[] = {
      "Coz, wedlug Ben-Alego,\n"
      "czarnomistrza Krakowa\n"
      "nie jest to nic wielkiego\n"
      "dorozke zaczarowac."} ;

fstream strum("wiersz.tmp", ios::in | ios::out);

     strum << tekst ;

                    // pozycjonowanie kursora pisania na literze 25
     strum.seekp(25) ;
     strum << "ABCDE" ;

                    // pozycjonowanie kursora pisania na znaku 6 od końca
     strum.seekp(-6, ios::end) ;
     strum << "X" ;
}
```

❐ **W rezultacie plik `wiersz.tmp` zawiera następującą treść**

```
Coz, wedlug Ben-Alego,
cABCDEmistrza Krakowa
nie jest to nic wielkiego
dorozke zaczaXowac.
```

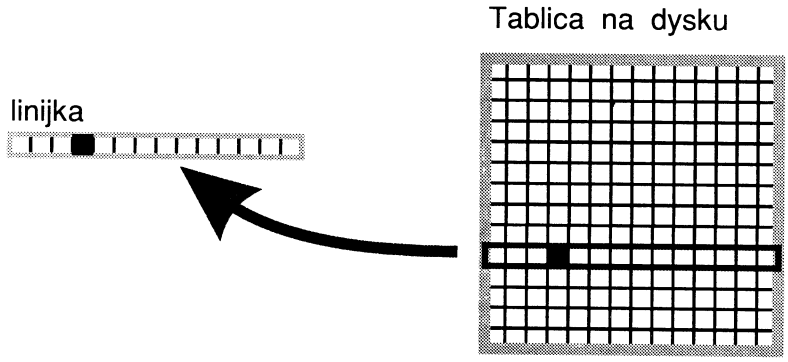
21.18 Przykład większego programu

W mojej praktyce najczęściej pozycjonowaniem wskaźników czytania i pisania posługiwałem się wobec plików zawierających nie tyle dane tekstowe, co dane binarne. Zobaczymy teraz przykładowy program. Rozwiązuje on problem, o którym już kiedyś wspominałem – jak posłużyć się dużą tablicą typu `int` – tak dużą, że aż nie mieści się w pamięci operacyjnej komputera. Rozwiązaniem jest realizacja takiej tablicy na dysku w postaci pliku.

Załóżmy, że taki plik już mamy i że istnieje już strumień, który nas z tą tablicą komunikuje. Dowolna operacja sięgnięcia do wybranego elementu tablicy polega więc na obliczeniu, gdzie w pliku zapisany jest właśnie potrzebny element. Następnie pozycjonuje się tam wskaźnik *czytania*, po czym odczytuje się go. Po ewentualnej modyfikacji elementu zwraca się go na dysk pozycjonując wcześniej wskaźnik *pisania*.

Poniższy przykład pokazuje postępowanie w przypadku, gdy mamy tablicę dwuwymiarową. (Tablica jednowymiarowa jest zbyt prosta – element nr 500 znajduje się przecież zawsze na pozycji `500 * sizeof(int)`).

Do pamięci nie sprowadza się pojedynczych elementów, ale cały wiersz. Mieści się on w jednowymiarowej tablicy `linijka`. Można to pokazać na rysunku



Jeśli chcemy odnieść się do elementu tablicy np. `tab[20][6]`, to do roboczej tablicy `linijka` sprowadzamy z pliku 20 wiersz tablicy, po czym odnosimy się do elementu `linijka[6]`.

Pracę można sobie usprawnić: nie odsyłamy za każdym razem wiersza z powrotem na dysk, bo możliwe, że zaraz potrzebować będziemy jakiegoś innego elementu z właśnie *tego samego wiersza*. Dopiero, gdy mamy następne odniesienie się do elementu tablicy `tab` – oceniamy sytuację: jeśli rzeczywiście jest to element z tego samego wiersza – to jesteśmy szczęśliwi – opłaciło się! ; jeśli nie – to najpierw odsyłamy na dysk wiersz, na którym pracowaliśmy poprzednio, po czym sprowadzamy żądany.

Aby praca z taką tablicą była tak naturalna, jak praca ze zwykłym typem wbudowanym – wszystkie operacje pisania i czytania pliku dyskowego realizujemy za pomocą przeładowania operatora `[]`. Operator ten odpowiada tylko za operacje na wierszach tablicy. Odniesienie się do poszczególnych elementów we wierszu załatwia już zwykły operator [ ]

Zapis

```
tab[20][6]
```

odpowiada zapisowi

```
(tab[20]) [6]
```
← *zwykły operator*

↑ *ten operator jest przeładowany*

To wyrażenie w nawiasie wywoła sprowadzenie do tablicy linijka wiersza nr 20. Definiując przeładowanie operatora – rezultatem tego wyrażenia uczynimy adres tablicy linijka, dzięki temu wyrażenie to będzie równoważne takiemu

```
(linijka) [6]
```

Tutaj już operator [ ] zastosowany jest w stosunku do zwykłej (jednowymiarowej) tablicy `int` – dlatego nie zadziała tu „nasz" przeładowany, ale ten zwykły pracujący zawsze dla tablic `int`

Oto program:

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>                    // dla exit()
/////////////////////////////////////////////////////////////
```

```
class duza_tablica {                                        // ❼
      char *nazwa_pliku ;                    // nazwa pliku dyskowego
      fstream plik ;

      const int rozmiar ;

      int * linijka ;                        // linijka w zapasie pamięci
      long dlug_linijki ;
      int nr_obecnego ;                      // nr wiersza nad którym pracujemy
public:
      duza_tablica(char *nazwa_pliku, int wym, int wypelniacz = 0);
      ~duza_tablica() ;

      int * operator[](int wiersz) ;
private:
      void error();
} ;
/*******************************************************/
duza_tablica::duza_tablica(char *nazwa, int wym,
          int wypelniacz) : rozmiar(wym)                    // ❽
{
     plik.open(nazwa, ios::in | ios::out | ios::binary) ;
                                                        // ↑---❾
     nazwa_pliku = nazwa ;                                  // ❿

     if(!plik){
          cout << "Blad otwarcia pliku " << nazwa ;
          error() ;
     }

     linijka = new int[rozmiar] ;                          // ❶❶
     dlug_linijki = rozmiar * sizeof(int);

     // ------------------------ wstępne wypełnienie
     for (int i = 0 ; i < rozmiar ; i++)
                    linijka[i] = wypelniacz ;

     // ------------------------zapis tego do pliku
     for (i = 0 ; (i < rozmiar) && plik ; i++)
          plik.write( (char*)linijka, dlug_linijki ) ;
                                                        // ❶❷

     if(!plik){
          cout << "Blad podczas wypelniania tablicy"
               << endl ;
          error();
     }
     nr_obecnego = 0 ;
}
/*******************************************************/
duza_tablica::~duza_tablica()                           // ❶❸
{
     // skoro koniec to musimy odesłać na dysk wiersz
     // który jest jeszcze w tablicy linijka

     if(! plik.seekp(nr_obecnego * dlug_linijki)) {
```

```
            cout << "Blad pozycjonowania "
                     "wskaznika pisania\n";
            error();
        }
        // zapisanie tam ---------
        if(!plik.write( (char*)linijka , dlug_linijki) ){
            cout << "Blad przy odsylaniu\n" ;
            error();
        }

        plik.close(); // zamknięcie pliku
        delete linijka ; // zwolnienie rezerwacji
}
/*****************************************************/
// przeladowanie operatora []
/*****************************************************/
int * duza_tablica::operator[](int nr_wiersza_chcianego)
{                                                    // ❶❹
        //============================
        // może dany fragment tablicy jest pod ręką ?
        //============================
        if(nr_wiersza_chcianego == nr_obecnego){
             return linijka ;  // nic nie trzeba robić !
        }

        //============================
        // jeśli nie, to najpierw odsyłamy obecny wiersz=
        // ==========================
        // obliczenie gdzie go posłać
        if(! plik.seekp(nr_obecnego * dlug_linijki)) {
            cout << "Blad pozycjonowania wskaznika pisania\n";
            error();
        }
        // zapisanie tam ----------------------
        if(!plik.write( (char*)linijka , dlug_linijki) ){
            cout << "Blad przy odsylaniu\n" ;
            error();
        }
        // ==================================
        // sprowadzamy potrzebny wiersz ===
        // ==================================
        // obliczenie gdzie on w pliku jest

        if(! plik.seekg(nr_wiersza_chcianego * dlug_linijki))
        {
            cout << "Blad pozycjonowania wskaznika czytania"
                 << nr_wiersza_chcianego * dlug_linijki
                 << endl ;
            error();
        }

        // przeczytanie tego fragmentu ----------------------
        if(!plik.read( (char*) linijka, dlug_linijki) ){
            cout << "Blad czytania wiersza "
                 << nr_wiersza_chcianego << endl ;
            error();
```

```
        }

        // uaktualnienie notatek
        nr_obecnego = nr_wiersza_chcianego ;

        // zwracamy wskaźnik do tablicy roboczej
        return linijka ;
}
/*****************************************************/
void duza_tablica::error()
{
 if(plik.eof()) cout << "Koniec pliku " ;
 if(plik.fail())
          cout << " -FAIL strumienia do pliku : "
           << nazwa_pliku << endl ;
 exit(1);
}
/*****************************************************/
main()                                                  // ❶
{
int i, k;
const int wymiar = 50 ;
duza_tablica t("macierz.tmp", wymiar) ;                 // ❷

        // wpisanie jakichś danych do tablicy t
        for(i = 0 ; i < wymiar ; i++)
             for(k = 0 ; k < wymiar ; k++)
                       t[i][k] = i * 100 + k ;          // ❸

        cout << "Element t [24][7] ma wartosc = "
             << t [24][7] << endl ;                     // ❹

        //-----------------------------------------
        cout << "Przykladowo robimy transpozycje " <<endl ;

        duza_tablica m("macierz1.tmp", wymiar) ;        // ❺
        for(i = 0 ; i < wymiar ; i++)
             for(k = 0 ; k < wymiar ; k++)
                  m[i][k] = t[k][i] ;                   // ❻

        cout << "transpozycja skonczona, oto rezultat\n"
                  "Przykladowo elementy t[7][31] = " << t[7][31]
             << ", \tt[31][7] = " << t[31][7] << endl ;

        cout << " Natomiast elementy m[7][31] = " << m[7][31]
             << ", \tm[31][7] = " << m[31][7] << endl ;
}
```

❒ **Po wykonaniu programu na ekranie pojawi się następujący tekst**

```
Element t [24][7] ma wartosc = 2407
Przykladowo robimy transpozycje
transpozycja skonczona, oto rezultat
Przykladowo elementy t[7][31] = 731,  t[31][7] = 3107
 Natomiast elementy m[7][31] = 3107,  m[31][7] = 731
```

Komentarz

❶ Jeśli przyjrzymy się funkcji `main` to, poza dwoma linijkami ❷ oraz ❺, nie ma w niej nic osobliwego. To jest najważniejsze przesłanie tego przykładu: mimo, że dokonujemy niesamowitych operacji z tablicą na dysku – ten, kto posługuje się nią, nic o tym nie musi wiedzieć.

❷ Definicja obiektu klasy `duza_tablica`. Odpowiada ona mniej więcej takiemu zapisowi

```
int t[50][50] ;
```

❸ Następnie widzimy pętlę, w której wpisuje się do takiej tablicy jakieś wartości. Instrukcja przypisania jest tak naturalna, jak do tego przywykliśmy przy typach wbudowanych.

❹ Podobnie wypisanie na ekranie treści jakiegoś elementu wygląda normalnie.

❺ Załóżmy, że konieczna jest transpozycja tablicy, czyli zamiana jej wierszy z kolumnami. Aby tego dokonać definiujemy sobie inną tablicę, w której znajdzie się wynik tej operacji. Definicja obiektu, który tu widzimy, odpowiada jakby definicji

```
int m[50][50]
```

❻ Sama transpozycja to po prostu operacja

```
m[i][k] = t[k][i]
```

przeprowadzona dla każdego elementu tablicy. Znowu ten sam wniosek: patrząc na ten zapis nie można się nawet domyślić, że tablice są przechowywane na dysku.

Gdzie tkwi tajemnica tej prostoty ?

❼ Tablice na dysku są zrealizowane jako obiekty klasy – którą w tym celu wymyśliliśmy. Oto składniki-dane tej klasy:

- – wskaźnik do stringu będącego nazwą pliku dyskowego. Pod taką nazwą tablica jest zapisana na dysku.
- – obiekt klasy `fstream`. Tak! – Ośrodek dowodzenia strumieniem jest składnikiem klasy `duza_tablica`. To zrozumiałe - przecież każdy następny egzemplarz dużej_tablicy musi mieć swój strumień, swój ośrodek dowodzenia.
- – stała określająca rozmiar tablicy ; jeśli definiujemy tablicę o rozmiarach [50][50] to tutaj zapisana jest liczba 50.
- – składnikiem w klasie powinna być też jednowymiarowa tablica `linijka` – do której będziemy sprowadzali poszczególne wiersze tablicy. Ponieważ jednak nie wiemy jakie rozmiary ma tablica, którą użytkownik klasy sobie zażyczy (może 50x50, a może 8192x8192 ?), dlatego też nie wiemy jak długa ma być linijka. Skoro jeszcze teraz nie wiemy, to załatwimy to za pomocą dynamicznej rezerwacji tablicy: tablica `linijka` zostanie założona z zapasie pamięci operatorem `new` już w trakcie

wykonywania programu. Składnikiem `duza_tablica` klasy zostaje tylko wskaźnik do tak założonej tablicy. To on nazywać się będzie `linijka`.

- – `dlug_linijki` – aby uniknąć ciągłego obliczania jak długa (w bajtach) jest `linijka` zapisujemy sobie jako składnik klasy. Jest to po prostu wartość

```
rozmiar * sizeof(int)
```

- – składnikiem klasy jest też zmienna `int`, w której zapamiętujemy numer wiersza, który jest sprowadzony z tablicy dyskowej i jest obecnie dostępny w `linijce`.

Klasa ma także funkcje składowe: konstruktor, destruktor, operator`[]` oraz funkcję `error` wypisującą informacje o błędach strumienia `plik`.

❽ Konstruktor. Ma trzy argumenty. Pierwszy to nazwa, pod jaką na dysku ma być zapisana duża tablica. Drugi argument to rozmiar tej tablicy – w tym przykładzie zakładamy, że tablica jest „kwadratowa", czyli ma tyle samo wierszy co kolumn. (W moich zastosowaniach tak właśnie najczęściej jest). Trzeci argument to wartość, którą należy wpisać do wszystkich elementów tablicy w momencie, gdy ją zakładamy.
Ponieważ `rozmiar` jest stałą (`const`), dlatego nadania mu wartości musimy dokonać w liście inicjalizacyjnej konstruktora.

W ciele konstruktora widzimy kolejno:

❾ Otwarcie pliku dyskowego. Plik otwierany jest zarówno do pisania i czytania. Dodatkowo widzimy „OR-owany" tryb `ios::binary` – tablice mają przechowywać dane zapisane w sposób binarny – (przykład kompilowałem kompilatorem Borland C++, w którym występuje ten tryb).

❿ Zapamiętanie nazwy pliku. Ponieważ na nazwę pokazuje wskaźnik, to jest to tylko przypisanie wskaźników, a nie funkcja biblioteczna `strcpy`.

❶❶ Rezerwacja w zapasie pamięci jednowymiarowej tablicy `linijka` o żądanej długości.

❶❷ Wypełnioną żądaną wartością `linijke` zapisujemy do tablicy dyskowej wielokrotnie, jako wszystkie kolejne wiersze. Oczywiście takie postępowanie nie musi nam odpowiadać. Tablica na dysku może być przygotowana przez inny program i być zbiorem danych pochodzących z trwającego tydzień eksperymentu — czyli już w momencie otwarcia pliku może zawierać wartościowe informacje. Wówczas oczywiście ten fragment ciała konstruktora jest niepotrzebny – należy go zmienić lub po prostu napisać drugi konstruktor na okoliczność otwarcia takiej „wartościowej" tablicy.

❶❸ Destruktor powinien dokonać ostatnich operacji przed zakończeniem pracy z tablicą. Do tych prac należy odesłanie na dysk wiersza, z którym ostatnio pracowaliśmy, zamknięcie pliku oraz zwolnienie rezerwacji pamięci na jednowymiarową tablicę `linijka`.

❶❹ Przeładowany `operator[]`. Jak wiemy funkcja `operator[]` **obowiązkowo** musi być funkcją składową klasy. Tak jest i u nas. Argumentem jest tu numer wiersza, o który chodzi. Rezultat to typ `int*`. Rezultatem będzie tu zawsze adres tablicy `linijka`. Dlaczego właśnie tak jest najlepiej – o tym szczegółowo mówiliśmy w rozdz. o przeładowaniu – patrz str. 464.
Ponieważ ciało tego operatora szczegółowo opisałem komentarzami, dlatego nie trzeba chyba dalszych wyjaśnień. Widzimy tam także operacje pozycjonowania wskaźników czytania i pisania. Widać tutaj jak bardzo są przydatne.

21.19 Tie – harmonijna praca dwóch strumieni

Mówiliśmy już, że strumień `cout` jest buforowany. Oznacza to, że wypadku takiego fragmentu programu:

```
cout << "Raz" ;
cout << "Dwa" ;
cout << "Trzy" ;
```

przesłanie znaków na ekran nie następuje po każdej poszczególnej linijce. Dla usprawnienia sobie pracy strumień `cout` czeka, aż uzbiera się więcej znaków do wypisania i wtedy wypisuje wszystko hurtem. Jednokrotne przesłanie dłuższej partii znaków zajmuje mniej czasu niż kilkakrotne przesłanie krótszych. W rezultacie program pracuje szybciej.

Rozważmy jednak taką sytuację

```
cout << "Podaj energie zderzenia : " ;
cin >> e ; ❶
cout << "Podaj parametr zderzenia : "
cin >> m ;
```

Gdyby i tutaj strumień `cout` czekał z wszystkimi pytaniami aż uzbiera się ich więcej wówczas w ❶ strumień `cin` czekałby na odpowiedź na pytanie, które jeszcze nie pojawiło się na ekranie. To bez sensu. Nie próbuję Cię tu jednak przekonać, że idea buforowania strumienia `cout` jest bezsensowna. Chodzi jednak o to, by robić to mądrze.

Niech strumień `cout` będzie buforowany i rzeczywiście optymalizuje sobie pracę, czekając aż uzbiera się więcej tekstów. Ale – gdy tylko strumień wejściowy `cin` zechce wczytywać jakieś znaki (np. odpowiedź) – niech najpierw zostanie na ekranie wypisane wszystko, co czeka jeszcze w buforze strumienia `cout`. Dopiero wtedy strumień `cin` zajmie się wczytywaniem.

Taki sposób jest bardzo korzystny, gdyż nie rezygnując z dobrodziejstwa buforowania mamy poprawne współdziałanie strumieni wejściowego i wyjściowego. Partie programu, które tylko piszą na ekranie, nie są dzięki tej metodzie spowalniane, natomiast te, które rozmawiają z użytkownikiem, nadal pracują poprawnie.

Zjawisko takie nazywamy powiązaniem strumieni[+)]. Powiązane dotyczyć może nie tylko strumieni `cout` i `cin`, ale także innych. Wiązać można dowolny strumień (wejściowy lub wyjściowy) z ja-

kimś strumieniem wyjściowym - to on będzie miał wypróżniany bufor.

Dlaczego jednak do tej pory nigdy nie zauważyliśmy wadliwego działania strumieni w takich sytuacjach? To dlatego, że strumienie predefiniowane są już dla nas powiązane automatycznie.

Strumienie `cin`, `cerr`, `clog`, są powiązane ze strumieniem `cout` – oznacza to, że jeśli chcemy dokonać jakiejkolwiek akcji strumieniami `cin`, `clog`, `cerr`, wówczas najpierw wypróżniany jest bufor strumienia `cout`.

Powiązanie ze sobą innych strumieni może odbyć się na życzenie programisty. Wystarczy w tym celu wywołać odpowiednią funkcję składową klasy `ios`

```
ostream * tie(ostream *) ;
ostream * tie() ;
```

Pierwsza z tych funkcji powoduje związanie strumienia z jakimś strumieniem wyjściowym. Argumentem funkcji jest wskaźnik do strumienia wyjściowego, z którym należy wiązać. Funkcja jako rezultat zwraca wskaźnik do strumienia, z którym było powiązanie do tej pory.

Druga z tych funkcji tylko zwraca rezultat, będący wskaźnikiem do strumienia, z którym obecnie dany strumień jest powiązany.

Przykładowo mamy strumienie

```
istream ii ;
ostream oo, mm ;
```

Oto jak powiązać strumienie `ii` oraz `oo`, do strumienia wyjściowego `mm`:

```
ii.tie(mm);
oo.tie(mm);
```

Od tej pory dwa strumienie `ii` oraz `oo` powiązane są ze strumieniem `mm`. Znaczy to, że jakakolwiek operacja we/wy na strumieniach `ii` oraz `oo`, spowoduje najpierw wypróżnienie bufora strumienia `mm`.

Jeśli istniejące powiązanie strumieni nam nie odpowiada, możemy je przerwać. Odbywa się to również za pomocą funkcji `tie` z tym, że teraz argumentem jest `0`. Przerwać możemy powiązanie, które zrobiliśmy my sami lub powiązanie już wcześniej istniejące.

Np. instrukcja

```
cin.tie(0) ;
```

przerwie powiązanie strumienia `cin` ze strumieniem `cout`.

†††† tie - ang: związać (czytaj: „taj")

Kiedy takie powiązanie strumieni może się przydać ?

Przede wszystkim wtedy, gdy dwa różne strumienie pracują na tym samym pliku. Przykładowo: jeśli jeden strumień wpisuje do pliku dane o rezerwacji miejsc w samolocie, natomiast drugi odczytuje tę informację, to strumień wpisujący może być buforowany dla usprawnienia jego pracy. Jeśli jednak strumień czytający ten plik spróbuje dowiedzieć się o stan rezerwacji miejsc – to wszystkie dane o ostatnio zajętych miejscach czekające jeszcze w buforze powinny być najpierw uaktualnione na dysku, a dopiero wtedy strumień wejściowy odczyta czy jeszcze jakieś miejsca są wolne. Taką pracę zapewni nam właśnie związanie ze sobą tych strumieni. Strumień czytający powinien po prostu wywołać funkcję składową `tie` mówiąc, że chce związać ze sobą strumień piszący.

21.20 Attach – zmiana koryta strumienia

Może zdarzyć się tak, że pracując z jakimś strumieniem nagle podejmiemy decyzję, iż zamykamy jego plik i od tej pory jego ujściem niech będzie np. ekran, czyli dalej będzie pisał on na ekran.

Do realizacji tego służy funkcja składowa `attach`†).

Funkcja ta jest składnikiem jednej z klas podstawowych klas `ifstream`, `ofstream`, zatem pracując z plikami mamy ją do dyspozycji.

```
void attach(int) ;
```

Argumentem tej funkcji jest liczba `int` oznaczająca tak zwany deskryptor pliku. Jest to numer oznaczający plik lub urządzenie wyjściowe (klawiatura, ekran). Oto deskryptory plików używane przez strumienie predefiniowane:

0 – standardowe wejście – (klawiatura) używa go strumień `cin`

1 – standardowe wyjście – (ekran) używa go strumień `cout`

2 – standardowe wyjście dla komunikatów o błędach – używa go strumień `cerr` oraz strumień `clog`

Przykład:

```
char c ;
ofstream s("t.tmp");
      s << "Probny wypis A " ;

      cout << "zamknac plik i pisac dalej "
              "na ekran ? t/n " ;
      cin.get(c) ;
      if(c == 't'){
           s.close();
           s.attach(1);
      }
      s << "Probny wypis B " ;
      s << "Probny wypis C " ;
```

†) (attach – czytaj: "etacz")

```
        s << "Probny wypis D " ;

    s.close() ;
```

Dzięki funkcji `attach` łatwo mogliśmy sprawić, że znaki, zamiast popłynąć do pliku "t.tmp", pojawią się na ekranie.

Na końcu widzimy wywołanie funkcji `close` – gdyż jeśli dane płyną rzeczywiście do pliku, to trzeba go w końcu zamknąć. Jeśli jednak płynęły do innego strumienia (`cout`), to funkcja ta będzie zignorowana. Zamknąć bowiem można ujście (lub źródło) tylko wtedy, jeśli jest się właścicielem strumienia.

21.21 Dlaczego tak nie lubimy biblioteki `stdio`?

Paragraf ten jest przeznaczony dla tych, którzy zetknęli się z biblioteką `stdio` (programiści klasycznego C). Pozostałym czytelnikom radzę ten (oraz następny) paragraf opuścić i przejść do czytania zagadnień o formatowaniu wewnętrznym.

Jeśli programując w klasycznym C przywykłeś do korzystania z biblioteki `stdio` zapewne zadasz mi pytanie:

„–Dlaczego właściwie tak nie lubisz biblioteki `stdio`?"

Odpowiadam: Ależ lubię, lubię! Tak samo, jak lubi się starego przyjaciela. Dokładnie tak samo: przeżyliśmy ze sobą dość dużo czasu, znamy swoje liczne wady, ale lubimy się mimo wszystko – dlatego, że się dobrze rozumiemy.

Trudno jednak nie zauważać wad

Oto one:

1) Z uwagi na to, że podstawowe funkcje wypisywania `printf(...)` i wczytywania `scanf(...)` są zdefiniowane jako mogące zawierać zmienną liczbę argumentów – kompilator nie potrafi nas ostrzec w wypadku, gdy zapomnimy o jakimś argumencie, albo damy o kilka argumentów za dużo w stosunku tego, co napisaliśmy w specyfikacji formatu.

```
printf("Wypis trzech liczb : %d %d %d \n", liczba1);
```

Kompilator nie skontroluje też typu tych argumentów. Jeśli liczbę całkowitą spróbujemy wypisać za pomocą formatu dla liczb zmiennoprzecinkowych, to kompilator nie może takiego błędu odszukać i nas ostrzec.

```
int a = 4 ;
printf("%6.2f", a);
```

Wydruk, który pojawi się na ekranie w rezultacie takiej błędnej instrukcji, wcale nie musi od razu wyglądać bezsensownie – to właśnie utrudnia wykrycie błędu jeszcze bardziej. Biblioteka powinna być tak skonstruowana, by dać kompilatorowi szansę wykrycia takich – elementarnych w końcu - błędów, jak niezgodność typu i liczby argumentów wywoływanej funkcji.

2) Przypomnij sobie ile trudu kosztowało Cię przyswojenie sobie wiedzy kiedy to w funkcji `scanf` należy użyć znaku `&`
Na przykład:

```
char tablica[10], c ;
int i, j ;

scanf("%d %s %c %s", &i, tablica,
                         &tablica[3],
                         &tablica[4]);
```

Czy w tej linii jest błąd czy nie?

Trzeba trochę pomyśleć, tyle że to szlachetne myślenie nie jest zbyt twórcze. Programista powinien myśleć bardziej nad sensem tego, co pisze, a nie nad gramatyką. O ileż prostszy jest zapis

```
cin  >> i >> tablica
     >> tablica[3] >> (char*)tablica[4];
```

gdyż nie musimy pamiętać o używaniu znaków `&`.

3) Jeśli piszesz krótki program, który wypisuje na ekranie tylko jedną liczbę całkowitą instrukcją

```
printf("%d", i );
```

to mimo wszystko z biblioteki `stdio` włączane są fragmenty odpowiedzialne za: wypisanie stringu, wypisanie pojedynczego znaku, wypisanie liczby zmiennoprzecinkowej z precyzją 10 miejsc znaczących itd, itd. Tymczasem używając instrukcji

```
cout << i ;
```

powodujemy, że w programie znajdą się tylko te fragmenty, które służą do wykonania wypisu liczby całkowitej w notacji dziesiątkowej. Oszczędzamy miejsce w pamięci, co może być czasem bardzo ważne. Program wynikowy jest wtedy mniejszy.

4) Jedną z najważniejszych wad biblioteki `stdio` jest fakt, iż posługując się nią, **nie mamy eleganckiego sposobu na wczytywanie i wypisywanie obiektów typów zdefiniowanych przez użytkownika.** To, co w bibliotece `iostream` załatwia się za pomocą przeładowania operatorów `>>` oraz `<<` – tutaj, w bibliotece `stdio` staje się bardzo skomplikowane. W programie nie możemy po prostu wydać polecenia - wczytaj z pliku dyskowego dwie liczby `int`, dwa obiekty klasy `osoba`, liczbę `float` i obiekt klasy `wektorek`.

Posługując się klasą zdefiniowaną przez kolegę musimy znać wszystkie jej składniki, mieć do nich dostęp, by żmudnie z dysku wczytywać do tych składników liczby czy stringi.

Gwałci to zasadę enkapsulacji – użytkownik musi znać wszystkie trybiki i kółka klasy po to, by jej obiekty swobodnie wczytywać z dysku.

Jedną ze sztandarowych zasad języka C++, jest jego rozszerzalność o nowe typy – zdefiniowane przez użytkownika. Biblioteka stdio nie zapewnia tego. Dlatego radzę się z nią pożegnać.

✦ 5) Użycie printf z biblioteki stdio jest po prostu wolniejsze niż cout. To dlatego, żeprintf **dopiero w czasie wykonywania programu** analizuje string formatu i zależnie od tego, co tam znajdzie, uruchamia odpowiedni podprogram wypisujący dany typ danej.

Tymczasem cout przygląda się typowi argumentu **już w czasie kompilacji** i wtedy decyduje, co będzie mu potrzebne. Tym sposobem, cout wykonując później daną linijkę programu miliony razy, nie musi się nad niczym zastanawiać. (Głupszy printf będzie miliony razy powtarzał oglądanie tego samego formatu).

✦ 6) Jeśli w bibliotece stdio coś nam się nie podoba, to nie możemy sobie tego odziedziczyć, zasłonić lepszą wersją i być szczęśliwym.

Podobnie: jeśli w bibliotece stdio coś nam się bardzo podoba, to nie możemy sobie tego odziedziczyć by w swojej klasie jeszcze bardziej udoskonalić. Ta biblioteka nie nadaje się do dziedziczenia. Jest bezpłodna.

Tymczasem jeśli chodzi o iostream, to za chwilę zobaczysz jak, dla potrzeb tzw. formatowania wewnętrznego, odziedziczono fragmenty biblioteki iostream i jakie powstało świetne narzędzie. Sam też możesz coś tak wykorzystać.

21.22 Niektóre aspekty współżycia biblioteki stdio z biblioteką iostream

Mówiliśmy, że przy korzystaniu z biblioteki iostream mamy następujące predefiniowane strumienie

```
cin, cout, cerr, clog
```

Jak Ci zapewnie wiadomo, w bibliotece stdio są odpowiadające im

```
stdin, stdout, stderr
```

Pamiętać trzeba, że nie ma żadnej łączności między nimi, poza faktem, iż używają one tych samych deskryptorów plików. Co to oznacza w praktyce?

Otóż jeśli mieszamy posługiwanie się jedną i drugą biblioteką, to możemy mieć kłopoty z synchronizacją. Przykładowo – poniższy fragment kodu

```
cout << "Jeden " ;
printf("Dwa ");
cout << "Trzy " ;
printf("Cztery ") ;
cout << "Piec " << endl ;
```

wcale nie musi spowodować pojawienie się na ekranie tekstu

```
Jeden Dwa Trzy Cztery Piec
```

Możemy otrzymać

```
Jeden Trzy Piec
Dwa Cztery
```

albo na przykład

```
Dwa Cztery Jeden Trzy Piec
```

Dlaczego?

> Otóż cout jest buforowany, a stdout nie. Inna jest zasada wypisywania, dlatego nie pracują one ramię w ramię.

Aby biblioteka iostream uwzględniała operacje wykonywane z biblioteką stdio, powinniśmy ją uprzedzić o tym, że zamierzamy korzystać z stdio. Konkretnie – strumień z nowej biblioteki uprzedzamy o tym za pomocą funkcji składowej

```
ios::sync_with_stdio()
```

(co znaczy: synchronizuj z stdio). Jeśli instrukcję

```
ios::sync_with_stdio() ;
```

Umieścimy na początku naszego ostatniego przykładu, to poszczególne wyrazy zawsze pojawiać się będą na ekranie we właściwej kolejności.

```
Jeden Dwa Trzy Cztery Piec
```

Co to za tajemnicza funkcja? Jest to funkcja składowa klasy ios

```
static void sync_with_stdio() ;
```

Jest to funkcja statyczna tej klasy, a więc nie jest wywoływana na rzecz jakiegoś konkretnego strumienia (np cin, cout, clog, ...), ale na rzecz klasy tych obiektów. Czyli: wszystkich strumieni pochodzących od klasy ios. Wywołanie tej funkcji w jakimś miejscu programu sprawia, że ustawia się flaga stanu formatowania ios::stdio. Od tej pory biblioteka iostream zaczyna postępować ostrożniej – strumienie cin, cout będą pracowały używając innego typu bufora (stdiobufs), co umożliwi im zgodną pracę z stdio, stdout, stderr.

Co na tym zyskaliśmy?

Poprawną synchronizację i możliwość mieszania użycia starej i nowej biblioteki.

Co na tym tracimy?

Tracimy na szybkości strumieni predefiniowanych, które przestają być buforowane.

Rezygnacja z buforowania jest obniżeniem sprawności pracy tych strumieni, ale robią one to po to, by staruszkowie ze starej biblioteki stdio mogli za nimi nadążyć. Ponieważ jest to czasem cena dość wysoka, dlatego synchronizacja taka nie jest robiona przez domniemanie[†)]. Programista sam decyduje czy tego chce. Do tego właśnie służy przedstawiona funkcja.

21.23 Formatowanie wewnętrzne – Operacje wyjścia

Kiedy mówiliśmy co to są operacje wejścia i wyjścia – powiedziałem, że chodzi o zagadnienie komunikowania się programu z urządzeniami zewnętrznymi — klawiaturą, ekranem, plikami na dyskach magnetycznych itd. Tymczasem teraz zajmiemy się operacjami we/wy nie komunikującymi nas wcale z plikami na urządzeniach zewnętrznych. Strumienie, o których będziemy mówić teraz — zamiast do plików, płynąć będą do jakichś obszarów pamięci. Tak, jakby to tam był właśnie plik. Zwykle takim miejscem pamięci jest tablica znakowa.

Dla programistów klasycznego C nadmienię, że chodzi tu o to samo co w bibliotece `stdio` *wykonywane było funkcjami* `sprintf` *i* `sscanf`*, natomiast dla znających FORTRAN wystarczy powiedzieć, że chodzi o ENCODE i DECODE*

Zagadnienie jest takie. Mamy instrukcję

```
cout      << "Awaria silnika "<< setw(2) << nr_silnika
          << ", temperatura oleju " << setprecision(3)
          << temperat << " stopni C " << endl ;
```

wypisującą na ekranie komunikat. Tymczasem z jakichś powodów chcielibyśmy mieć cały ten tekst nie na ekranie, ale wpisany do tablicy

```
char komunikat[80] ;
```

Innymi słowy chodzi nam o strumień, tyle że płynący nie do pliku, ale do tablicy. Z uwagi na podobieństwo akcję tę nazywamy nadal operacją we/wy. Istnieje w bibliotece klasa strumieni, które oddają nam właśnie te usługi. Aby posłużyć się takimi strumieniami niezbędne jest włączenie do programu pliku nagłówkowego `strstream.h` zawierającego odpowiednie deklaracje. Robimy to dyrektywą

```
#include <strstream.h>
```

Aby wykonać zadanie wypisania naszego stringu do tablicy `komunikat` posłużymy się strumieniem klasy `ostrstream` (Output STRing STREAM - wyjściowy strumień do tablicy znakowej). Ta klasa jest pochodną klasy `ostream`, a zatem dziedziczy wszystkie jej zachowania – możemy więc sobie używać manipulatorów, czy znanych nam funkcji składowych `write`, `put` itd.

Wpisanie czegoś do tablicy znakowej to ostatecznie nic trudnego – robiliśmy to wielokrotnie funkcjami `strcpy`. Tutaj jednak chodzi o to, że możemy skorzystać z całego dobrodziejstwa formatowania oferowanego nam przez strumienie we/wy. Możemy zadać precyzję, szerokość, znaki wypełniające, typ notacji itd. Tego wszystkiego nie mieliśmy przy zwykłej bibliotecznej funkcji `strcpy`. Nasz strumień zapewnia nam więc formatowanie – a ponieważ to „formatowanie" odbywa się nie na użytek pliku zewnętrznego tylko na użytek tablicy w pamięci - dlatego zagadnienie to nazywa się też – **formatowaniem wewnętrznym**.

+) Dawniej była.

A oto jak w praktyce zrealizujemy nasze zadanie:

```
#include <iostream.h>
#include <iomanip.h>
#include <strstream.h>
/*****************************************************/
main()
{
int nr_silnika = 4 ;
float temperat = 156.7123 ;

char komunikat[80] ;
ostrstream strumyk(komunikat, sizeof(komunikat) );   // ❶

// a teraz właściwe wypisanie

 strumyk << "Awaria silnika "<< setw(2) << nr_silnika
         << ", temperatura oleju " << setprecision(2)
         << temperat << " stopni C \n" ;              // ❷

 strumyk << "Musisz cos zrobic !!!\n" << ends ;       // ❸
 cout << "Aby sie przekonac czy w tablicy "
          "znalazl sie\n"rzeczywiscie zadany tekst "
          "wypisujemy jej tresc\nna ekran \n" ;
 cout << "*****************************************\n"
      << komunikat
      << "*****************************************\n";

 strumyk.seekp(8, ios::beg);                          // ❹
 strumyk << "XYZ" ;
 cout << "Po zabawie z pozycjonowaniem :\n"
      << komunikat ;
}
```

Po wykonaniu programu na ekranie pojawi się

```
Aby sie przekonac czy w tablicy znalazl sie
rzeczywiscie zadany tekst wypisujemy jej tresc
na ekran
*****************************************
Awaria silnika 4, temperatura oleju 156.71 stopni C
Musisz cos zrobic !!!
*****************************************
Po zabawie z pozycjonowaniem :
Awaria sXYZika 4, temperatura oleju 156.71 stopni C
Musisz cos zrobic !!!
```

Komentarz

❶ Definicja obiektu `strumyk` będącego egzemplarzem obiektu klasy `ostrstream`. W nawiasie widzimy argumenty wywołania konstruktora. Pierwszym jest adres tablicy, z którą `strumyk` ma pracować (ujście strumienia). Drugim argumentem jest rozmiar tej tablicy. (Mógłby być jeszcze trzeci argument, ale na razie go pomijamy i korzystamy z domniemania.)

❷ Tu nawet nie ma co objaśniać. Operacja wygląda tak samo, jakbyśmy wypisywali na ekran albo na plik.

❸ Ponowne wstawianie do strumienia. Efekt będzie taki sam, jakbyśmy dopisali coś do pliku. Wypisywanie na ekranie (czy pliku) nie powoduje wysłania tam kończącego string znaku NULL. Na ekranie jest on przecież niepotrzebny. My jednak chcielibyśmy, by w naszej tablicy komunikat – na zakończenie wpisanej tam treści - znalazł się znak NULL. Dlatego stosujemy manipulator `ends` (end string), który do istniejącej treści tablicy `komunikat` dopisze znak końca stringu (NULL).

Jeśli w trakcie kolejnych operacji wpisywania do naszej tablicy przekroczylibyśmy liczbę 79 znaków, wówczas wpisywanie tam zostanie przerwane na 79 znaku, a jako 80 znak wpisany zostanie znak NULL.
Równocześnie w ośrodku dowodzenia strumieniem ustawi się bit błędu `ios::badbit`

❹ Praca strumienia płynącego do tablicy znakowej przypomina pracę na pliku – nawet w tym, że także i tu można stosować pozycjonowanie wskaźnika pisania. Ustawiając go na ósmym znaku sprawiamy, że następny wpis do tej tablicy rozpocznie się właśnie od tego miejsca.

W naszym przykładzie posłużyliśmy się konstruktorem strumienia klasy `ostrstream`. Przyjrzyjmy mu się bliżej

```
ostrstream::ostrstream(char *tab, int rozm, int tryb = ios::out);
```

Gdzie poszczególne argumenty to:

`char *` – to adres miejsca w pamięci, które jest ujściem strumienia,

`int` – jest informacją ile bajtów w pamięci przysługuje tejże tablicy,

`int` – jest określeniem trybu pracy tego strumienia (podobnym jak w wypadku pracy z plikiem dyskowym).

Domniemanym trybem jest `ios::out`, czyli wstępnie wskaźnik pisania w tej tablicy ustawiony jest na jej początku.

Jeśli zastosujemy tryb `ios::app` wówczas strumień uzna, że w tablicy już jakiś string jest i chodzi o dopisywanie do tej zastanej treści. Odszukuje znak NULL kończący ten stary string i od tego miejsca zacznie się dopisywanie do tej tablicy.

Oto przykład z zastosowaniem trybu `ios::app`

```
char tekscik[40] = { "UWAGA: " } ;

ostrstream skryba(tekscik, sizeof(tekscik), ios::app);

     skryba << "Zapiac pasy" << ends ;

     // dla kontroli co tam jest ----------
     cout << "*** " << tekscik << " ***\n" ;
```

Na ekranie zobaczymy

```
*** UWAGA: Zapiac pasy ***
```

21.23.1 Mechanizm automatycznej rezerwacji miejsca

W poprzednim paragrafie poznaliśmy sposób posługiwania się strumieniem płynącym do tablicy znakowej. To, czego się nauczyliśmy, zaspokoi wszystkie nasze potrzeby na ten temat. W tym paragrafie mówić będziemy o tym, jak to robić jeszcze sprytniej.

W naszym ostatnim przykładzie konstruktor strumienia `skryba` wiedział, że pracuje na tablicy o pojemności 40 znaków. Próba wpisania tam, dłuższego tekstu uznana zostaje za błąd. Nie zawsze jednak wiadomo ile znaków ma być wpisywanych do tablicy – czyli jak duża powinna być tablica, by nas zadowolić.

Pomyślałeś pewnie, że wtedy trzeba zarezerwować tablicę z zapasem – tak, by na pewno pomieściła cały wpisywany tekst. Nie jest to jednak dobre rozwiązanie. Wyobraź sobie, że masz wpisać do tablicy nie jedno zdanie, ale bardzo długi tekst. Możliwe, że długość tego tekstu nie jest nawet jeszcze w przybliżeniu znana – tekst może pochodzić z pliku dyskowego znanego dopiero w trakcie wykonywania programu. Kłopot.

Klasa `ostrstream` oferuje bardzo ciekawe narzędzia do rozwiązania tego kłopotu.

Otóż definiuje się strumień `ostrstream` bez podawania mu z jaką tablicą ma pracować. Oto przykład takiej definicji:

```
ostrstream skryba ;
```

Widząc taką definicję, strumień uruchamia swój domniemany (czyli bezargumentowy) konstruktor. Powstały w ten sposób obiekt – ośrodek dowodzenia takim strumieniem – wie, że musi sam zajmować się rezerwacją miejsca. Sam ma zdefiniować sobie tablicę odpowiedniego rozmiaru.

Nic w tym szczególnego – po prostu strumień używa sobie operatora `new`. Tak zdefiniowana tablica jest ujściem strumienia, czyli tam wpisuje on znaki.

Najciekawsze jest jednak to, że rezerwuje się „tylko trochę" pamięci. Odbywające się kolejne wpisania tekstów do tej tablicy – np.

```
skryba << " Jakis napis " ;
skryba << " Inny dlugi tekst 111111111111112222222222 " ;
```

mogą spowodować w końcu zapełnienie całej zarezerwowanej tablicy. To jednak nas nie obchodzi – ośrodek dowodzenia strumieniem zajmie się tym sam. (Widzisz jak wspaniałe są klasy ?!)

Dzieje się to tak:

Jeśli zarezerwowana tablica okazuje się za mała, `skryba` zarezerwuje sobie nową, większą. Do tej większej przepisuje cały ten tekst, który do tej pory był w „starym" miejscu. W rezultacie więc cała treść jest przepisana do nowej, o wiele większej tablicy – jest tam jeszcze dużo pustego miejsca na dalsze ewentualne

operacje pisania. Stara tablica jest już niepotrzebna, więc można ten obszar pamięci zwolnić (zwykła instrukcja `delete`).

Gdyby jednak i ta nowa tablica została w rezultacie dalszych operacji całkowicie zapełniona – procedura powtórzy się. Najważniejsze jest jednak to, że wszystkie to dzieje się bez nas, nie musimy wcale o tym wiedzieć.

Przekonajmy się na przykładzie jak prosto się to realizuje

```
#include <iostream.h>
#include <iomanip.h>
#include <strstream.h>
/*****************************************************/
main()
{
ostrstream skryba ;                                    // ❶

     skryba << "To jest poczatek " ;

     for(int i = 0 ; i < 100 ; i++)
     {
          skryba << "\nOperacja nr " << i ;            // ❷
          if(!skryba)
          {
               cout << "Blad strumienia skryba " ;
               return (1);          // koniec programu
          }
     }
     skryba << ends ;               // dopisujemy znak NULL

     char *wsk ;
     wsk = skryba.str() ;                              // ❸

     cout << "Odebralem juz adres do tablicy "<< endl
          << "oto jej tresc : \n"
          << wsk
          << "\nto juz koniec \n" ;

     delete wsk ;                                      //❹
}
```

❐ **Po wykonaniu tego programu na ekranie zobaczymy poniższy tekst**

Ze względu na oszczędność miejsca nie zamieszczam go w całości – rząd kropeczek oznacza miejsce skrótu.

```
Odebralem juz adres do tablicy
oto jej tresc :
To jest poczatek
Operacja nr 0
Operacja nr 1
Operacja nr 2
..............
Operacja nr 98
Operacja nr 99
```

```
to juz koniec
```

Komentarz

❶ Definicja ośrodka dowodzenia strumieniem klasy `ostrstream`. Brak argumentów dla konstruktora oznacza, że chodzi nam o tablicę rezerwowaną automatycznie.

❷ Operacja wielokrotnego wpisywania do tablicy. Wygląda identycznie, jak przy wkładaniu informacji do zwykłego strumienia wyjściowego.
Pytanie: Widzimy tu wielokrotne operacje wkładania do strumienia `skryba`. Gdzie właściwie płyną nasze znaki?

Odpowiedź: Nie wiadomo. Jest to prywatna sprawa strumienia `skryba`. Zdenerwujesz się pewnie: „–No, ale chyba mam prawo wiedzieć! – w końcu będę potrzebował tych znaków!"

Nie bądź taki niecierpliwy. Na etapie kolejnych operacji wkładania do strumienia `skryba`, tablica do której wpisujemy, może być w pamięci raz tu a raz tam. Taki jest bowiem mechanizm ewentualnego powiększania jej rozmiarów.

Zapytasz: „–Czy nigdy się nie dowiem, gdzie znajduje się wypisywany tekst?". Dowiesz się, dowiesz. Przeczytaj to, co poniżej.

❸ W momencie, gdy uznajemy, że wypisaliśmy wszystko co było do wypisania – wywołujemy na rzecz strumienia `skryba` funkcję składową `str` (- zapewne skrót od: string). Jest to funkcja składowa klasy `ostrstream` informująca nas o tym, gdzie **ostatecznie** znajdują się wszystkie wypisywane przez nas znaki.

```
char* str() ;
```

Rezultatem tej funkcji jest adres tablicy, na której pracował strumień i w której obecnie znajdują się znaki.

Wywołanie tej funkcji ma ważną konsekwencję.

> Od tej pory strumień `skryba` – że tak powiem – „umywa ręce" od dalszych operacji i „przechodzi na emeryturę".

Trzeba go zrozumieć: skoro powiedział nam już gdzie tablica się znajduje, to musi być konsekwentny. Gdyby pozwolił nam na dalsze wpisywanie do niej, to możliwe, że natychmiast byśmy ją zapełnili. On jednak nie mógłby wtedy zrobić zwykłej akcji z rezerwacją innej, większej, bo już nam przekazał jej adres – i teraz musi dotrzymać słowa.

❹ Mówiąc, że strumień po wykonaniu funkcji `str` „umywa ręce" – miałem na myśli coś więcej. On od tej pory nie tylko, że nie wykona dalszych operacji wpisywania do tablicy, ale jeszcze dodatkowo uzna, że „sprzedał" nam tę tablicę.
Mamy wskaźnik do niej i jest już naszą sprawą ewentualne późniejsze zlikwidowanie jej instrukcją `delete`. Od tego właśnie skryba „umywa ręce".

Co by jednak było, gdybyśmy się rozmyślili i tej tablicy od strumienia nie odkupili – czyli, gdybyśmy nigdy nie wykonali na tym strumieniu funkcji:

```
skryba.str()
```

Wówczas strumień byłby przez cały czas w gotowości do wykonania na tej tablicy dalszych operacji. Wreszcie w momencie, gdy kończyłby się zakres ważności obiektu klasy strumień – jego destruktor skasowałby tę tablicę. To zrozumiałe: skoro nie odsprzedał nikomu tej tablicy, więc czuł się dalej odpowiedzialny za jej los.

21.23.2 Anonimowy strumień wyjściowy

Jest to inne usprawnienie dla wygody posługiwania się strumieniami do wewnętrznego formatowania.

Chodzi tutaj o sytuację, gdy z danym strumieniem zamierzamy pracować tylko w jednej jedynej instrukcji programu. Skoro później nie zamierzamy już odnieść się do jego usług, zatem nazwa jest strumieniowi niepotrzebna.

Dlatego strumień rzeczywiście nie ma nazwy. Jest anonimowy. Oszczędzamy sobie w ten sposób trudu wymyślania nazwy dla czegoś, co za chwilę będzie nam niepotrzebne. Warunkiem jest jednak załatwienie sprawy w jednej instrukcji.

Oto przykład:

```
     char tablica [20] ;
     int  sekc = 44 ;

ostrstream(tablica, 20) << " Wylacz napiecie sekcji nr:"
                        << sekc << ends ;
```

W rezultacie w tablicy znajdzie się żądany tekst. Użyty do tego celu strumień będzie tu jakby obiektem chwilowym. Przy najbliższej okazji zostaje zlikwidowany.

21.24 Formatowanie wewnętrzne – operacje wejścia

Za pomocą strumieni można też wyczytywać informację znajdującą się w jakiejś tablicy znakowej. Tak samo, jakby znaki te przychodziły z klawiatury lub pliku.

Innymi słowy jeśli mamy tablicę

```
     char schowek[40] = { "7.15" } ;
```

w której zapisany jest pewien ciąg znaków – (u nas znaki: `'7','.','1','5',` `NULL`) – to możemy te znaki wczytać i zinterpretować tak, jakby były pisane na klawiaturze i wczytywane instrukcją

```
     float liczba ;
          cin >> liczba ;
```

Aby tego dokonać, posługujemy się strumieniem, który płynie nie od klawiatury, a od tablicy znakowej `schowek`. Klasa takich strumienia nazywa się `istrstream` (Input STRing STREAM – czyli strumień wczytujący z tablicy znakowej).

Zobaczmy to na przykładzie. Aby było ciekawiej, w tablicy jest jeszcze dodatkowa treść.

```
#include <iostream.h>
#include <strstream.h>                                          // ❶
/*****************************************************/
main()
{
char schowek[40] = { "7.15 wtorek" } ;
istrstream potok(schowek, sizeof(schowek) );                    // ❷

float liczba ;
char wyraz[40] ;

     potok >> liczba >> wyraz ;                                 // ❸

     liczba = liczba + 1 ;

     cout << "Na dowod, ze sie udalo :\n"                        // ❹
          << "Biezaca wartosc zmiennej liczba = " << liczba
          << "\nnatomiast wyraz brzmi = " << wyraz ;

     // demonstracja mozliwosci pozycjonowania
     potok.clear( potok.rdstate() & ~ios::eofbit) ;             // ❺
     potok.seekg(1, ios::beg) ;                                 // ❻
     char c ;

     potok.get(c) ;

     cout << "\nwydobyty znak c = " << c << endl ;

     potok.seekg(4, ios::cur) ;
     potok.get(c) ;
     cout << "Czwarty znak dalej to = " << c << endl ;
}
```

Po wykonaniu programu na ekranie pojawi się

```
Na dowod, ze sie udalo :
Biezaca wartosc zmiennej liczba = 8.15
natomiast wyraz brzmi = wtorek
wydobyty znak c = .
Czwarty znak dalej to = t
```

Komentarz

❶ Włączenie pliku nagłówkowego niezbędnego przy pracy ze strumieniami odpowiedzialnymi za formatowanie wewnętrzne. Zawarta jest tam m.in. definicja klasy `istrstream`.

❷ Definicja ośrodka dowodzenia – będącego egzemplarzem obiektu klasy `istrstream`. Widzimy też argumenty dla konstruktora – nazwę tablicy, która ma być dla strumienia `potok` źródłem wczytywanych znaków, oraz argument określający jej długość.

❸ Właściwa akcja wczytywania znaków z tablicy `schowek`. Wygląda ona tak samo, jak zwykła operacja wczytywania z klawiatury

```
cin >> liczba >> wyraz ;
```

❹ Na dowód, że ta akcja odbyła się poprawnie wypisujemy na ekran to, co wczytaliśmy ze `schowka`.

❺ Dodatkowo pobawimy się jeszcze pozycjonowaniem wskaźnika czytania, który jest nam dostępny także i w tej klasie. Najpierw jednak trzeba pamiętać o tym, że instrukcją ❸ wczytaliśmy wszystko, co było do wczytania – nie ma już dalszych danych, które moglibyśmy wczytać. Skoro tak, to w ośrodku dowodzenia strumieniem `potok` została ustawiona flaga `ios::eofbit`, oznaczająca jakby koniec „pliku" danych w tej tablicy.
Teraz jednak będziemy chcieli wrócić na początek tablicy i przeczytać coś jeszcze raz. Zanim to zrobimy, musimy skasować flagę błędu w naszym strumieniu. To właśnie robione jest w tej linijce.

❻ Pozycjonowanie wskaźnika czytania z tej tablicy na elemencie

```
schowek[1] ;
```

Następnie znak ten czytamy i dla kontroli wypisujemy na ekranie. W dalszych linijkach programu widzimy także inny sposób pozycjonowania.

Wszystko to służy przekonaniu Cię, że strumieniem tym można wykonywać operacje tak samo, jakby płynął z pliku.

W przykładzie wspominaliśmy o konstruktorze tego strumienia

```
istrstream::istrstream(char*, int );
```

Drugi argument określa rozmiar tablicy, z którą strumień ma pracować. Możemy jednak posłużyć się inną wersją konstruktora

```
istrstream::istrstream(char *) ;
```

Co w naszym wypadku wyglądałoby tak

```
istrstream potok(schowek);
```

Konstruktor ten stosujemy, gdy strumień ma pracować z tablicą znakową, w której zapisany jest rzeczywiście string – czyli zbiór znaków, a ostatnim znakiem jest znak NULL. (Nie jest to obowiązkowe). Wówczas nie jest potrzebne określenie rozmiaru stringu – gdyż strumień potrafi sobie to sam obliczyć odszukując ten znak NULL.

W naszym przykładzie z tablicą `schowek` mieliśmy w niej rzeczywiście string – była przecież inicjalizowana tekstem ujętym w cudzysłów. Mogliśmy więc użyć obu wariantów konstruktora

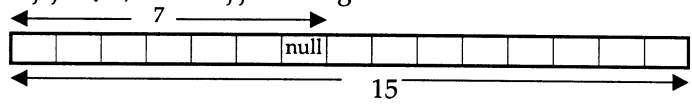
```
istrstream potok(schowek) ;
istrstream potok(schowek, sizeof(schowek) );
```

Czy jest zatem jakaś różnica w zastosowaniu tych konstruktorów?

Tak. W pierwszym wypadku strumień obliczy długość stringu – (która może być krótsza niż rozmiar tablicy – czyli niż wyrażenie `sizeof(schowek)`.

Ma to wpływ na późniejsze ewentualne pozycjonowanie wskaźnika czytania. Wskaźnik ten nie może znaleźć się za końcem obszaru pracy (czyli dla niego

jakby: „pliku"). Obszar ten w drugim wypadku to cała tablica schowek, a w pierwszym tylko ta jej część, w której jest string.



schowek

Próba pozycjonowania poza obszarem pracy uznana zostanie za błąd `ios::eofbit`. Dobrze jest też pamiętać, że poniższa instrukcja pozycjonowania wskaźnika (kursora) czytania na którymś znaku od **końca**

```
potok.seekg(6, ios::end) ;
```

będzie miała różny efekt w jednym i drugim wypadku. To dlatego, że w obu wypadkach co innego uznawane jest za koniec „pliku".

21.24.1 Przykładowe zastosowanie oraz anonimowy strumień wejściowy

Pokazałem już jak wczytywać coś z tablicy znakowej. Możesz zapytać: „Dobrze, ale po co? Po co zapisywać liczbę w tablicy znakowej za pomocą ciągu znaków – a za chwilę przetłumaczyć te znaki na liczbę. Nie można od razu?"

Czasem rzeczywiście nie można. Na przykład jeśli do programu przesyłasz jakąś liczbę z linijki wywołania programu

```
program 6.33 slowo 777.1
```

to te trzy parametry dostają się do wnętrza programu jako stringi. (Mówiliśmy o tym na str. 223).

W programie natomiast, chcemy tak podane liczby umieścić w obiektach typu `int` czy `float`. Tak samo jakbyśmy wczytywali je z klawiatury. Musimy więc dokonać wczytania z formatowaniem – czyli właśnie formatowaniem wewnętrznym.

Oto jak to łatwo zrobić:

```
#include <iostream.h>
#include <strstream.h>
/*****************************************************/
main(int argc, char* argv[])                          // ❶
{
float x, z ;
char nazwisko[40] ;

istrstream s1(argv[1]) ;                              // ❷
           s1 >> x ;                                  // ❸

istrstream s2(argv[2]) ;                              // ❹
           s2 >> nazwisko ;

// wygodniej anonimowo !

istrstream(argv[3]) >> z ;                            // ❺
```

```
        cout << "Oto wartosci przyjetych parametrow\n"
             << x << endl
             << nazwisko << endl
             << z << endl
             << "suma tych liczb = " << (x+z) << endl ;
}
```

W rezultacie na ekranie zobaczymy

```
Oto wartosci przyjetych parametrow
6.33
Goethe
777.099976
suma tych liczb = 783.429993
```

Komentarz

❶ Jak pamiętamy, przysłane z linii wywołania programu parametry dostępne są w specjalnych tablicach. Następujące wywołanie programu

```
program 6.33 Goethe 777.1
```

spowoduje umieszczenie ich w tablicach i przysłanie ich do funkcji `main` jako jej argumenty. Jeśli odbierzemy je w taki sposób, jak to jest widoczne w ❶ to w rezultacie mamy tablicę, w której elementami są wskaźniki do następujących stringów

`argv[0]` – zawiera wskaźnik do stringu – `"program"`

`argv[1]` – zawiera wskaźnik do stringu – `"6.33"`

`argv[2]` – zawiera wskaźnik do stringu – `"Goethe"`

`argv[3]` – zawiera wskaźnik do stringu – `"777.1"`

Natomiast argument `argc` (argument counter) określa liczbę takich stringów — w naszym wypadku 4.

❷ Definicja strumienia, który wczyta coś ze stringu `argv[1]`.

❸ Moment wczytania ze stringu. W rezultacie w `x` znajdzie się liczba. Strumień `s1` spełnił swoje zadanie - jest więc odtąd niepotrzebny. Mogę go zamknąć instrukcją `s1.close()` - jeśli nie uczynię tego - zrobi to za mnie destruktor obiektu `s1`.

❹ To samo przy wczytaniu następnego parametru. Po wczytaniu strumień `s2` staje się niepotrzebny.

❺ Skoro strumień potrzebny jest nam na użytek tylko jednej instrukcji, to definiujemy go jako strumień anonimowy - czyli nie dajemy mu żadnej nazwy (por. paragraf o anonimowych strumieniach wyjściowych). Nazwa strumieniowi jest potrzebna tylko do tego, by potem można było się do niego odwołać. Tyle, że tutaj nie ma żadnego „potem". Definiujemy strumień i od razu wczytujemy z tablicy – wszystko w jednej instrukcji.
Oczywiście gdybyśmy chcieli wczytać dopiero w następnej instrukcji (tak jak w ❷ i ❸), wówczas nazwa byłaby potrzebna.

Anonimowe strumienie upraszczają nam często życie. Nie zawracamy sobie głowy nadawaniem nazwy czemuś, co i tak za chwilę nie będzie nam już potrzebne.

21.25 Ożenek: klasy wejściowo – wyjściowe dla formatowania wewnętrznego

Istnieje klasa pochodna `strstream` będącą pochodną klas `istrstream` oraz `ostrstream`. Jest to strumień, który może zarówno pisać, jak i czytać z tablicy znakowej. Klasa ta ma takie właściwości, jak strumień do pliku otwartego do czytania i pisania równocześnie.

```
strstream::strstream(char* tablica, int rozmiar, int tryb);
```

Argumenty konstruktora są identyczne jak konstruktora klasy wyjściowej `ostrstream`. Jest też dostępna funkcja `str`, która pozwala nam gromadzić znaki w tablicy, której rozmiaru nie określamy.

Oto przykład:

```
#include <iostream.h>
#include <strstream.h>
/*******************************************************/
main()
{
char pasek[80] ;

strstream p(pasek, sizeof(pasek), ios::in | ios::out);

     p << "0x33 10 44.55" ;

int i, j ;
float x ;

     p >> i ;
     p >> j >> x ;

     cout << "suma = "<< (i + j + x) << endl ;

                    // można także pozycjonować kursory pisania i czytania...
     p.seekg(2, ios::beg) ;

                    // ... i stosować funkcje składowe klas podstawowych
char t[10] ;
     p.getline(t, 8) ;
     t[6] = NULL ;
     cout << "Znaki wyjete ze srodka = " << t << endl ;
}
```

Wykonanie objawi się na ekranie jako

```
suma = 105.550003
Znaki wyjete ze srodka = 33 10
```

21.26 Jeszcze o strumieniach anonimowych

Co prawda ze strumieniami anonimowymi spotkaliśmy się przy formatowaniu wewnętrznym - jednak strumień anonimowy nie musi mieć nic wspólnego z tym zagadnieniem.

Strumieniem anonimowym może być dowolny strumień - na przykład taki zapisujący na dysku (lub czytający). Jest tylko jeden warunek: sprawę należy załatwić w jednej instrukcji.

Zatem jeśli chcemy strumieniem anonimowym zapisać coś do zbioru dyskowego, to powinniśmy to zrobić tak:

```
#include <iostream.h>
#include <fstream.h>
    //...
    ofstream("zemsta.txt", ios::out) << "Mocium Panie!";
    //...
```

W rezultacie założony zostanie plik o nazwie "zemsta.txt", a jego treścią będzie: Mocium Panie!

Drukowanie przez program tekstu na drukarce (IBM PC)

Dwóch czytelników - jeden z Polski, drugi z Kanady - zapytało mnie w listach o tę sprawę, po pierwszym wydaniu tej książki. O co chodzi dokładnie:

Jak niektórym wiadomo, w bibliotece `stdio` do drukowania przez program tekstu na drukarce istnieje instrukcja

```
#include <stdio.h>
    //....
    fprintf(stdprn, "Mocium Panie!");
```

Skoro tak odradzam posługiwanie się biblioteką `stdio`, to jak to samo sprytnie zrobić za pomocą biblioteki `iostream`?

Oczywiście pamiętasz moje słowa, że biblioteka `iostream` nie jest częścią definicji języka i mogą wystąpić różnice w jej implementacjach na różnych komputerach.

> *Zapewne teraz pomyślałeś, że w końcu każdy typ komputera ma jakąś drukarkę. Rzeczywiście, jednak sprawa nie jest tak prosta, bo drukarka może być wspólna dla wielu użytkowników i znajdować się w innym budynku. Aby na niej pracować trzeba ustawiać się w kolejce (w systemie operacyjnym).*

To dlatego pewnie w bibliotece `iostream` nie ma czegoś, co odpowiada instrukcji

```
fprintf(stdprn, "Mocium Panie!") ;
```

Po prostu strumień płynący do drukarki nie jest predefiniowany. Zatem aby coś takim strumieniem popłynęło musimy go otworzyć, pisać, a potem zamknąć.

Za dużo pracy? Nie, nie są tu potrzebne te trzy grupy instrukcji! **Dzięki anonimowemu strumieniowi wyjściowemu wszystko to można załatwić w jednej instrukcji.**

Ponieważ na IBM PC drukarka, to po prostu plik: "prn", zatem odpowiednia instrukcja wygląda tak:

```
ofstream("prn", ios::out) << "Mocium Panie!" ;
```

Jeśli nie chce Ci się stukać tylu znaków przy pisaniu programu, możesz sobie to, co jest po lewej stronie znaków <<, zdefiniować jako tekst DRUKARKA. Załączam przykładowy program, w którym są oba sposoby. Zauważ, że co prawda anonimowy strumień wyjściowy żyje tylko w przeciągu jednej instrukcji - ale w tej instrukcji może być wiele żądań wypisania. Taka przykładowa instrukcja ciągnie się u mnie przez 3 linijki.

Potem strumień przestaje istnieć, ale gdy znowu chcemy coś wydrukować - można zdefiniować następny - też anonimowy.

```
#include <iostream.h>
#include <fstream.h>          // <--- nie zapomnij o tym !
#define DRUKARKA ofstream("prn", ios::out)
/*****************************************************/
main()
{
float liczba = 3.14 ;
 ofstream("prn", ios::out) << "Mocium Panie!" << endl ;

 ofstream("prn", ios::out)    << "Jakis tekst " << liczba
                              << " znowu tekst " << 8+43.2
                              << " po liczbie " << endl ;
     // Sposób dla leniuchów ---------------------------

     DRUKARKA << "Mocium Panie! " << endl ;

     DRUKARKA << "Jakis tekst " << liczba
              << " znowu tekst " << 8+43.2
              << " po liczbie " << endl ;
}
/*****************************************************/
```

22 Projektowanie programów obiektowo orientowanych

Dotarliśmy wreszcie do ostatniego rozdziału tej książki. Wiemy już jak posługiwać się językiem C++. Masz oto w ręce narzędzie do programowania obiektowo orientowanego – wiesz jak tego narzędzia używać, nie wiesz tylko kiedy i po co.

Przyznam się, że gdy świeżo nauczyłem się C++, miałem ten sam problem: wiedziałem jak się buduje klasy czy definiuje obiekty, nie wiedziałem tylko co, w przypadku pisania konkretnego programu, mam zdefiniować jako klasę i co robić z jej obiektami.

W tym rozdziale porozmawiamy właśnie o tym, jak w przypadku pisania konkretnego programu wymyślić klasy, które nam uproszczą i uprzyjemnią programowanie. Inaczej mówiąc będziemy tu mówić, nie tyle o obiektowo orientowanym *języku* programowania, co o obiektowo orientowanej *technice* programowania.

Tradycyjne techniki programowania, którymi posługiwałeś się Czytelniku w innych znanych Ci językach programowania, pozwalały Ci na pisanie małych i dużych programów z lepszym czy gorszym rezultatem. Teraz zaś pojawia się nowa technika programowania: programowanie obiektowo orientowane. Co to oznacza w praktyce? Czy wobec tego tamte techniki nie są już ważne?

Odpowiedź jest taka: nie musisz wcale stosować techniki obiektowo orientowanej. Nawet programując w C++ możesz stosować stare sposoby, do których jesteś tak przywiązany. To tak, jak ze starym poczciwym chirurgiem, który polega tylko na instrumentach chirurgicznych znanych mu od czasu studiów. Jest rzeczą bardzo delikatną wytłumaczenie mu, by używał narzędzi nowoczesnych.

Jesteś dla mnie właśnie takim chirurgiem. Spróbuję Cię przekonać, że programując techniką obiektowo orientowaną zaoszczędzisz wiele wysiłku. Jeśli jesteś

menagerem przewodzącym zespołowi programistów, to stosowanie przez Twój zespół tej techniki pozwoli zmniejszyć koszt pracy nad projektem.

22.1 Przegląd kilku technik programowania

Aby sobie dobrze uświadomić czym jest w naszych rękach obiektowo orientowany język C++, najpierw przyjrzyjmy się kilku podstawowym technikom programowania, które prawdopodobnie stosujesz. Techniki te przedstawiam w porządku chronologicznym, a więc jest to historia o tym, jak programowanie stawało się coraz łatwiejsze.

22.1.1 Programowanie liniowe

- mówiąc obrazowo polegało na tym, że poszczególne instrukcje programu pisało się jedna pod drugą. Na górze był początek programu, gdzieś na dole koniec programu, a pomiędzy – wszystkie instrukcje – łącznie z dziesiątkami lub setkami instrukcji `goto`.

Program pracował na danych, które wszystkie były dostępne w czasie całego przebiegu programu – dziś nazwalibyśmy je globalnymi. Nie było zmiennych lokalnych; jeśli raz w jakiejś definicji użyliśmy nazwy zmiennej `k` – to przepadło – nigdzie już definicja tej nazwy nie mogła się powtórzyć. Obecność licznych instrukcji skoków `goto` sprawiała, że dłuższy program był niezrozumiały dla innego programisty - był to kod, który nazwano obrazowo: „kod a la spaghetti".

Jeśli programujesz w języku BASIC (w wersji bez instrukcji GOSUB), to stosujesz właśnie tę technikę programowania. Powiedzmy dosadniej: jeśli programujesz w BASICU to znaczy, że zatrzymałeś się na tym etapie.

22.1.2 Programowanie proceduralne

Gdy powstało – od razu uznane zostało za ogromny krok w nowoczesność. Technika ta cechowała się tym, że w programie można było wyodrębnić części realizujące pewne określone czynności. Tymi częściami były procedury (funkcje). W ramach takiej jednej funkcji można było definiować dodatkowe zmienne – tzw. zmienne lokalne. Lokalne – gdyż dostępne tylko dla niej.

Przykładem języka programowania wymuszającego tę technikę programowania jest język FORTRAN.

Ogólnie mówiąc programowanie proceduralne polega na tym, by pisząc program dla danego problemu, zamienić ten problem na serię zadań do wykonania. Te konkretne zadania realizują właśnie funkcje. Wykonywanie takiego programu, to określona sekwencja wywołań różnych funkcji.

Napisanie dobrego programu tą techniką polegało na tym, by zdecydować jakie funkcje będą potrzebne, następnie zdefiniować je, po czym starać się wymyślić najlepszy algorytm wywoływania ich.

Niewątpliwą wadą tej techniki programowania jest to, że zajmuje się ona głównie funkcjami, a nie dba o to, czy poszczególne dane w programie są ze

sobą związane czy też rozrzucone w programie w luźny sposób. Wracając do FORTRANU: – jedyną sytuacją, gdy dane mogą być tam ze sobą związane jest umieszczenie ich w tablicy. Tablica jednak nie może zawierać danych, które są różnego typu. (W tablicy nie może być np. tak, że pierwszy element jest liczbą zmiennoprzecinkową, a drugi stringiem).

Jest to do dziś bardzo popularna technika. Wielu moich kolegów fizyków zatrzymało się właśnie na tym etapie.

22.1.3 Programowanie z ukrywaniem danych

Jest to krok do przodu w stosunku do techniki poznanej poprzednio. Polega on na wzbogaceniu programu w możliwość zapakowania danych w jednym module. W ten sposób dane te mogą być traktowane jako grupa – pewna całość.

Tę technikę programowania stosujesz, jeśli programując w PASCAL'u posługujesz się tzw. rekordami, względnie gdy programując w języku C (klasycznym), posługujesz się strukturami.

Jeśli nie masz doświadczeń z żadnym z tych języków programowania, a chciałbyś tę technikę sobie jakoś uświadomić, to posłużę się terminami z poznanego już języka C++: jest to tak, jakbyś posłużył się klasą, która ma tylko składniki dane – a wszystkie publiczne.

W czym jest lepsza ta technika od poprzednio omówionej? Chociażby w tym, że wywołując funkcję możesz wysłać do niej wszystkie dane naraz. Moduł taki (struktura), przesyłany jest jako jeden argument. Ważniejsze jednak, że dane, które „w życiu" powiązane są ze sobą logicznie - są także powiązane ze sobą w programie.

Jeśli jesteś na tym etapie programowania – to z łatwością przyjdzie Ci przestawienie się na technikę obiektowo orientowaną.

22.1.4 Programowanie obiektowe - programowanie „bazujące" na obiektach

Jest w Polsce często mylone z programowaniem obiektowo orientowanym. Angielską nazwę tej techniki konstrukcji programów: object based design - można przetłumaczyć jako: programowanie bazujące na obiektach. W stosunku do techniki poprzedniej – innowacja polega na tym, że zgrupowanym danym dodaje się metody postępowania z nimi – my w terminach C++ powiedzielibyśmy: funkcje składowe. Powstaje wtedy nie tylko moduł kryjący w sobie dane, ale wręcz nowy typ danej. Są to ukryte w module dane, plus możliwe działania na nich.

Jeśli programujesz w języku Ada, to jesteś właśnie na tym etapie.

Programowanie tą techniką cechuje się tym, iż danym zgrupowanym w moduł (obiekt), nie mówi się już **jak** mają coś zrobić, ale tylko **co** mają zrobić. (To: jak mają robić, wiedzą na podstawie swoich funkcji składowych).

Niedogodnością tej techniki jest fakt, że poszczególne typy danych są sobie obce. Typ danej opisujący np. samochód nie ma nic wspólnego z typem opisującym pojazd. Jest mu tak samo obcy, jak liczbie zmiennoprzecinkowej. Znaczy to, że jeśli funkcja może przyjąć jako argument jeden typ danej, to nie może przyjąć

innego. Dla tego innego typu musi być zdefiniowana osobna funkcja, choćby jej ciało miało być identyczne jak już istniejącej. W związku z tym modyfikacja programu polegająca na dodaniu ulepszonego typu danej w stosunku do typu już istniejącego – jest bardzo kłopotliwa i wymaga zdefiniowania od nowa wszystkich funkcji, na których ma on pracować.

22.1.5 Programowanie Obiektowo Orientowane (OO)

Najchętniej nazywałbym to: programowanie obiektowo orientowa(L)ne. Jest to technika obiektowa wzbogacona o dziedziczenie i polimorfizm (czyli funkcje wirtualne). Sprawia to, że istniejący kod potrafi się sam zorientować w trakcie pracy programu z jakim obiektem przyszło mu w danej chwili pracować i odpowiednio na to reaguje. Jest to jakaś „wyższa inteligencja" dodana do programu. Nie zrozum źle nazwy tej techniki: to nie my programując jesteśmy „zorientowani na obiekty". To kod programu jest zdolny się sam zorientować z jakim obiektem pracuje – dlatego napisałem orientowa(L)ne. Nie lansuję tej nazwy, bo na pewno się nie przyjmie – ale powtarzaj sobie to czasem.

Tego typu technika doskonale odzwierciedla zależności klas obiektów otaczających nas w realnym świecie. To dlatego tak dobrze nadaje się do opisania świata realnego.

Mimo, że lubię powtarzać, iż wprowadzenie techniki obiektowo orientowanej jest rewolucją w programowaniu – to sam chyba widzisz, że jest to raczej ewolucja: technika OO jest logicznym rozwinięciem technik poprzednich.

Język C++ nie jest jedynym, który umożliwia tę technikę programowania. Ma on jednak jedną cechę szczególną:

Można w nim stosować technikę OO w stopniu, który się już opanowało - lub nie stosować jej wcale. Mówimy – język ten jest językiem hybrydowym – umożliwia nawet programowanie „pół na pół" – trochę tej techniki, trochę tamtej. To dlatego C++ jest tak popularne. Umożliwia łagodne przejście w świat programowania OO. Z techniki OO stosujemy tylko to, czego się do tej pory nauczyliśmy. Nie musimy od razu umieć wszystkiego. W języku C++ można napisać program techniką proceduralną, lub nawet - o zgrozo – techniką liniową. Programiści C zaczynając C++ wybierają sobie najpierw z niego tylko jakieś rodzynki – np. referencje, potem nieśmiało oswajają się z coraz bardziej wyszukanymi elementami.

Inne języki programowania – takie, które wymagają od razu wejścia na głęboką wodę i stosowania programowania obiektowo orientowanego od razu – mają moim zdaniem mniejsze szanse na taką popularność.

22.2 O wyższości programowania obiektowo orientowanego nad Świętami Wielkiej Nocy

Zbierzmy sobie te cechy programowania OO, które sprawiają, że jest ono lepsze od tradycyjnych technik programowania, którymi zapewne posługiwałeś się do

tej pory. (Przez tradycyjne techniki uznaję te, które możliwe są w klasycznym języku C).

Będzie to powtórzenie rzeczy, poznanych w różnych miejscach tej książki. Teraz zobaczmy je na tle technik, którymi posługiwałeś się do tej pory.

Reusability – wtórne użycie wcześniej zdefiniowanych fragmentów kodu

Tradycyjne metody programowania nie kładły nacisku na możliwość wtórnego użycia kodu funkcji. Nie chodzi o to, że funkcję można wywołać dwa razy - bo przecież można wywołać nawet milion razy – to żaden cud.

O co więc chodzi? Otóż w dotychczasowych technikach programowania jeśli funkcja przyjmowała jako argument strukturę o nazwie `pojazd`, to nie mogła zostać wywołana dla struktury `samochod`. Dla samochodu trzeba było napisać identyczną funkcję jeszcze raz, tyle, że ze zmienionym typem argumentu formalnego.

Programowanie OO umożliwia i wręcz zachęca do wtórnego używania kodu funkcji. Jeśli tylko klasy (pojazd – samochód) są połączone związkami dziedziczenia, to funkcja pracująca na argumencie pojazd nadaje się do pracy z argumentem klasy samochód. Oszczędzamy w ten sposób wysiłku powtórnego kodowania funkcji, program staje się przez to mniejszy, a wreszcie jakakolwiek późniejsza zmiana postępowania w stosunku do pojazdów nie wymaga wprowadzania tych samych zmian postępowania w samochodach – samochody są przecież także pojazdami.

Extensibility – rozszerzalność, rozbudowalność

Tradycyjne metody programowania bardzo utrudniają ewentualne późniejsze modyfikacje programu. Jeśli program pracuje na obiektach będących np. różnymi typami samochodów, to w wielu miejscach programu są zapewne takie fragmenty kodu:

> Jeśli to Renault – to zrób tak...
> Jeśli to Porsche – to zrób siak...
> Jeśli Rolls-Royce – to zrób owak...

Takie fragmenty są rozsiane po całym programie. Wyobraź sobie, że program powstawał w większym zespole i pewnego dnia Twój szef mówi Ci, że wprowadza się do programu nową markę samochodu – Ferrari. Musisz teraz zaangażować wszystkich ludzi po to, by w swoich częściach odnaleźli w programie wszystkie te liczne miejsca i dopisali linijkę:

> Jeśli Ferrari to zrób tak...

Wyższość techniki obiektowo orientowanej polega na tym, że żadnych takich modyfikacji robić nie trzeba. Dzięki polimorfizmowi (wywoływaniu funkcji wirtualnych) istniejący już kod potrafi zareagować na nowe typy obiektów. To dlatego, że w takim programie obiektowo orientowa(l)nym nie ma wcale linijek typu:

> Jeśli Renault to tak...

wymieniających wszystkie samochody, na których program pracuje. Jest za to w takim miejscu tylko jedna linijka:

> Dla tego konkretnego typu samochodu (sam się zorientuj jakiego) zrób przewidzianą dla niego funkcję.

Wierz mi, korzyść z takiego rozwiązania oblicza się w tysiącach dolarów. Doświadczyłem tego osobiście.

Dawniej (w latach siedemdziesiątych), gdy kwitło programowanie proceduralne, wymóg łatwej modyfikowalności nie był jeszcze tak istotny. Dziś – gdy programy stały się ogromnie rozbudowane i pracują nad nimi równocześnie dziesiątki ludzi nie bardzo wiedzących co robią koledzy z innego zespołu – dziś musi istnieć łatwość wszelkich modyfikacji. Każda godzina pracy nad projektem to ogromne koszty.

Modelowanie świata rzeczywistego

Jeśli program odnosi się do jakiegoś zagadnienia (systemu) ze świata rzeczywistego, to pisanie programu metodą proceduralną polega na rozbiciu danego zagadnienia na zbiór liczb oraz funkcji, które na tych liczbach pracują. Po prostu sama matematyka.

Programowanie w technice obiektowo orientowanej polega na zbudowaniu w programie modeli obiektów ze świata rzeczywistego (np. czeków bankowych, pionków do gry, albo wnętrza układów elektronicznych). Praca programu to ożywienie tych obiektów i pozwolenie im na odegranie swojej roli. O tym wszystkim szczegółowo mówić będziemy na następnych stronach.

Łatwiejsze oprogramowanie systemu okienek

Programowanie OO bardziej przystaje do nowego „image" komputerów. Na przestrzeni lat bardzo zmieniły się sposoby rozmawiania komputera z użytkownikiem. Dawniej program zadawał pytania, na które czekał odpowiedzi. Później zastosowano technikę polegającą na tym, że program wypisywał na ekranie kilka ewentualności i pytał o numer tej, którą miał wybrać. Słowem: dawniej rozmowa z użytkownikiem polegała na wypisaniu krótszej lub dłuższej linijki tekstu.

Dziś tak już się nie robi. Przed użytkownikiem na ekranie otwierają się okienka, pojawiają się na ekranie ikony, guziki, które należy wskazać na ekranie myszką – cały ten zestaw, który znasz zapewne z używania systemów, w których nazwie pojawia się słowo *window* (okno). Takie programy nie jest łatwo pisać w technikach tradycyjnych (oczywiście da się).

Do takich programów doskonale nadają się języki obiektowo orientowane. Na ekranie widzimy przecież obiekty klasy ikona, obiekty klasy menu itd. Programowanie obiektowo orientowanie ułatwia zadanie programiście mającemu napisać taki program.

Większy odsiew błędów

Programowanie proceduralne kładło podstawowy nacisk na definiowanie funkcji, które odzwierciedlać miały pewne czynności konieczne do wykonania. Pamiętasz zapewne algorytmy programów rysowane w postaci schematów blokowych.

Dane w tej technice nie są zintegrowane z funkcjami. Dana typu `float` reprezentująca temperaturę mogła zostać wysłana jako argument funkcji

`oblicz_pole_kola(float)` i nie zostałaby wykryta żadna niezgodność mimo, że zachowanie `oblicz_pole_kola` nie jest zachowaniem, stosuje się wobec temperatury.

Przy stosowaniu techniki obiektowo orientowanej może być to od razu wykryte jako błąd. Funkcja nie może być zastosowana do danych (obiektów), na których pracować nie jest uprawniona. Już na etapie kompilacji zostaniemy poinformowani o niedopuszczalnym zastosowaniu funkcji i uchroni nas to przed błędami logicznymi. To dlatego mówi się, że jeśli program w C++ uda się poprawnie skompilować, to jest duża szansa, że od razu będzie działał poprawnie. Przy programowaniu proceduralnym trzeba się o wiele więcej namęczyć przy uruchamianiu programu.

Uogólnianie bez konieczności precyzowania szczegółów

Jeśli w tradycyjnej technice programowania chciałeś zdefiniować kolejkę to musiałeś wyraźnie określić jakie elementy mają w takiej kolejce stać. Czyli, aby uzyskać uogólnienie jakim jest kolejka, najpierw szczegóły („kolejkowicze") musiały być ściśle określone.

W programowaniu OO jest odwrotnie. Możesz zdefiniować uogólnienie nie definiując, ani nawet nie znając szczegółów.

Przykładowo – można napisać definicję klasy kolejka, w której stoją elementy znane, a także elementy, których w momencie pisania programu jeszcze nie znamy. Zostaną wymyślone dopiero wtedy, gdy życie pokaże, że są potrzebne. Nie wywoła to konieczności zmiany tego uogólnienia jakim jest kolejka.

22.3 Obiektowo Orientowane: Projektowanie

Najprzyjemniejszym etapem programowania obiektowo orientowanego jest samo *projektowanie* programu.

Przypomnij sobie jak to było do tej pory. Siadałeś przed monitorem, uruchamiałeś edytor i zaczynałeś pisać program wystukując kolejne instrukcje wprost na klawiaturze. Najpierw zaczynałeś pisać segment główny programu (w języku C jest to funkcja `main`). To, co miałeś zaprogramować, rozbijałeś na proste czynności, z których robiłeś funkcje. To było właściwie kwintesencją programowania – napisanie określonej liczby funkcji wywoływanych w określonym porządku.

Z programowaniem OO tak się nie da. Nie można usiąść przed monitorem i *a vista* pisać program. Trzeba sobie ten program najpierw obmyślić.

„– Eeee" – pomyślałeś sobie zapewne – „miało być wspaniałe narzędzie, a tymczasem już utrudnienia..."

Nie zniechęcaj się. Oczywiście, że możesz od razu zasiąść przed monitorem i pisać, tylko że to, co napiszesz, będzie trochę pokraczne. – Tak, jakbyś chcąc budować dom, poszedł od razu na plac budowy i zaczął murować. To, co zbudujesz, domem mimo wszystko będzie. Jednak będzie pokraczne. Każdy wie, że lepiej najpierw narysować sobie projekt tego domu. Podobnie z programowaniem.

Ten etap obmyślania programu (projektowania) dostarcza naprawdę dobrej zabawy. Tak, jak krawiec artysta cieszy się najbardziej, gdy projektuje. Natomiast wtedy, gdy siada do maszyny i by uszyć – to już nie jest takie podniecające. Po prostu rzemiosło.

Porozmawiamy teraz właśnie o projektowaniu.

Ironia losu polega na tym, że na etapie projektowania programu musimy walczyć z naszymi dotychczasowymi nawykami z tradycyjnych technik programowania. Natomiast ktoś, kto nigdy nie programował – technikę OO uzna za zupełnie oczywistą. Oto dlaczego:

Pisany program zawsze odnosi się jakoś do zjawisk w świecie realnym. Jeśli jest to program do księgowania – odzwierciedla obiekty i metodykę pracy księgowego. Są czeki, faktury, kasa, konta. Z kolei program, który służy do gry w Chińczyka – operuje takimi obiektami jak pionki, plansza, kostka do gry. W programie sterującym promem kosmicznym obiektami są poszczególne podzespoły, którymi trzeba w określony sposób sterować.

W sytuacji, gdy posługujemy się tradycyjnymi metodami programowania — musimy ten wycinek świata realnego zapisać jakoś w komputerze. Rozbija się więc wszystko na luźne liczby, na których później pracuje program. Programowanie wówczas polega na – mówiąc obrazowo – wydawaniu poleceń małpie, którą nauczono dodawać. Mówi się: „a teraz dodaj tę liczbę do tamtej i rezultat umieść tutaj". Aby w ten sposób sterować promem kosmicznym – trzeba się bardzo natrudzić.

Programowanie obiektowo orientowane opiera się na innej idei. Jeśli program zajmuje się jakimiś obiektami i zjawiskami ze świata rzeczywistego, to nie rozbijamy tego wszystkiego na luźne liczby. Budujemy model. W programie tworzymy modele rzeczywistych obiektów. Tak zbudowane obiekty wyposażane są nie tylko w dane, ale także w zachowanie.

Mój przyjaciel Dirk Fischer mówi, że wykonywanie programu OO przypomina mu party, na które zaproszono wiele różnych obiektów. Obiekty te chodzą, rozmawiają ze sobą, załatwiają ze sobą różne sprawy. Każdy obiekt ma swój stan wewnętrzny – polegający na grubości swojego portfela, lub ilości wypitej whisky. Obiekty mają też swoje określone zachowania - ktoś jest kelnerem serwującym drinki, ktoś jest bardzo ważną osobą, ktoś jest gospodarzem. O stanie wewnętrznym obiektów nie można dowiedzieć się wprost (private!), ale można to wywnioskować z rozmowy z takim obiektem. Napisanie programu obiektowo orientowanego polega na zorganizowaniu takiego party.

Podkreślmy jeszcze raz: projektowanie programu OO polega na modelowaniu. W komputerze buduje się model zagadnienia, którego dotyczy program.

Jeżeli chcemy nauczyć się nowego języka programowania – to sprawa nie jest trudna. Zawsze jest jakaś pętla `do`, pętla `for`, instrukcja `if`. Nauka polega tylko na uchwyceniu tych niewielkich różnic w składni powyższych instrukcji.

Jeśli zdarzyło Ci się kiedyś tłumaczyć program z Pascal'a na C – to przyznasz mi chyba rację. Jest to dość łatwe: instrukcje tłumaczy się – że tak powiem –

w skali 1:1. Tu w Pascalu jest tak zapisana pętla `for` – to w języku C zapisuję ją w taki-a-taki sposób.

Przejście od programu w C do obiektowo orientowanego programu w C++, to już nie tłumaczenie 1:1. Pojawia się nowa technika programowania - zatem trzeba zmienić sposób rozumowania, sposób patrzenia na rozwiązanie danego problemu. Na szczęście przy technice OO proponuję Ci zmianę sposobu patrzenia na bardziej naturalny – taki, jakim posługujesz się na co dzień w życiu. Inaczej mówiąc program OO nie jest w stosunku 1:1 do programu tradycyjnego. On jest w stosunku 1:1 do rzeczywistości.

22.4 Praktyczne wskazówki dotyczące projektowania programu techniką OO

Pracę nad programem można podzielić na następujące fazy:

- Rozpoznanie problemu
- Projektowanie programu
- Implementacja (kodowanie)
- Testowanie

Najpierw należy zapoznać się z zagadnieniem i zrozumieć je – to faza rozpoznania. Następnie, wiedząc co mamy zrobić, projektujemy program. Kolejna faza to implementacja, czyli moment właściwego kodowania programu (np. w języku C++). Potem następuje testowanie. Testowanie – jak powszechnie wiadomo – nie wieńczy dzieła. Najczęściej okazuje się, że trzeba wrócić i coś przeprojektować lub nawet lepiej zrozumieć, bo nastąpiło nieporozumienie, albo zamawiający program zażądał dodatkowych modyfikacji. Praca nad programem odbywa się metodą kolejnych przybliżeń.

Prześledzimy teraz dokładniej kolejne fazy pracy nad programem.

22.4.1 Rekonesans – czyli rozpoznanie zagadnienia

Dowcipni twierdzą, że praca nad programem zaczyna się wtedy, gdy szef lub klient przychodzi do Ciebie i – machając rękami oraz bardzo dużo i niezrozumiale mówiąc – wysuwa serię dziwacznych żądań. W tym momencie przystąpić trzeba do rozpoznania zagadnienia.

Jest to sztuka sama w sobie, wkraczająca niemal w psychologię – nie możemy tutaj dłużej się nad tym zatrzymywać. W skrócie powiem tylko, że jeśli klient jest nerwowy, to sadzamy go w wygodnym fotelu i nie dopuszczamy, by z niego wstawał. Następnie zadajemy mu proste pytania, które zmuszą go do udzielania prostych odpowiedzi. Metodą takich pytań po dłuższym czasie dochodzimy do tego, co program ma robić i w jaki sposób.

A teraz uwaga: jeśli klient powie, że program ma spełniać takie-a-takie warunki, ale liczyć się należy, że w przyszłości będzie musiał spełniać inne, nieznane jeszcze dzisiaj – to jest to sygnał, że program musisz napisać w języku obiektowo orientowanym (np. w C++).

Jeśli tak nie powie, to oczywiście też możesz pisać techniką OO bo:

- 1) lubisz ten nowoczesny styl programowania,
- 2) znasz klientów i wiesz, że za miesiąc zmieniają zdanie i już wtedy będą potrzebne modyfikacje.

To wszystko, co opowiedział klient o jakimś swoim zagadnieniu, które należy oprogramować – ten kawałek rzeczywistości, którym zająć się ma program – to nazywać będziemy **systemem.** Księgowość danej firmy to jakiś system. Gra w Chińczyka to także jakiś system.

Chodzi więc o rozpoznanie systemu. Takie rozpoznanie już kiedyś zapewne w życiu przeprowadzałeś – wtedy, gdy ktoś uczył Cię w tego Chińczyka grać. Ktoś Ci to zawile tłumaczył, Ty zadawałeś proste pytania po to, by jego skomplikowaną wypowiedź uprościć. Co jakiś czas upewniałeś się czy dobrze zrozumiałeś dany fragment. Dodatkowo jeszcze przyjrzałeś się jak grają w tego Chińczyka inni. Suma tych działań stanowiła dla Ciebie rozpoznanie tego systemu.

Nie inaczej jest w wypadku rozpoznania poważnego systemu – księgowości, sterowania samolotem, czy systemu kontrolującego aparaturę elektroniczną w eksperymencie fizycznym.

Dopiero, gdy rozpoznamy system – możemy przystąpić do projektowania. Możesz być jednak pewien, że jeszcze powrócisz do tego etapu. Wcześniej czy później okaże się, że czegoś dokładnie nie zrozumiałeś. Nic w tym złego. To jest wkalkulowane w metodę.

22.4.2 Faza projektowania

Po analizie systemu (zagadnienia) wiemy już *co* mamy zrobić. Przystępujemy więc do projektowania programu, czyli do ustalenia *jak* to zrobimy. Ogólnie biorąc chodzi teraz o znalezienie obiektów, za pomocą których modeluje się system.

Istnieje wiele metod projektowania programu OO, tak jak może istnieć wiele dróg prowadzących do tego samego celu. Osobiście posługuję się połączeniem dwóch metod. Ich nazwy brzmią trochę skomplikowanie: „Analiza Zachowań Obiektów" oraz „ Projektowanie na Zasadzie Obserwacji Obowiązków Systemu".

Nazwa „Analiza Zachowań Obiektów " – jest raczej jasna. Najważniejszym słowem w niej jest: „...zachowań..."

Z kolei nazwa „Projektowanie na Zasadzie Obserwacji Obowiązków Systemu" mówi, że przy projektowaniu zwracamy uwagę na obowiązki, które spełnia system. Obowiązki w tym kontekście, to wszystkie usługi, które wykonują obiekty wobec innych obiektów.

Przykładowo: obowiązkiem radaru jest narysować punkt na ekranie. Obowiązkiem kostki do gry jest podać liczbę przypadkową z przedziału 1-6.

Jak w życiu – obowiązek obiekt może spełnić sam, a może go zlecić obiektowi współpracującemu. Kostka do gry sama spełnia swój obowiązek. Obowiązek automatycznego podlewania plantacji pomarańczy spełniany jest przez obiekt zawór wodny i obiekt czasomierz.

Projektowanie według obowiązków systemu polega na obserwacji tych obowiązków i rozdzieleniu ich między obiekty: **kto ma robić co**. (To samo w terminach C++: która klasa ma mieć jakie funkcje składowe).

Projektowanie takie składa się z kilku etapów.

Etapy projektowania programu obiektowo orientowanego:

- 1) Identyfikacja zachowań systemu,
- 2) Identyfikacja obiektów występujących w systemie,
- 3) Klasyfikacja obiektów:
  - pod względem dziedziczenia (hierarchie),
  - pod względem mieszczenia w sobie obiektów składowych,
- 4) Określenie wzajemnych zależności klas obiektów,
- 5) Składanie modelu – Określanie sekwencji działań obiektów,

Etap 1 to jakby usystematyzowanie wiedzy o problemie. Etap 2 to jakby etap poszukiwania. Etapy 3,4,5 to analiza tego, co znaleźliśmy.

W rezultacie takiego projektowania powstaje model, który staje się podstawą przy kodowaniu programu. Wymyślonych jest kilka klas, określone ich cechy i wymagane funkcje składowe (zachowania). Można przystąpić do fazy implementacji czyli kodowania programu. Kodowaniu poświęcone były wszystkie poprzednie rozdziały tej książki.

Istotą tego rozdziału, który właśnie czytasz, jest pokazanie jak w praktyce zrealizować powyższe pięć punktów fazy projektowania. Nie martw się jeśli początkowo czegoś nie zrozumiesz – po teoretycznym przedstawieniu zasad pokażemy przykład projektowania konkretnego programu.

22.4.3 Etap 1: Identyfikacja zachowań systemu

Całą swoją wiedzę o systemie zapisujesz w formie scenariusza. Chronologia zdarzeń nie jest w tym miejscu konieczna, jednak jeśli uda Ci się ją zachować, wówczas oszczędzisz sobie pracy przy etapie 5 (składanie modelu).

Sporządzenie takiego scenariusza powinno wyglądać tak, że kartkę papieru dzielisz na kolumny, którym dajesz tytuły:

Kto – Działanie – Kogo – Rezultat

Następnie chwilkę się zastanawiasz i zaczynasz wypełniać te kolumny. Powstaje lista zachowań systemu. Zachowań – bo ważna tu jest kolumna druga i czwarta. Oto jak może zaczynać się taki „scenariusz":

Kto	Działanie	Kogo	Rezultat
Gracz	rzuca	kostką	Liczba przypadkowa z zakresu 1-6
.....	.....	.....	.....

22.4.4 Etap 2: Identyfikacja obiektów (klas obiektów)

Poszukiwanie obiektów

Próbujemy odnaleźć obiekty działające w systemie, który przyszło nam oprogramować. Cały wysiłek skupiony jest tu na to, by znaleźć obiekty jak najnaturalniej odwzorowujące rzeczywiste zachowania systemu. Jak się to robi?

Sprawa nie jest wcale trudna. Wystarczy spojrzeć na sporządzony przed chwilą scenariusz. W rubrykach: KTO, KOGO mamy właśnie nazwy klas tych obiektów.

Uwaga:

klasy, które projektujemy mogą być nie tylko reprezentacją istniejących namacalnie obiektów. Klasą może być też okienko menu na ekranie monitora – a także szerzej – cały sposób rozmowy programu z użytkownikiem.

Określenie jakie obiekty mają obowiązki, współpracowników oraz widoczne własności

W tym miejscu posłużymy się narzędziem zwanym kartami modelującymi[†]. Karty te pomogą nam nauczyć się identyfikować obiekty, ich zachowania i współpracowników. Sporządza się je na luźnych kartkach po to, by można było nimi łatwo manipulować – rozkładać na biurku czy stole – a niektóre po prostu wyrzucać jeśli z nich rezygnujemy. Jak zrobić taką kartę? Na górze pisze się nazwę klasy obiektów, które reprezentuje. Poniżej, po lewej stronie spisuje się obowiązki tej klasy, natomiast po prawej współpracowników wymaganych dla spełnienia danego obowiązku. Na dole rubryka na widoczne własności obiektów tej klasy.

Klasa : *(nazwa klasy)*	
Obowiązki:	**Współpracownicy:**
.....	.....
Widoczne własności:	

Zauważ jeszcze jedno: na tym etapie jedynie identyfikujemy obiekty i ich zależności natomiast szczegóły ich implementacji nie są tu wcale ważne. Tym zajmiemy się dopiero na samym końcu projektowania. Dzięki temu nic nie staje się przesądzone od razu i jakiekolwiek zmiany wymagane później w implementacji nie ruinują nam wstępnych etapów projektu.

†) Zaproponowane przez K.Becka i H. Cunninghama – Laboratory for Teaching Object-Oriented thinking – in Proceedings of OOPSLA'89

Zapamiętaj to jako zasadę:

> Tak dalece, jak to tylko możliwe, odsuwaj na później szczegóły dotyczące struktury wewnętrznej klasy.

Wypisane własności nie są zwykle ostateczne. Zdarza się, że niektóre obowiązki przenosimy z jednych klas obiektów na drugie. Przypominam - projektowanie odbywa się metodą kolejnych przybliżeń.

Sporządzone w ten sposób karty modelujące posłużą nam teraz do określenia zachowań i własności konkretnych klas. Inaczej mówiąc zastanowimy się jakie widzialne własności (cechy) musi mieć każda klasa. Te widzialne własności mogą stać się kiedyś danymi składowymi klasy, natomiast obowiązki – funkcjami składowymi.

Oto przykład wypełnienia takiej karty modelującej dla hipotetycznej klasy: pralka automatyczna

Klasa : PRALKA AUTOMATYCZNA	
Obowiązki:	**Współpracownicy:**
wirowanie nabieranie wody odpompowanie wody . . .	silnik zawór pompa wodna . . .
Widoczne własności: -pojemność bębna -cena, waga, -bieżący program -bieżąca temperatura . . .	

W rezultacie tego etapu mamy dobrze zidentyfikowane klasy obiektów występujących w naszym systemie. Zidentyfikowaliśmy też ich obowiązki i ich współpracowników.

22.4.5 Etap 3: Usystematyzowanie klas obiektów

Pora teraz przystąpić do analizy tego, co znaleźliśmy na etapie poszukiwania. Temu, co zidentyfikowaliśmy – przyjrzymy się teraz krytycznie. Tutaj chodzi konkretnie o to, by zastanowić się czy klasy nie są powiązane (związkami dziedziczenia) w jakieś hierarchie, ewentualnie czy obiekty jednej klasy nie mieszczą w sobie obiektów innej klasy. Etap ten jest bardzo ważny, gdyż tutaj zdecyduje się czy mamy szanse na *reusability* - wtórne użycie jakichś klas, funkcji, ogólniej: części programu.

Etap 3 a) Ustalenie hierarchii

Dobrze zaprojektowane hierarchie dziedziczenia są istotą dobrego programowania OO. Przy dobrym zdefiniowaniu klas abstrakcyjnych możemy korzystać z nich wielokrotnie w różnych klasach programu i w ten sposób zaoszczędzimy sobie dużo pracy. Dlatego na tym całym etapie projektowania należy mieć wyostrzoną uwagę na wszelkie podobieństwa zachowań między klasami.

Ewentualne powiązanie klas w związki dziedziczenia sprawdza się bardzo łatwo. Wystarczy wypowiedzieć głośno zdanie

> Obiekt klasy A jest szczególnym rodzajem obiektu klasy B

Jeśli zabrzmi to nonsensownie to znaczy, że nie ma tu związku dziedziczenia.

Oto przykładowe zdania:

Pojazd jest szczególnym rodzajem czasomierza
- bzdura -

Pojazd jest szczególnym rodzajem samochodu
- bzdura -

Ale zdanie:

Samochód jest szczególnym rodzajem pojazdu
- brzmi sensownie

Jeśli zdanie brzmi sensownie - to znaczy, że klasa A (samochód) jest klasą pochodną od klasy B (pojazd).

Jeśli klasy, które znaleźliśmy w poprzednim etapie, nie są powiązane związkami dziedziczenia, to nie świadczy jeszcze o tym, że żadnych hierarchii nie będzie. Możemy spróbować poszukać klas abstrakcyjnych - wspólnych dla dwóch konkretnych klas. Robi się to rozważając jakie wspólne obowiązki mają dwie klasy.

Zauważone podobieństwa powinny w tym właśnie momencie owocować klasami abstrakcyjnymi: jedno wspólne zachowanie, które mają dwie klasy wystarczy, by zbudować klasę abstrakcyjną. Starajmy się wyszukać tak dużo klas abstrakcyjnych, jak to tylko możliwe. Czy wytrzymają one próbę czasu - to już sprawa następnych etapów.

(Jest rzeczą bardzo ważną, by znaleźć te klasy abstrakcyjne jeszcze przed rozdziałem obowiązków, który robimy w następnych etapach).

Etap 3 b) Mieszczenie (zawieranie) w sobie obiektów innych klas

Związek klas polegający na mieszczeniu w sobie obiektu innej klasy łatwo sprawdzamy wygłaszając stwierdzenie:

> Obiekt klasy A składa się między innymi z obiektu klasy B

Jeśli zdanie zabrzmi sensownie, to obiekt klasy B jest składnikiem klasy A.

Przykładowo:

Samochód składa się między innymi z krowy
(bzdura)

Samochód składa się między innymi z kierownicy
(prawda!)

Krowa składa się między innymi z dwóch oczu
(prawda)

22.4.6 Etap 4: Określenie wzajemnych zależności klas

Na tym etapie musimy zdecydować o wzajemnych relacjach klas – czyli o tym, która jest ważniejsza od której, która wydaje polecenia, a która polecenia te spełnia, albo które klasy są partnerami – prosząc się nawzajem o przysługi.

Określanie takich relacji dokonuje się dwojako. Po pierwsze sporządza się spis relacji łączących obiekty, a po drugie rysuje się tzw. graf współpracy obiektów.

Spis relacji

sporządza się w ten sposób, że mając przed sobą scenariusz, a także karty modelujące wypisuje się poszczególne relacje. Powinny one być zebrane w grupy. To znaczy, że najpierw wypisujemy wszystkie relacje takie, gdzie jakiś obiekt **zmusza** inny obiekt do wykonania jakiegoś działania, potem na przykład relację zwykłego **powiadamiania się** o czymś (komunikowanie się). Inaczej – kto żąda – a kto tylko realizuje żądania.

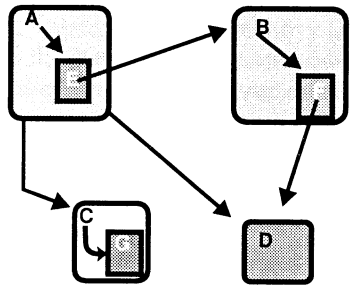
Oto przykład – dla hipotetycznego obiektu klasy: programator pralki automatycznej

Obiekt	Zależność	Kogo
Programator	zmusza do otwarcia zmusza do zamknięcia zmusza do wirowania ... dowiaduje się o bieżący poziom wody dowiaduje się czy drzwiczki są zamknięte ...	Zawór wodny Zawór wodny Silnik ... hydrostat czujnik drzwi ...
Następny obiekt	...	...

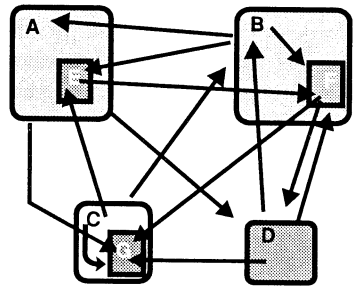
Graf współpracy

Graf współpracy ma tę zaletę, że pokazuje jasno ścieżki komunikacji między klasami. Graf taki sporządzany zostaje na podstawie ustaleń z dotychczasowych etapów projektowania. Rysujemy go w ten sposób, że prostokątami oznaczamy poszczególne klasy (obiekty), a strzałkami oznaczamy zależności. Jest wiele szkół rysowania takich grafów. Ja osobiście stosuję najprostszy sposób. Strzałkami oznaczam kto komu rozkazuje. Groty z obu końców strzałki oznaczają, że dwie klasy nawzajem proszą się o przysługi.

Narysowanie grafu to jeszcze nie wszystko. Teraz trzeba mu się krytycznie przyjrzeć. To bardzo ważne zobaczyć nasze obiekty narysowane w ten sposób, gdyż za jednym rzutem oka widzimy miejsca niepotrzebnego skomplikowania lub sytuacje, gdy naruszona jest enkapsulacja.



Graf dobry



Graf zły

Jeśli patrząc na graf zauważysz, że jakaś klasa współpracuje z więcej niż trzema innymi klasami – oznacza to zwykle, że coś strasznie pokomplikowaliśmy. Tak, tak – i tutaj możesz wyprodukować „kod a la spaghetti". Jeśli z Twojego grafu wynika, że wszyscy współpracują z wszystkimi – to trudno Ci będzie opanować taką hołotę. Zacznij od minimalizacji liczby współpracowników. Zwróć też uwagę czy niektóre klasy nie działają na jakieś obiekty tkwiące we wnętrzu innych klas. To byłoby pogwałceniem zasady enkapsulacji. Po to obiekt tkwi w środku innego obiektu, aby bezpośrednio do niego się nie zwracać (tylko przez jego szefa).

Innym rodzajem uproszczenia jest zminimalizowanie typów obowiązków spełnianych przez klasę (czyli tzw. kontraktów). Niech klasa zajmuje się tylko jednym zadaniem. Niech np. pionek do gry zajmuje się tylko poruszaniem się po szachownicy, a nie oblicza dodatkowo sinusa i logarytmu. Jeśli ten typ obowiązków (sin, log) musi być spełniany, to rozważmy budowę dodatkowej klasy.

Patrząc na graf współpracy możesz łatwiej zorientować się co i jak można uprościć. Zwracam uwagę: do tego najlepiej nadaje się graf, bo tu widać wszystko jak na dłoni. Jeśli więc patrząc na niego wymyślisz jakąś modyfikację, to wróć do jednego z poprzednich etapów i popraw.

Podsystemy

Jeszcze jedną zaletą grafu jest to, że widać na nim także grupy klas, które nie mają ze sobą zupełnie nic wspólnego. Wówczas możemy spróbować zgrupować klasy w tak zwane **podsystemy**, czyli grupy klas zajmujących się spełnianiem jednej, bardzo ogólnie pojętej roli.

Na przykład – jeśli program na grę w Chińczyka wzbogacimy o grupę klas odpowiedzialnych za granie chińskiej muzyki, to ta nowa grupa klas nie ma nic wspólnego z grupą do której należą pionki, kostka, itd.

Oto inny przykład: program obsługujący eksperyment fizyczny, w którym to programie jest kilka klas realizujących rozmowę z użytkownikiem za pomocą eleganckich menu. Ta grupa klas to właśnie podsystem. Jeśli w tym samym programie jest inna grupa klas odpowiedzialnych za sterowanie dopływu ciekłego azotu do detektorów germanowych – to klasy te tworzą kolejny podsystem.

Na grafie takie grupy klas odgraniczamy czerwoną linią – co daje nam pojęcie, które klasy należą do danego podsystemu, a które do sąsiedniego. Przy wykreślaniu tej linii jako kryterium przyjmujemy zasadę: klasa należy do danego podsystemu wtedy, gdy spełnia zadania jedynie na rzecz tego podsystemu.

Zapytasz pewnie: „–Dlaczego to takie ważne – przecież w C++ istnieje coś takiego jak klasa, ale nie istnieje nic w stylu *'podsystem klas'*?"

Rzeczywiście, jednak praca nad programem i projektem bardzo upraszcza się jeśli dokonaliśmy podziału na podsystemy. Najpierw zajmiemy się zaprojektowaniem – i nawet implementacją – części odpowiedzialnej za ciekły azot i nie pamiętamy o istnieniu reszty. Jeśli pracuje się w zespole, to jeden z programistów opracowuje ten podsystem, a drugi inny. Ich klasy nawzajem się nie kontaktują, więc taka rozłączna praca jest możliwa. Aby próbnie uruchomić jeden podsystem – drugi może nawet jeszcze nie być zaczęty.

22.4.7 Etap 5: Składanie modelu. Określanie sekwencji działań obiektów i cykli życiowych

Czas na to, by skonstruować model systemu będącego przedmiotem naszego programu.

Jeśli podzieliliśmy zagadnienie na kilka podsystemów, to teraz czas, by modelować jeden z nich. Gdy to zrobimy, to weźmiemy na warsztat inny podsystem. Ta metoda bardzo ułatwi nam pracę nad modelowaniem całości systemu.

Jak postępuje się w przypadku składania modelu?

Bardzo prosto: bierze się do ręki scenariusz, który powstał w etapie 1. Jeśli udało Ci się go wtedy zrobić chronologicznie — to jest dużo pracy zrobionej. Jeśli nie — to czas zrobić to teraz.

To jednak nie wszystko. Teraz musimy ustalić sekwencje działań obiektów każdej z klas. To znaczy ustalamy kto inicjuje jaką akcję, co jest robione potem.

Powstają wówczas takie sformułowania:

> *Jak tylko obiekt klasy A otrzyma sygnał, by zrobił to - wydaje on następujące polecenie obiektowi klasy B i... (np. czeka na rezultat).*

Obiekty mają swoją indywidualność więc nie zawsze musi być to takie bezmyślne. Np. po otrzymaniu sygnału obiekt A może dowiedzieć się o parę spraw, po czym zależnie od oceny sytuacji może wybrać jeden z wariantów postępowania. Jak w życiu. Niewolnictwo i feudalizm dawno już się skończyły.

Następnie opisujemy cykle życiowe poszczególnych obiektów.

> *Np. w grze w Chińczyka najpierw obiekt pionek jest w stanie „w drodze" potem, gdy zdarza się coś szczególnego, przechodzi w stan „w Pekinie" itd.*

Rezultat tego etapu przyda się przy definiowaniu ciał poszczególnych funkcji składowych danej klasy.

Zwróć uwagę, że etap ten nie produkuje niczego nadzwyczajnego. Jest to raczej usystematyzowanie naszej wiedzy o obiektach. Praca tutaj przypomina trochę reżyserowanie spektaklu: najpierw wchodzisz ty, i robisz *to...*

Albo: jeśli tylko on zwróci się do ciebie z takim żądaniem, wtedy zrób *tamto*.

Na tym zakończył się proces projektowania programu obiektowo orientowanego. Teraz można przystąpić do implementacji czyli zamieniania tego na instrukcje w języku C++.

22.5 Faza implementacji

Przystępując do implementacji staramy się napisać definicje klas. Nie jest to trudne, gdyż robimy to w oparciu o określone wcześniej:

- widoczne własności (cechy) klasy – to będą dane składowe,
- zachowania klasy – to będą funkcje składowe.

Funkcje składowe piszemy na razie jedynie w postaci deklaracji. Ich definiowaniem zajmiemy się na samym końcu.

Kiedy wszystkie klasy mamy zdefiniowane – można przystąpić do pisania definicji funkcji składowych. Tutaj dopiero przydaje się tradycyjna technika programowania: rozbijanie wymaganej czynności na serię mniejszych. Niektóre z nich mogą okazać się często używane, więc opłaci się z nich zrobić prywatną funkcje składową klasy. Jak powiedziałem – technikę proceduralną stosuje się dopiero do działań wewnątrz klasy. Z zewnątrz obiekt musi być widoczny jako obiekt, natomiast jaką sobie techniką zrealizujemy wnętrze funkcji składowej — to nasza sprawa. Wewnętrzna struktura obiektu nie jest już (zwykle) obiektowa. Jeśli dane składowe są liczbami, to wymagają tradycyjnego podejścia. Oczywiście – jeśli daną składową jest obiekt jakiejś innej klasy, to wtedy w stosunku do niego używa się techniki OO.

Mówi się, że funkcja składowa powinna być tak duża, by mieściła się w całości na ekranie monitora. Jeśli jest większa, to znaczy, że pewnie jest zbyt skomplikowana i powinna zostać uproszczona – zwykle przez wyodrębnienie z niej jakichś mniejszych funkcji (robi się je zwykle prywatnymi funkcjami składowymi klasy).

Na tym kończy się cykl moich rad, albowiem o tym, jak postępować przy pisaniu konkretnych funkcji – traktowały wszystkie poprzednie rozdziały tej książki.

Zobaczymy teraz, jak opisany proces projektowania przebiega w praktyce. Tytułem przykładu według powyższych zasad zaprojektujemy konkretny program.

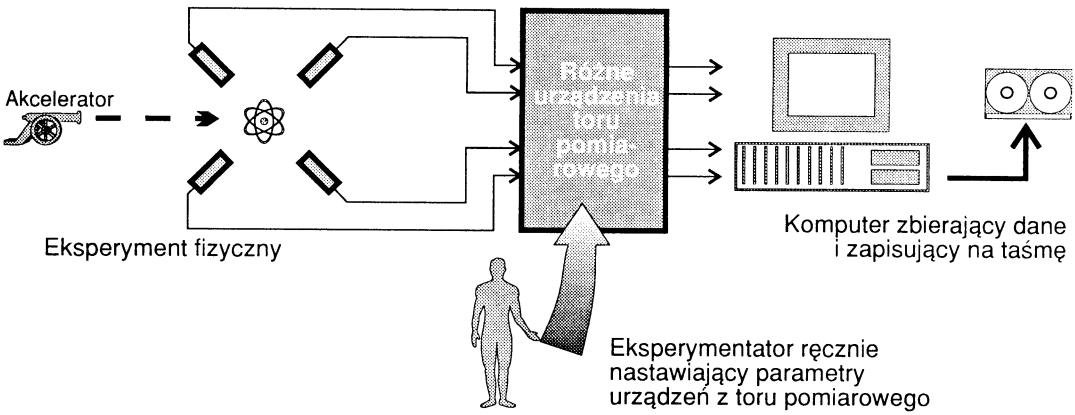
22.6 Przykład projektowania

Nie jest to przykład wydumany – taki właśnie program musiałem kiedyś napisać. Tu przedstawię go w wersji bardzo uproszczonej – chodzi nam bowiem o to, by poznać samą istotę projektowania.

Nie przejmuj się jeśli „fizyczno – elektroniczna" strona tego przykładu jest dla Ciebie nieinteresująca lub niezrozumiała. Jest to przecież dla nas tylko pretekst do napisania programu.

22.7 Faza: Rozpoznanie naszego zagadnienia

W tej fazie wyjaśnimy sobie co program ma robić. Otóż wyobraź sobie, że masz duże urządzenie pomiarowe służące do eksperymentów odkrywających strukturę jądra atomowego. Urządzenie, o którym myślę, rzeczywiście istnieje i nazywa się OSIRIS. W uproszczeniu mówiąc jest to 12 czujników promieniowania†) ustawionych pod różnymi kątami w stosunku do miejsca skąd promieniowanie pochodzi. To wszystko! Z tych 12 czujników wychodzą sygnały elektryczne informujące o zarejestrowaniu kwantu promieniowania – tak, jakby czujniki wykryły błysk światła. Sygnał elektryczny – niosący informację o tym – z czujnika przechodzi przez cały tzw. tor pomiarowy i wędruje do komputera, którego rolą jest tę informację zapisać na taśmie magnetycznej. (Schematycznie pokazuje to rysunek poniżej).



Komputer, który te informacje zbiera, już istnieje i jest poprawnie zaprogramowany – zatem dane rzeczywiście zapisywane są na taśmach.

Zapytasz pewnie: „Skoro tak, to gdzie tu jest nasza rola ?"

Otóż sam tor pomiarowy to dziesiątki urządzeń elektronicznych, z których każde ma mnóstwo przełączników i gałek. Aby móc przeprowadzić eksperyment trzeba te wszystkie urządzenia nastroić. To może trwać nawet parę dni. Potem przeprowadza się właściwy eksperyment (trwający np. 7 dni). Kiedy po jakimś czasie chce się wykonać następny eksperyment – znowu trzeba wszystko ustawiać od nowa.

†) fizycy mówią: detektorów promieniowania gamma

Pomyśl: – jakby to było cudowne, gdyby móc nastawy jakoś zapamiętywać i w parę sekund odtwarzać te, które używaliśmy w trakcie eksperymentu w zeszłym miesiącu, albo kiedy indziej odtworzyć te z zeszłego roku. To jest właśnie nasza rola.

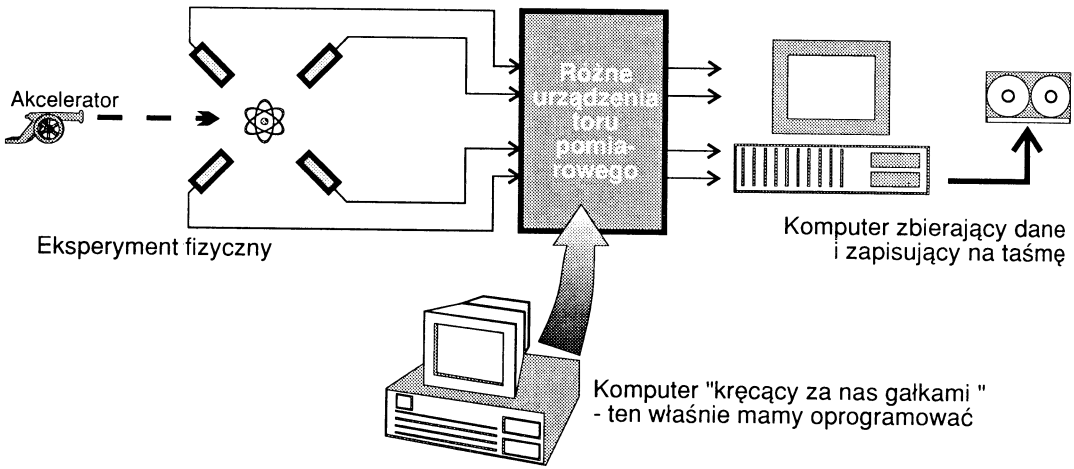
Nie jest to takie trudne w realizacji

Otóż większość urządzeń, o które chodzi, przystosowana jest do regulowania swych nastaw za pomocą komputera.

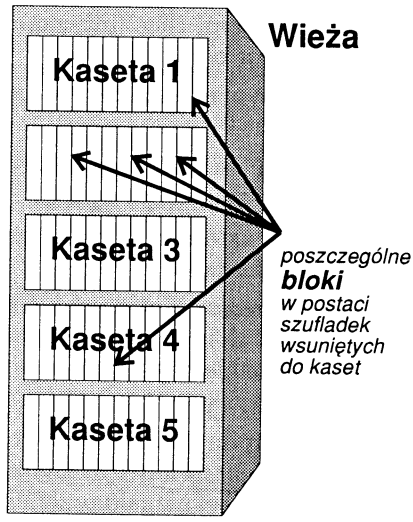
Nie mają one nawet gałek, bo gdyby trzeba regulować je za pomocą gałek i przełączników – to gałek tych musiałyby być setki). Są to tzw. urządzenia serii ECL (Electronic Controlled Logic – logika sterowana elektronicznie).

Nasi specjaliści elektronicy postarali się o komputer i dołączyli go do toru pomiarowego. Teraz my mamy napisać program.

Sytuacja wygląda mniej więcej tak, jak na kolejnym rysunku.



Skoro nasz komputer ma sterować nastawami konkretnych urządzeń w torze pomiarowym – przyjrzyjmy się co to za urządzenia. Wszystkie te urządzenia są czymś w rodzaju pionowych szufladek – dlatego nazywamy ja „blokami". Szufladkę można wsunąć do kasety, w której jest miejsce na ok. 20 takich bloków. Słowem w kasecie takiej może obok siebie pracować 20 urządzeń. Dwadzieścia – to mało, dlatego ustawia się więcej kaset - jedna na drugiej. Stanowi to jakby „wieżę" – składającą się z pięciu kaset. (Jak na kolejnym rysunku)



Nasi elektronicy sprawili, że możemy z komputera zwrócić się do bloku stojącego na przykład w kasecie nr 2 na miejscu nr 19. Jest do tych celów specjalna funkcja – będziemy ją tylko wywoływali.

Jeśli nic nie zrozumiałeś z dotychczasowych wyjaśnień, nie przejmuj się. W skrócie chodzi o to, by:

- 1) z komputera móc regulować urządzenia stojące w odpowiednich szufladkach wieży,
- 2) móc zapisać nastawy wszystkich tych urządzeń na dysku,
- 3) móc odtwarzać te nastawy na podstawie danych zapisanych wcześniej na dysku.

Skoro rozpoznanie zagadnienia polega na zadawaniu zleceniodawcy pytań — zapytajmy naszego kolegę:

„-Co chcesz by program robił?"

Kolega odpowiada tak:

- ❖ – Gdy uruchomię program chciałbym na ekranie zobaczyć tę wieżę. W całości, albo lepiej w postaci wybranej jednej z pięciu kaset.
- ❖ – Gdy uruchamiam program po raz pierwszy, to program nie zna jeszcze konfiguracji mojej rzeczywistej wieży. Chciałbym więc mieć możliwość powiedzenia programowi, w którym miejscu wieży włożony jest jaki blok. Niestety te urządzenia są tak zbudowane, że komputer nie może ich sam rozpoznawać.
- ❖ – Kiedy już program zna umiejscowienie wszystkich bloków, chciałbym mieć możliwość wybrania jednego konkretnego bloku, po to, by móc z nim samym rozmawiać. Wtedy właśnie dokonam regulacji bloku. Za-

miast kręcić gałkami chcę (myszką) na ekranie komputera regulować odpowiednie wartości.

- ❖ – Potem chciałbym mieć możliwość zrobienia tego samego z dowolnym innym blokiem.
- ❖ – Gdy już będę zadowolony ze wszystkich nastaw, chciałbym mieć możliwość zapisania ich na dysku.
- ❖ – Oczywiście musi również istnieć możliwość odczytania tych zapisanych na dysku nastaw.
- ❖ – Ponieważ do różnych eksperymentów potrzebuję czasem innych zestawów bloków, dlatego chciałbym mieć możliwość usunięcia bloku, który mi obecnie nie odpowiada.
- ❖ – Ponieważ nasz OSIRIS ciągle się rozwija, dlatego należy się liczyć z tym, że dokupimy (lub skonstruujemy) nowe bloki ECL. Program powinien pozwalać na łatwe modyfikacje. Tak, by wprowadzenie nowego typu bloku nie wymagało przerabiania całości programu.

Jakimi urządzeniami mamy regulować? No cóż, tym akurat nie chcę Ci, drogi Czytelniku, specjalnie zawracać głowy. Ważne jest, że spełniają one dla fizyków jakieś istotne role. Urządzeń tych może być dużo, dla przykładu:

- a) blok wysokiego napięcia,
- b) młynek do kawy,
- c) ekspres do kawy,
- d) jednostka logiczna,
- e) matryca opóźnień,
- f) przelicznik.

Oto jakie znaczenie mają dla fizyka poszczególne urządzenia:

ad a) Blok wysokiego napięcia.

Czujniki (detektory) promieniowania są zwykle zasilane wysokim napięciem. Wartość tego napięcia jest regulowana w zależności od detektora. Nasz blok potrafi sterować 256-ścioma źródłami takiego napięcia.

ad b) Młynek do kawy
ad c) Ekspres do kawy

Eksperyment na wiązce z akceleratora trwa zwykle 4-7 dni — bez przerwy: dzień i noc. Nietrudno więc zrozumieć jak wielką rolę odgrywają dla fizyka powyższe dwa urządzenia.

To oczywiście żart. Niezbyt jednak płochy, gdyż ma dwa cele:

- dać szansę zrozumienia czytelnikom, którzy chcą nadal czytać ten rozdział, a fizyka jądrowa ich zupełnie nie interesuje,
- unaocznić, że urządzenia znajdujące się w wieży nie muszą mieć nic wspólnego ze sobą. Konkretniej: nie muszą koniecznie ze sobą współpracować. Młynek do kawy nie współpracuje z blokiem wysokiego napięcia.

ad d) Jednostka logiczna

To po prostu dwa przełączniki. To, w jakiej mają być pozycji, sterowane będzie właśnie komputerem.

ad e) Matryca opóźnień

W tym bloku jest 16 układów potrafiących opóźnić 16 oddzielnych sygnałów — każdy o wybrany czas. Fizycy bardzo lubią takie sztuczki.

ad f) Przelicznik (tzw. scaler)

To jakby zespół 32 liczników. Na nasz sygnał otwiera się do nich wejście i przez określony czas liczniki liczą przychodzące do nich skądś impulsy. Po pewnym czasie wejście impulsów zamykamy i patrzymy ile każdy z liczników tych impulsów zliczył.

Tyle rozmowa z kolegą. Rozpoznaliśmy zagadnienie. Teraz możemy przystąpić do projektowania. Uwaga kolegi – ta przewidująca ciągłe modyfikacje – od razu sugeruje, że najlepiej zrobimy posługując się techniką obiektowo orientowaną.

22.8 Faza: Projektowanie

Jak wiemy, faza ta składa się z pięciu etapów. Oto one:

22.8.1 Etap 1 – Identyfikacja zachowań naszego systemu

To oczywiście ten scenariusz.

Umówmy się, że to miejsce w kasecie, w które można wsunąć blok nazywać będziemy tu *szufladką*. Bardziej fachowa nazwa to: *„stanowisko"*, lub z angielska: *„slot"*

Oto scenariusz w postaci tabeli:

Kto	Działanie	Kogo	Rezultat
Kaseta	rysuje	się	Przygotowuje to program do pracy
Menu	rysuje	się	Gotowość przyjmowania poleceń
Użytkownik	określa numer	kasety	Do tej kasety będzie się zaraz zwracał
Użytkownik	określa numer	szufladki	Z tą szufladką chce pracować
Użytkownik	w tak wybrane miejsce wstawia	blok	Wieża zapełnia się konkretnymi blokami

Menu	oferuje zestaw	bloków	Umożliwia to nam wybór konkretnego urządzenia
Użytkownik	reguluje	blok	Tak odbywa się „kręcenie gałkami" konkretnego urządzenia w torze pomiarowym
Użytkownik	żąda zapisu wszystkich nastaw	wieży	Nastawy wszystkich bloków w torze pomiarowym zostają zapamiętane na dysku
Użytkownik	żąda odczytania nastaw	wieży	Nastawy wszystkich bloków w torze pomiarowym zostają odczytane z dysku i poszczególne bloki są tak ustawiane.
Użytkownik	usuwa z szfladki	blok	Konkretny blok jest usuwany z szufladki i staje się ona pusta.

W scenariuszu pominąłem wszystkie sytuacje, gdy użytkownik rozmawia z konkretnym blokiem – blokiem wysokiego napięcia, czy z młynkiem do kawy. To po prostu dla oszczędności miejsca w tej książce.

22.8.2 Etap 2 – Identyfikacja klas obiektów, z którymi mamy do czynienia

Jak pamiętamy, trzeba się przyglądnąć tabeli. Kolumnie pierwszej i trzeciej. Wszystko jest w zasadzie jasne. Jedynie chciałbym się wytłumaczyć z klasy „użytkownik". Oczywiście nie chodzi tu o samego człowieka. Człowiek zleca zrobienie czegoś programowi za pomocą wybrania jakiejś akcji z menu. Zatem użytkownik jest reprezentowany przez klasę „menu".

Szybki rzut oka na kolumny 1 i 3 powyższego scenariusza i widzimy, że mamy tu do czynienia z następującymi klasami obiektów:

> wieża, kaseta, menu (a przez nie: użytkownik), szufladka, blok wysokiego napięcia, młynek do kawy, ekspres do kawy, jednostka logiczna, matryca opóźnień, przelicznik.

Skoro znaleźliśmy już klasy obiektów występujących w naszym systemie – to sporządźmy dla nich karty modelujące. Wybacz, że nie zrobię tego już w ładnych tabelach – zajęłyby one zbyt dużo miejsca na stronach tej książki.

Klasa: Wieża

- Obowiązki:
  - umożliwić określenie, którą z kaset wieży chcemy zobaczyć na ekranie
  - przeprowadzenie zapisu nastaw wszystkich urządzeń na dysk

- przeprowadzenie odczytu wszystkich nastaw z dysku

❖ Współpracownicy:
- kasety, którym wieża zleca działania
- menu, od którego otrzymuje rozkazy

❖ Widoczne własności:
- ma pięć kaset
- ma określony numer kasety, którą oferuje właśnie do operacji
- ma zestaw bloków, które oferuje do możliwego wstawienia

Klasa: Kaseta

❖ Obowiązki:
- narysować się na ekranie zaznaczając wstawione do niej bloki
- skoro kaseta ma 20 szufladek – to trzeba umożliwić użytkownikowi powiedzenie, o której szufladce w danej chwili myśli.
- wstawić blok do danej szufladki (jeśli jest pusta)
- usunąć blok z danej szufladki
- regulować blok tkwiący w danej szufladce

❖ Współpracownicy:
- wieża, która jej coś zleca
- bloki, które w tkwią w kasecie
- użytkownik (menu)

❖ Widoczne własności:
- 20 szufladek
- numer wybranej właśnie przez użytkownika szufladki
- zestaw bloków, które właśnie w kasecie tkwią

Klasa: Szufladka

❖ Obowiązki:
- mieścić w sobie blok

❖ Widoczne własności:
- czy jest w niej blok czy nie, i ewentualnie jaki to blok

❖ Współpracownicy:
- blok, kaseta

Klasa: Menu

❖ Obowiązki:
- pokazanie się na ekranie po starcie programu
- umożliwienie wybrania kasety
- umożliwienie wybrania szufladki w kasecie
- oferowanie użytkownikowi następujących działań:

- załadowanie z dysku konfiguracji wieży wraz z nastawami
- zapis na dysku konfiguracji wieży wraz z nastawami
- proponowanie zestawu bloków możliwych do wykorzystania w wieży i akcja wstawienia ich
- zakończenie programu

❖ Współpracownicy:

– wieża, kasety, poszczególne bloki

❖ Widoczne własności:

– sklepik z nazwami bloków do wykorzystania

– rozmiar menu, kolory itd.

Klasa: Młynek do Kawy

❖ Obowiązki:

– pozwalać regulować nastawienie grubości mielenia

– pozwalać uruchomić akcję mielenia. (Młynek zgasi się sam, jak skończy mielić)

❖ Współpracownicy:

– kaseta, w której ten młynek tkwi. (Uwaga: młynek wcale nie współpracuje z ekspresem do kawy. To nasza sprawa by zmieloną w domu, lub w tym młynku kawę umieścić w jakimś ekspresie)

❖ Widoczne własności:

– nastawiona grubość mielenia

Klasa: Ekspres do kawy

❖ Obowiązki:

– pozwalać regulować swoje nastawy

– ....

❖ Współpracownicy:

– kaseta, w której tkwi ten młynek

❖ Widoczne własności:

– ...

Klasa: Blok Wysokiego Napięcia

❖ Obowiązki:

– pozwalać ustawiać napięcia swych 256 źródeł

❖ Współpracownicy:

– kaseta, w której tkwi

❖ Widoczne własności:

– wartości 256-ściu napięć wytwarzanych przez ten blok

Klasa: Przelicznik

- Obowiązki:
  - dawać regulować swoją nastawę, czyli czas, przez który mają być zliczane impulsy
  - na dany znak uruchamiać proces zliczania
  - pokazywać zliczone wartości na ekranie
- Współpracownicy:
  - kaseta, w której tkwi
- Widoczne własności:
  - czas zliczania

Klasa: Jednostka Logiczna

- Obowiązki:
  - regulować nastawienie dwóch przełączników
- Współpracownicy:
  - kaseta, w której tkwi
- Widoczne własności:
  - dwa przełączniki o określonym w danym momencie ustawieniu

Klasa: Matryca opóźnień

- Obowiązki:
  - regulować nastawienie szesnastu obwodów opóźniających
- Współpracownicy:
  - kaseta, w której tkwi blok
- Widoczne własności:
  - obowiązujące w danym momencie ustawienie 16 czasów opóźnień

Na tym zakończyliśmy poszukiwanie klas. Pora teraz przejść do następnego etapu.

22.8.3 Etap 3 - Usystematyzowanie klas obiektów z występujących w naszym systemie

Tutaj, jak pamiętamy, mamy sprawdzić istnienie związków dziedziczenia oraz związków polegających na mieszczeniu (zawieraniu) w klasie obiektów innych klas.

Ustalenie hierarchii

Jak pamiętamy, dziedziczenie sprawdzamy wygłaszając zdania: „obiekt klasy A jest rodzajem obiektu klasy B". Próbując takich zdań na naszych klasach

dostajemy same zdania nieprawdziwe – bo: ani młynek nie jest rodzajem kasety, ani kaseta rodzajem menu itd, itd.

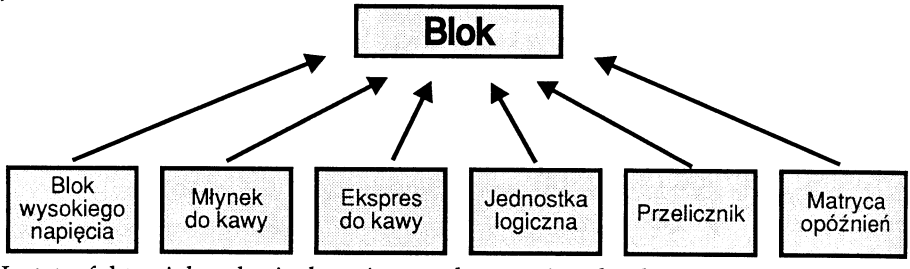
Zatem nie ma tu żadnych związków dziedziczenia? Jak widać takich zwykłych rzeczywiście nie ma. Skoro nie ma, to nie ma - **nie należy dziedziczenia robić na siłę !**

Zapytasz pewnie: „-A dziedziczenie od jakichś klas abstrakcyjnych ?"

Brawo! Czekałem na to, bo to najważniejsza rzecz w tym całym przykładzie. Otóż chyba od samego początku było jasne, że w naszym systemie jest jedna bardzo ważna klasa abstrakcyjna. Jaka? Zauważ, że ciągle mówiliśmy: wkładamy blok, usuwamy blok, regulujemy blok. Blok jest czymś abstrakcyjnym, bo czasem jest to przelicznik, czasem młynek, czasem blok wysokiego napięcia, a czasem jednostka logiczna.

Co jednoczy te wszystkie urządzenia? Mianowicie to, że wszystkie są wkładane do kasety, wszystkie pozwalają się regulować, wszystkie będą miały nastawy zapisywane na dysku lub odtwarzane z niego.

W rezultacie więc odkryliśmy, że kilka klas z naszego systemu powiązanych jest taką hierarchią:



Jest to fakt wielce doniosły – i prawdę mówiąc dzięki istnieniu tej właśnie hierarchii nasz program będzie właśnie obiektowo orientowany, a nie jedynie: „bazujący na obiektach". Wrócimy do tego jeszcze.

Mieszczenie (zawieranie) w sobie obiektów innych klas

Sprawdzając tę cechę wypowiadamy zdania typu: „A składa się między innymi z obiektu klasy B". W rezultacie znajdujemy kilka takich zdań, które brzmią sensownie:

- Wieża składa się m. inn. z kaset (pięciu!)
- Kaseta składa się m.inn. z bloków (max. dwudziestu)
- Kaseta składa się z szufladek

Oznacza to, że składnikiem obiektu klasy wieża ma być 5 obiektów klasy kaseta. (Lub wskaźników do takich obiektów). Z kolei składnikiem obiektu klasy kaseta jest 20 obiektów klasy szufladka. Składnikiem klasy szufladka może być obiekt klasy *blok* (lub wskaźnik do takiego obiektu).

Widzisz: już zaczynamy korzystać z klasy abstrakcyjnej! Gdyby nie ona musielibyśmy mówić: kaseta składa się czasem młynka do kawy, czasem bloku wysokiego napięcia, czasem...

Proponuję całość od razu trochę uprościć. Skoro szufladka jest tak mało inteligentną klasą – to po prostu otwór w kasecie – dlatego nie ma sensu z szufladki robić osobnej klasy.

Muszę ci się już teraz przyznać, że z 20 szufladek zamierzam zrobić zwykłą 20-elementową tablicę (wskaźników).

22.8.4 Etap 4 - Określenie wzajemnych zależności klas

Robimy to dwojako: sporządzając spis relacji i sporządzając graf.

Porozmawiajmy jednak najpierw o tym, jak na dysku zamierzamy zapisywać informację o stanie wieży. Zapisać trzeba:

- – jakiego typu bloki umieszczone są w jakich szufladkach kaset,
- – jakie te konkretne bloki mają nastawy.

Najlepiej zrealizować to w ten sposób, że zapisywać będziemy tak:

```
kaseta 1
     szufladka 4
          jej zawartość i nastawy
     szufladka 15
          jej zawartość i nastawy
kaseta 2
          .............itd
```

Odczyt tej zapisanej na dysku informacji odbywa się tak, że zaczyna czytanie wieża. Wiąże się to oczywiście z tym, że musi otworzyć plik, w którym te informacje są zapisane i zacząć go czytać. Kiedy napotka linijkę treści „kaseta nr..." oddaje inicjatywę właściwej kasecie by sobie poczytała odpowiedni dla niej fragment.

Zatem kaseta czyta dalej i dowiaduje się, że np. w szufladce nr 4 ma stać blok wysokiego napięcia. Umieszcza więc tam taki blok. Teraz nastąpią nastawy. Kto najlepiej odczyta z dysku nastawy tego bloku? Oczywiście on sam! Zatem kaseta zleca temu blokowi by sam czytał dalej. Blok wysokiego napięcia wie, że potrzebne są mu nastawy 256 napięć – więc czyta z dysku 256 liczb regulując się natychmiast. Gdy blok skończy czytanie, to czyta dalej kaseta i dowiaduje się, że w szufladce następnej ma stać np. młynek do kawy. Wstawia więc tam blok tej klasy. Nowonarodzony blok sam odczytuje z dysku swoją nastawę – grubość mielenia. Gdy już się „nareguluje" kaseta czyta dalej i dowiaduje się, że w szufladce nr 15 stoi jeszcze jeden młynek do kawy. Wstawia więc tam taki blok i przekazuje inicjatywę blokowi – temu drugiemu młynkowi. Blok wczytuje sobie swoje nastawy, po czym oddaje inicjatywę kasecie.

W ten sposób kaseta odczytuje informacje o wszystkich blokach tkwiących w szufladkach. Gdy skończy się odczytywanie informacji o tej kasecie, wieża poleca to samo zrobić kasecie następnej – i tak wszystkim pięciu.

Co tu jest ważne? To mianowicie, że ani wieża ani kaseta nie martwią się o odczytywanie określonych nastaw. To każdy blok samodzielnie odczytuje informacje dla siebie.

> *Wieża mówi kasecie by odtworzyła swą konfigurację, ale nie poucza ją jak to robić. Kaseta wie, że składa się z 20 szufladek więc czyta ich zawartość i dowiaduje się jakie to bloki w niej stoją.*
> *Kaseta jednak nie interesuje się tym, jakie to nastawy zapisują sobie na dysku poszczególne bloki. Po prostu poleca im by same sobie poczytały, a to jak to zrobić i co przeczytać, wiedzą już one same najlepiej*

Spis relacji

Według poprzednich zaleceń musimy spisać zależności klas. Na podstawie scenariusza a także na podstawie powyższych ustaleń

Menu:

- – zleca wieży narysowanie się na ekranie,
- – zleca wieży zapisanie na dysku jej: konfiguracji (w jakich kasetach stoją jakie bloki) i wszystkich nastaw tych bloków,
- – zleca wieży odczytanie z dysku jej konfiguracji i nastaw,
- – zleca wieży rozmowę w celu dokonania regulacji nastaw,
- – zleca wieży by wstawiła wybrany blok do określonej kasety i szufladki,
- – zleca wieży wyrzucić blok z określonej kasety i szufladki.

Wieża:

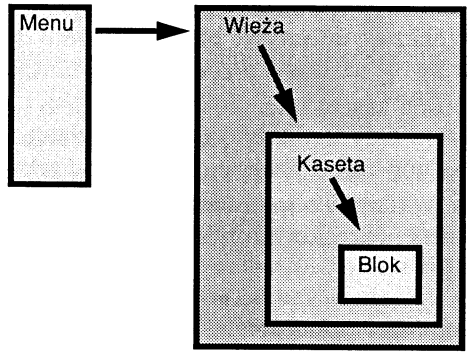
- – dowiaduje się o zapisaną na dysku konfigurację,
- – zleca kasecie by odczytała z dysku zawartość swych 20 szufladek (jakie bloki w nich stoją),
- – zleca kasecie by zapisała na dysk swą zawartość,
- – wybranej kasecie zleca by narysowała się na ekranie.

Kaseta:

- – zleca blokowi by zapisał/odczytał swoje nastawy na/z dysku,
- – zleca blokowi by powiedział kim jest (odpowiedział skrótem nazwy swojej klasy),
- – zleca blokowi by rozmawiał z użytkownikiem (jest to regulacja nastaw),

Graf

A oto jak wygląda graf dla takiego podziału obowiązków



Zauważmy kilka rzeczy:

❖ 1) Jasno widać, że składnikiem klasy wieża jest obiekt klasy kaseta. Z kolei we wnętrzu kasety tkwi blok. Abstrakcyjna klasa blok reprezen-

tuje na tym grafie wszystkie konkretne realizacje bloków — tak wymyśliliśmy w poprzednim etapie.

- 2) Żadna klasa nie rozmawia nigdy bezpośrednio z obiektem będącym składnikiem innej klasy. Naruszałoby to zasadę enkapsulacji. W niektórych momentach można odczuć taką pokusę – jednak na dłuższą metę to się nie opłaca.
- 3) Graf wygląda stosunkowo prosto. Nie popełniliśmy tu błędu, polegającego na tym, że każdy współpracuje z każdym.

22.8.5 Etap 5 - Składamy model naszego systemu

Na podstawie tego, co wymyśliliśmy w poprzednich etapach, spróbujemy wreszcie dokonać złożenia modelu. Aby to uczynić, najpierw próbujemy napisać definicje klas. Pamiętamy przy tym, że zachowania poszczególnych klas stają się ich funkcjami składowymi, a widoczne własności stają się danymi składowymi.

Właściwie sprawa jest prosta i jasna, gdyby nie istnienie klasy abstrakcyjnej

Jakie zachowania ma abstrakcyjna klasa blok?

Przykładowo musi rozmawiać z użytkownikiem i pozwalać mu regulować konkretny blok. Czyli w klasie abstrakcyjnej blok powinna być jakaś funkcja składowa o nazwie „rozmowa". Z drugiej strony jednak w wypadku każdego z konkretnych bloków taka rozmowa wygląda inaczej. Rozumiesz już co to oznacza? Ależ tak, oczywiście – funkcja ta musi być funkcją wirtualną!

Podobnie w wypadku innych zachowań klasy blok – czyli zachowań polegających na zapisaniu lub odczytaniu swoich nastaw z dysku. To oczywiste — inne nastawy zapisuje/odczytuje z dysku młynek do kawy, a inne blok wysokiego napięcia.

Oto jak wobec tego wygląda definicja takiej klasy:

```
class blok {
public:
     //————————— funkcje składowe
     virtual void rozmowa() = NULL;      // czysto wirtualna
     virtual void zapisz_sie()= NULL;
     virtual void odtworz_sie()= NULL;
     virtual char * kto_jestes() ;
     virtual ~blok() ;
} ;
```

Same nazwy mówią do czego funkcja służy. Jeśli jesteś mało domyślny, to wyjaśniam, że funkcja `odtworz_sie` służy do odczytania z dysku nastaw.

Pojawiła się natomiast nowa funkcja `kto_jestes`. Jest ona najlepszym dowodem na to, że projektowanie robi się metodą kolejnych przybliżeń. Konieczność istnienia tej funkcji wypłynęła mi naprawdę dopiero w trakcie implementacji klasy kaseta.

Jeśli kaseta ma narysować się, to musi narysować także wszystkie bloki, które aktualnie w niej tkwią. Blok na płycie czołowej powinien mieć jakiś

napis, który odróżnia go od innych bloków. Kaseta pyta więc po kolei wszystkie bloki tkwiące w jej szufladkach. Blok zapytany kto_jesteś odpowiada stringiem będącym nazwą bloku. Tę nazwę wypisuje się na obrazku. Po to, by użytkownik patrząc na obrazek łatwo rozpoznawał co w kasecie jest.

W zestawie funkcji składowych jest też destruktor. Ponieważ spodziewamy się, że w programie możemy stosować zdania typu: „wyrzucam w diabły ten blok" (zamiast: „wyrzucam w diabły tę jednostkę logiczną lub ten młynek do kawy") – zatem aby wówczas zadziałał naprawdę destruktor wyrzucanego młynka, a nie destruktor abstrakcyjnego bloku – destruktor musi być wirtualny.

Przyjrzyjmy się teraz deklaracjom innych klas.

Zacznijmy od klasy, która najwięcej rozkazuje

```
class menu {
wieza * wiezyczka ;                          // ❶
public:
     menu(wieza * x);                        // ❷
     void narysuj_menu();
     int wybor_opcji();                      // ❸
} ;
```

❶ Składnikiem daną tego menu jest wskaźnik do wieży, której menu ma rozkazywać. Pominąłem tutaj składniki opisujące kolor menu, czy jego rozmiary na ekranie. To wszystko, to kosmetyka i zależy od tego, jakim komputerem się posługujemy.

W klasie tej są także funkcje składowe. Oto kilka słów o ich roli:

❷ Konstruktor powoduje pojawienie się tego menu na ekranie, a także informuje klasę, który to obiekt klasy wieża ma być przedmiotem jego zainteresowania.

❸ Funkcja składowa `wybór_opcji` to ta, która oferuje cały garnitur usług. To w niej oferujemy użytkownikowi wybór działań na wieży. To tutaj decyduje on, którą z wielu szufladek wieży zamierza się w danym momencie zajmować. To tutaj użytkownik ma szanse coś do szufladki wstawić, czy to coś, co tam jest wstawione – regulować, lub coś stamtąd usunąć. To tutaj też podejmuje on decyzje o zapisie wszystkich nastaw wieży na dysku, lub też decyzje o odczytaniu starych nastaw z dysku.

Oto deklaracja klasy wieża:

```
const int ile_kaset = 5 ;
//////////////////////////////////////////////////////////
class wieza {
     kaseta k[ile_kaset] ;        // pięć obiektów klasy kaseta   ❶
     int akt_ka ;                 // nr aktywnej kasety           ❷
public:
     // funkcje składowe
     wieza();                                                     // ❸
     void narysuj() ;             // narysowanie wieży na ekranie
                                  // całej albo tylko interesującej części

                    //————————funkcje odpowiadające za zapis na dysk
                                  // wszystkich nastaw danej wiezy
```

```
        void zapis() ;
        void odczyt();
                        // będziemy pracować nad konkretnym miejscem w tej
                        // wieży zatem:
        void wybierz(int nr);                  // wybranie nru kasety
        void wyb_szuf(int nr) ;                // wybranie nru szufladki

                        // dowiadujemy sie o bieżący nr kasety i szufladki
        int ktora_szufladka() ;
        int ktora_kaseta() ;

                                        // w konkretne miejsce wieży można coś :
        void dodaj() ;                         // wstawić
        void usun() ;                          // można TO usunąć
        void regulacja() ;                     // można TO regulować
    } ;
```

❶ Składnikiem obiektu klasy wieża jest pięć obiektów klasy kaseta. W świecie realnym są one na stałe umieszczone w wieży za pomocą śrub, więc w klasie wieża widzimy po prostu pięcioelementową tablicę obiektów klasy kaseta.

❷ Widoczną własnością wieży jest też numer kasety, z którą właśnie się zajmujemy. (Oczywiście mamy do wyboru jedną z pięciu kaset).

> *Na ekranie mojego komputera jest to naprawdę widoczne. To jeden z 5 kwadratów, który muszę kliknąć myszką. Kiedy to uczynię, guzik ten staje się od tej pory czarny, przypominając mi ciągle, którą to kasetę na ekranie właśnie widać.*

Nazwy funkcji składowych od razu sugerują co dana funkcja robi. Uwag wymaga tylko konstruktor ❸. Skoro rezygnuję z rysowania całości wieży na ekranie – konstruktor nie robi nic szczególnego. W moim wypadku po prostu inicjalizuje tylko daną składową `akt_ka`. Przypominam, że jeśli w klasie są obiekty składowe (a są!), to ich konstruktory ruszą do pracy przed wykonaniem konstruktora klasy wieża.

Deklaracja klasy kaseta

```
const int ile_szufladek = 20 ;
/////////////////////////////////////////////////////////
class kaseta {
blok * szufladka[ile_szufladek] ; // tablica wskaźników                ❶
int akt_sz ;                      // nr właśnie wybranej szufladki      ❷
public:
    //————————————————————funkcje składowe
    kaseta () ;                                   // konstruktor         ❸
    void narysuj_sie();                   // rysowanie kasety            ❹

    void wybierz(int n) ;                 // wybranie szufladki          ❺
    int akt() ;                       // inform. o aktywnej szufl.       ❻

                          // wieża zleca takie funkcje każdej z 5 kaset
    void zapisz_bloki();                  // zapis bloków kasety         ❼
    void czytaj_blok() ;                  // odczyt bloków               ❽

    void skasuj() ;                       // wyrzucanie bloku            ❾
    void wstaw() ;                        // wstawianie bloku            ❿
```

```
        void edycja() ;                    // będzie regulacja      ❶❶
        void stworca(int nr) ;             // pomocnicza funkcja    ❶❷
} ;
```

❶ Jak pamiętamy, obiekt klasy kaseta składa się z kilkunastu szufladek, w których wsunięte mogą być bloki. Bloki te stają się składnikami kasety. Obiekt może być składnikiem kasety albo na tej zasadzie, że jest jej składnikiem naprawdę, albo na tej zasadzie, że klasa kaseta ma do niego wskaźniki. Ten drugi wariant odpowiada nam bardziej, bo skoro bloki mają być czasem wstawiane a czasem wyrzucane, to lepiej je tworzyć w zapasie pamięci, a dostęp do nich mieć za pomocą wskaźników. Dlatego właśnie daną składową klasy `kaseta` jest tablica wskaźników do obiektów klasy `blok`.

❷ Drugą daną składową jest numer szufladki, nad którą w danym momencie zamierzamy się pastwić.
Co do funkcji składowych to:

- ❖ – konstruktor❸ odpowiada za wstępne wyzerowanie kasety (kaseta zaraz po wyprodukowaniu jest przecież pusta);
- ❖ – funkcja `narysuj_sie` ❹ aktywowana jest przez wieżę. Tym sposobem wieża rozkazuje jednej z 5 kaset by właśnie ona narysowała się na ekranie;
- ❖ – funkcja `wybierz` ❺, którą wieża nakazuje kasecie by nastawiła się na pracę z określoną szufladką kasety;
- ❖ – na wypadek gdyby się ktoś potrzebował dowiedzieć o numer tej aktualnie wybranej szufladki jest funkcja `akt` ❻;
- ❖ – jeśli my z menu zażądamy by wieża zapisała swe nastawy na dysku, wieża przygotowuje plik dyskowy (czyli otwiera go i coś tam wstępnie zapisuje). Następnie rzeczona wieża zmusza do pracy po kolei wszystkie 5 kaset. Robi to wywołując dla każdej kasety jej funkcję `zapisz_bloki` ❼. W ramach tej funkcji każda kaseta załatwi się ze wszystkimi tkwiącymi w niej blokami zmuszając je do pracy ;
- ❖ – łatwo się domyślić, że podobnie jest w wypadku funkcji `czytaj_blok` ❽, za pomocą której wieża zmusza daną kasetę do odczytania z dysku informacji o wszystkich tkwiących w niej blokach ;
- ❖ – jeśli my za pomocą menu nakażemy wieży wyrzucenie jakiegoś bloku, to wieża – wiedząc gdzie ten blok stoi – wyda polecenie odpowiedniej kasecie. Zrobi to właśnie tą funkcją `skasuj` ❾;
- ❖ – natomiast do wstawienia bloku służy funkcja `wstaw` ❿;
- ❖ – na identycznych zasadach odbędzie się aktywacja funkcji `edycja` ❶❶, gdy zapragniemy regulować jakiś blok ;
- ❖ – w klasie widzimy też funkcję `stwórca` ❶❷. Ta funkcja służy do powoływania do życia odpowiednich bloków. Stwórca stwarza je w zapasie pamięci.

A oto jedna z klas pochodnych od abstrakcyjnej klasy blok – klasa młynek

Zauważ publiczne dziedziczenie klasy blok.

```
class mlynek : public blok {
     int grubosc ;
          //————————————funkcje składowe
public:
     mlynek(){ grubosc = 5 ; }
     miel();
     void rozmowa() ;
     void zapisz_sie() ;
     void odtworz_sie() ;
     char * kto_jestes() ;
     ~mlynek() { cout << "destruktor mlynka" << endl ; }
} ;
```

Zwróć uwagę, że są tu funkcje, które w abstrakcyjnej klasie `blok` są określone jako wirtualne. Tutaj więc określimy co to znaczy regulować młynki, czy też zapisywać nastawy takich młynków na dysk.

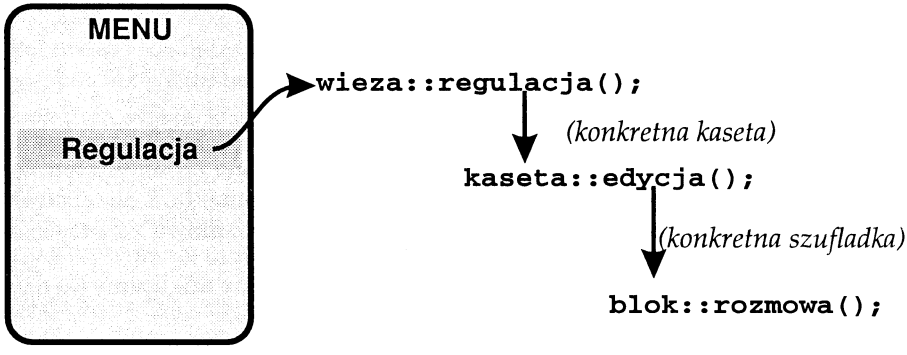
22.9 Implementacja modelu naszego systemu

Teraz w zasadzie należałoby przystąpić do implementacji czyli definiowania poszczególnych funkcji. Ze względu na oszczędność miejsca w tej książce — całości listingu Ci oszczędzę†). Przyjrzymy się tylko miejscom wyjątkowym.

Z czego w tym programie jestem dumny najbardziej? Oczywiście z tych miejsc, gdzie opłaciło mi się stosować technikę OO. Innymi słowy z tych miejsc, gdzie polimorfizm ułatwił mi życie.

Regulacja nastaw bloków

Weźmy na przykład sytuację, gdy chodzi nam o regulację danego bloku.



Menu wywołuje w naszej wieży funkcję składową `regulacja`. Jej deklarację widzieliśmy 2 strony wcześniej. Teraz ją zdefiniujmy. (Jest to tak prosta funkcja,

†) Cały program możesz ewentualnie sobie sprowadzić przez Internet z mojej strony WWW. Razem z innymi programami z tej książki jest tam pod hasłem "Symfonia C++/Source code"

że oczywiście zrobię ją `inline` – jednak nie zawracajmy sobie teraz głowy takimi sprawami).

```
void wieza::regulacja()
{
     k[akt_ka].edycja();
}
```

Funkcja ta wiedząc, którą kasetę mamy na myśli (określiliśmy to wcześniej) wywołuje funkcję składową `edycja` dla tej właśnie kasety. Jej deklarację także widzieliśmy na poprzednich stronach (w klasie `kaseta`). Oto definicja tej funkcji

```
void kaseta::edycja()
{
     if(szufladka[akt_sz])     // jeśli szufladka nie jest pusta...
     {                                 // ...to porozmawiamy z blokiem
                                       // który w niej tkwi
          szufladka[akt_sz] -> rozmowa(); // !!
     }
}
```

Wewnątrz tej funkcji widzimy, instrukcję `if`, która sprawdza czy cokolwiek jest wsunięte do danej szufladki. (Numer interesującej szufladki określiliśmy wcześniej).

Jeśli w szufladce rzeczywiście coś jest to... Tak, tak mamy tu właśnie cudowność polimorfizmu. Ponieważ `szufladka` jest tablicą wskaźników, więc linijkę tę możemy prościej zapisać tak:

```
wskaźnik_do_bloku -> rozmowa();
```

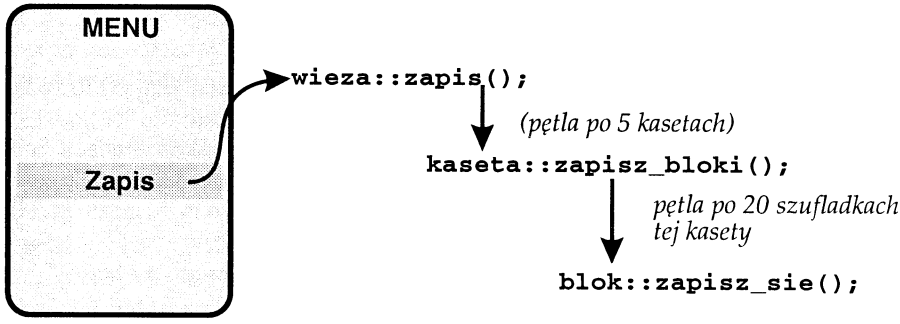
Jak pamiętamy, w klasie `blok` funkcja `rozmowa` jest funkcją wirtualną, a zatem nasz komputer sprawdzi sobie najpierw na jaki typ bloku ten wskaźnik pokazuje i uruchomi funkcję `rozmowa` z konkretnej (już nie-abstrakcyjnej) klasy. Czyli będzie to rozmowa z młynkiem, albo rozmowa z matrycą opóźnień.

A teraz pytanie: jak trzeba zmienić tę linijkę, gdy dokupimy do naszego systemu nowy typ bloku?

Odpowiedź: wcale nie trzeba zmieniać – i to właśnie jest najwspanialsze w programach obiektowo orientowanych. Wystarczy, że pisząc klasę dla tego nowego bloku uczynimy ją klasą pochodną od klasy `blok` i wtedy automatycznie nadaje się ona do takich miejsc w programie.

Zapis konfiguracji i nastaw na dysk

Teraz zobaczmy co (przez technikę OO) zyskaliśmy przy zapisie nastaw na dysk



Akcję tę inicjuje się w menu. Klasa `menu` wywołuje wtedy funkcję składową `zapis` z klasy wieża. Oto jak ją zdefiniujemy:

```
void wieza::zapis()
{
                                        //————-otwarcie pliku
     plik.open("t.sav", ios::out);                          // ❶
                                        // zapis treści poszczególnych kaset
                                        // pętla po wszystkich kasetach
     for(int i = 0 ; i < ile_kaset ; i++)                   // ❷
     {
          plik << "Kaseta " << i << endl ;                  // ❸
          k[i].zapisz_bloki();
     }
     plik.close();                      //———— zamknięcie pliku
     if(!plik){                         //———— imituję obsługę błędów
          cout << "\aBlad w trakcie pisania pliku" ;
     }
}
```

Idea jest taka: wieża, której zlecono dokonać zapisu wszystkich nastaw do pliku dyskowego – najpierw otwiera plik❶. Pomijam tu pytania o nazwę pliku. To jest przecież trywialne. Tu dla prostoty plik nazywa się zawsze `t.sav`

❷ Następnie wieża kolejno zmusza kasety będące jej składnikami, by kolejno zapisały swe nastawy. (Pętla). Ostatecznie plik jest zamykany i ewentualnie sprawdza się poprawność operacji z dyskiem.

❸ Zapis nastaw bloków znajdujących się w kasecie odbywa się – jak widzimy – przez wywołanie funkcji składowej `zapisz_bloki`. Oto jej definicja

```
void kaseta::zapisz_bloki()
{
                                        // pętla po wszystkich szufladkach kasety
     for(int i = 0 ; i < ile_szufladek ; i++)
     {
          if(szufladka[i]){                  // jesli coś w szufladce jest
                                             // zapis numeru szufladki
               plik < "\tszufladka " < i < endl ;
                                        // zapis informacji specyficznej dla
                                        // konkretnego bloku wykonuje sam ten blok
               szufladka[i]->zapisz_sie() ;
          }
```

```
        }
    }
```

Jak widzimy, jest tu znowu pętla po wszystkich (zapełnionych) szufladkach kasety. Jeśli szufladka jest zapełniona, to zapisze się jej „adres" czyli numer kasety i numer szufladki. Można to załatwić jakoś inaczej, zapisując numer kasety tylko raz, przy pierwszej szufladce danej kasety. To jest już szczegół. Ważne jest, że na dysku znajdzie się informacja o tym, gdzie jaki blok stoi. Czyli o tym, jaka jest „konfiguracja" wieży.

A teraz sprawa samych nastaw. Najważniejsza jest tu linijka następna, kiedy to kaseta zleca abstrakcyjnemu blokowi, by sam zapisał swoje nastawy. Ponieważ za pomocą wskaźnika wywołujemy funkcję, która jest funkcją wirtualną — dlatego znowu objawi się tu cud polimorfizmu. To znaczy, że wystartuje funkcja `zapisz_sie` nie z klasy abstrakcyjnej `blok`, ale z takiej klasy pochodnej od niej, do jakiej ów konkretny, pokazywany przez wskaźnik blok należy. Jeśli pokazuje na młynek, to będzie to `zapisz_sie` z młynka, jeśli pokazuje na jednostkę logiczną, to będzie to `zapisz_sie` z tej jednostki.

Nie zapytam Cię już jak zmienić tę funkcję w sytuacji, gdy dokupimy do systemu nowy blok. Oczywiście cud polega na tym, że żadna zmiana nie będzie potrzebna.

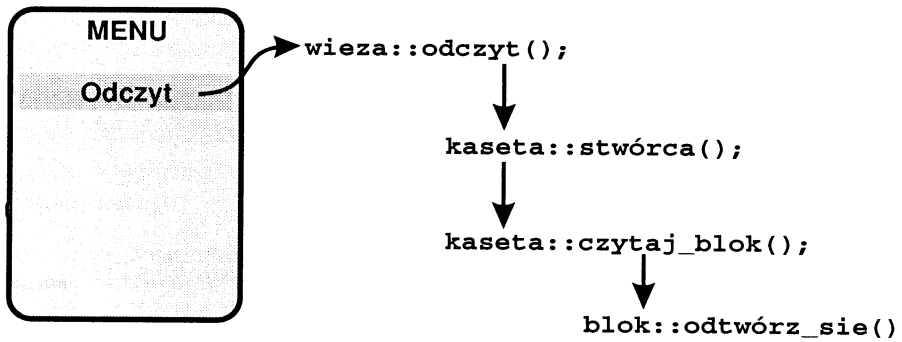
Powtórzymy jeszcze raz:

> Zapis nastaw na dysk polega na tym, że wieża przelatuje sobie wszystkie składające się na nią szufladki (5*20) i mówi:
> **„Niezależnie od tego co stoi w tej szufladce – proszę się zapisać na dysk".**

Nie powiedziałem tu jeszcze jednego: otóż blok zanim zapisze swoje nastawy, zapisze informację jakim jest blokiem. To po to, by czytając potem tę informację kaseta wiedziała komu zlecić pracę. Po samym zbiorze nastaw nie musi być to od razu jasne. Zapamiętajmy to, bo za chwilę z tego będziemy skorzystali: a więc przed nastawami konkretnego bloku zapisany jest jego identyfikator.

Odczyt konfiguracji i nastaw z dysku

Na dysku oprócz nastaw znalazła się także informacja o tym, w jakich szufladkach umieszczono jaki typ bloków.



Gdy chcemy teraz odczytać to, co w zeszłym tygodniu powyższym sposobem zapisaliśmy, to wybieramy w menu opcję „odczyt". Wówczas menu wywołuje w naszej wieży jej funkcję składową `odczyt`.

```
void wieza::odczyt()
{
int sz ;                                  // lokalne nr kasety i szufladki
int typ ;
char wyraz[50] ;
     plik.open("t.sav", ios::in); //————otwarcie pliku                ❶

                    // sukcesywne czytanie pliku i tworzenie wg tej informacji
                    // odpowiednich bloków w odpowiednich miejscach
     while(plik){                                                      // ❷
          plik >> wyraz ;                                              // ❸
          if(plik.eof()) break ;                    // gdy koniec pliku
                    --
          if(strcmp(wyraz, "kaseta") == 0)                             // ❹
          {                                         // będzie nr kasety
               plik >> akt_ka ;                                        // ❺
          }else if(strcmp(wyraz, "szufladka") == 0)                   // ❻
          {                                         // będzie nr szufladki
               plik >> sz ;                                            // ❼
               k[akt_ka].wybierz(sz);   // uaktywnienie tej szufladki

               plik >> typ ;            // ma być ten typ              ❽

               // kreacja takiego bloku, który jeszcze nie istnieje
               k[akt_ka].stworca(typ);                                 // ❾

               // niech kaseta inicjuje akcję wczytania parametrów bloku
               k[akt_ka].czytaj_blok();                                // ❿
          }
     }
     plik.close();                                                     // ❶
                         //————— zrobić gdzieś obsługę błędów
     if(!plik){
          cout << "\aBlad w trakcie czytania pliku" ;
     }
}
```


Przyjrzyjmy się tej definicji.

❶ Jest oczywiste, że plik z danymi musi zostać otwarty, zamknięty, i że powinna być jakaś kontrola czy praca z plikiem była poprawna.

Zobaczmy jak odbywa się samo czytanie. Jest to pętla `while` ❷, która skończy

Jeśli przeczytane słowo brzmi: „szufladka" ❻, to oznacza, że teraz możemy wczytać numer szufladki❼. Po nim nastąpi numer identyfikacyjny typu bloku ❽. Ten identyfikator to nic strasznego – oto jak go zrealizowałem

```
enum typy_blokow {
     mlyn = 1, expr, napi, skal, jedn, matr
};
```

Jak widać to zwykły typ wyliczeniowy. Jeśli dokupię jakiś nowy blok, to po prostu na końcu tej listy napiszę skrót jego nazwy.

❾ Gdy przeczytam identyfikator (do zmiennej `typ`), to wywołuję funkcję `stwórca`. Ta funkcja w zapasie pamięci kreuje (operatorem `new`) obiekty żądanego typu.

Oto jak wygląda `stworca`:

```
void kaseta::stworca(int nr)
{
     switch(nr){
          default :
               cout << "nieznany blok" ;
               break ;
          case mlyn :
               szufladka[akt_sz] = new mlynek ;
               break ;
          case expr :
               szufladka[akt_sz] = new express ;
               break ;
          case napi :
               szufladka[akt_sz] = new wys_napiecie ;
               break ;
          case skal :
               szufladka[akt_sz] = new skaler ;
               break ;
          case jedn :
               szufladka[akt_sz] = new jednostka_logiczna;
               break ;
          case matr :
               szufladka[akt_sz] = new matryca_opoznien;
               break ;
          // ......... tu ewentualne nowe bloki
     } // end switch
}
```

Jak widać nic tu specjalnego. Instrukcja `new` w kilku wariantach. Rezultatem działania operatora `new` jest wskaźnik do nowostworzonego obiektu. Ten to wskaźnik wstawiamy w odpowiednie miejsce tablicy `szufladka`.

Jeśli dokupię nowy blok do naszego systemu, to to jest jedno z tych niewielu miejsc, gdzie będę musiał dokonać modyfikacji. Pojawi się po prostu następny przypadek `case`. Tylko tutaj. Przy klasycznych technikach programowania trzeba by przekopać cały program.

❿ Skoro już wiemy jak działa `stworca`, wróćmy na poprzednią stronę do funkcji `odczyt`, która go wywołała. Nowonarodzony blok ma nastawy domniemane, co nas niekoniecznie zadowala. Najważniejsze jednak, że już istnieje. Skoro

istnieje, to można mu od razu kazać, by sobie właściwe nastawy z dysku odczytał. Dlatego wywołujemy funkcję składową klasy kaseta

```
k[akt_ka].czytaj_blok();
```

Oto jak wygląda definicja tej funkcji w klasie `kaseta`

```
void kaseta::czytaj_blok()
{                                        // tu jest polimorfizm
     szufladka[akt_sz] -> odtworz_sie() ;
}
```

Widzimy, że mając wskaźnik do obiektu klasy `blok` wywołujemy funkcję wirtualną z tej klasy – zatem znowu polimorfizm. Tutaj więc ruszy do pracy nie funkcja składowa abstrakcyjnego `bloku`, ale funkcja z młynka, matrycy opóźnień czy innego typu bloku, który naprawdę tam się od niedawna znajduje. Teraz powinny znowu nastąpić moje zachwyty nad polimorfizmem. (Patrz zachwyty przy zapisie danych na dysk).

Ręczne wstawianie bloków do pustych szufladek

Bloki mogą się pojawiać w kasecie nie tylko dlatego, że tak zostało przeczytane z dysku. Równie dobrze mogą tam zostać umieszczone przez użytkownika „ręcznie". To jest jakby łatwiejsza wersja tego, o czym mówiliśmy poprzednio. Zamiast czytać coś z dysku użytkownik określa szufladkę, a następnie z menu wybiera nazwę bloku, który chciałby tam umieścić. To jest ostatnie miejsce kiedy w razie zakupu nowego bloku musimy dokonać poprawki w programie. Po prostu powinniśmy do wachlarza starych bloków dopisać ten nowy typ – oferując go do użytku. Oto jak (w wersji prymitywnej – bez okienek czy myszki) wyglądać może taka funkcja. Uruchamiamy ją z klasy menu wybierając opcję „wstawianie". Powiadomiona o takim zamiarze wieża wywołuje funkcję składową `wstaw` z klasy kaseta. To ta funkcja da ofertę. Tak to wymyśliłem sądząc, że kaseta najlepiej powinna wiedzieć co do niej można wstawić. Nie upierałbym się przy tym jednak i może funkcja ta powinna być częścią klasy `menu`.

```
void kaseta::wstaw()
{
int nr ;
char c ;
     if(szufladka[akt_sz]) {
          cout << "szufladka jest zajeta\n\a"
                    "Wcisnij ENTER..." ;
     }
     cout << "Masz do wyboru takie bloki:\n"
                    " 1 - mlynek do kawy\n"
                    " 2 - ekspres do kawy\n"
                    " 3 - przelicznik\n"
                    " 4 - blok wysokiego napiecia\n"
                    " 5 - jednostka logiczna\n"
                    " 6 - matryca opoznien\n"
                    " 0 - ZADEN Z POWYZSZYCH \n"
                    " Podaj numer wstawianego bloku : " ;
```

```
        cin >> c ;

        if(!isdigit(c))return ; // sprawdzenie legalności
        cin.putback(c);

        cin >> nr ;
        stworca( (typy_blokow) nr) ; // akt stworzenia
}
```

Nie potrzebuję dodawać, że z polimorfizmu korzystamy także w kilku innych sytuacjach. Te wszystkie sytuacje łatwo rozpoznać po słowie `virtual` przy odpowiednich funkcjach w definicji abstrakcyjnej klasy `blok`. Ponieważ jednak ten rozdział nie jest o polimorfiźmie, ale o projektowaniu – skończmy na tym.

Nie od razu Kraków...

Oczywiście zdajesz sobie sprawę, że nie od razu wpadłem na wszystkie te pomysły. Jak wielokrotnie powtarzałem – praca nad takim projektem odbywa się metodą kolejnych przybliżeń. Dlatego nie martw się jeśli w Twoim pierwszym projekcie nie pójdzie Ci tak gładko.

22.10 Symfonia C++, Coda

Tym sposobem dotarliśmy do końca tej książki. Jeśli – jak radziłem – opuściłeś jakieś paragrafy – teraz możesz do nich wrócić. Myślę, że zrozumienie ich przyjdzie Ci z łatwością skoro teraz jesteś już „wtajemniczony".

Zaczniesz teraz pisać w C++ swoje własne, większe programy. Początkowo posługiwanie się C++ wymagało będzie częstego zaglądania do podręcznika. To nic. Najważniejsze by językiem tym móc wyrazić wszystko, co pomyśli głowa.

Mam nadzieję, że w trakcie tej opowieści zacząłeś już doceniać język C++ i jego finezję. Parafrazując znane powiedzenie: z językiem C++ jest jak z symfonią — to, co dla jednych jest zgiełkliwą kakofonią, dla innych jest harmonijnym, niebiańskim śpiewem.

22.11 Posłowie

Tak kończyła się „Symfonia" pierwotnych wydaniach. Dziś, z perspektywy tych kilku lat, które minęły od wydania pierwszego – mogę dodać coś jeszcze.

Gdy pisałem „Symfonię" zapytałem autora słynnej książki „Nukleare Festkörperphysik", a mojego przyjaciela Güntera Schatza: „–Günter, jeśli dokonasz

podsumowania tego wysiłku, który wkładasz w pisanie książki, a także czasu, który to zabiera – czy potem masz poczucie, że wartało?"

Günter zamyślił się i odrzekł: „–Tak, książka naprawdę daje satysfakcję".

Günter nie potrafił wówczas tej satysfakcji nazwać dokładniej, ale ja postanowiłem mu po prostu uwierzyć i pisałem dalej. Dziś – z perspektywy tych lat, które minęły od pierwszego wydania „Symfonii"– wiem, jak bardzo Günter miał rację. Pewnie bardziej niż myślał. Satysfakcja, która stała się moim udziałem przewyższyła pewnie to, o czym myślał prof. Schatz.

Gdy wchodzę do księgarni naukowej stoi tam także „Symfonia", a księgarze mówią o niej „bestseller" i „longseller". Co było powodem takiego sukcesu książki? Chyba to, że co prawda jest napisana z rzetelnością podręcznika dla studentów, ale stylem takim, jak rozmowa z szesnastoletnim chłopcem. Te dwa czynniki sprawiły, że Symfonia stała się podręcznikiem na wielu uniwersytetach i politechnikach w Polsce, i – co najważniejsze – podręcznikiem, który studenci po prostu lubią. Poeto, czy może być jeszcze piękniej?

Dostałem wiele wspaniałych listów od czytelników. Okazuje się, że Symfonię czytają nawet Polacy w Niemczech, Szwajcarii, Francji, USA i Kanadzie. Wspaniałym doświadczeniem były też bezpośrednie spotkania z czytelnikami. Czasem był to zaczytany w Symfonii student w pociągu, czasem siąsiad na bankiecie na zamku w Niedzicy, czasem niewidomy naukowiec, któremu Symfonię czyta lektor. W każdym z tych spotkań odkrywałem ludzi, którzy (nie znając mnie) od dawna uważali mnie za swojego dobrego przyjaciela. Czy może być jeszcze piękniej?

Większość czytelników podkreślała z jaką pasją czytała Symfonię. Pewna dziewczyna napisała tak:

> *...Kiedy kupiłam ksiąkę, wróciłam do domu, zaparzyłam herbatę, usiadłam i zaczęłam czytać. I na tym zakończył się mój dzień...*

Przemek S. z Łodzi napisał tak:

> *...Ponieważ Symfonię często czytałem u mojej dziewczyny i często zastawała mnie z książką – pewnego razu, gdy na chwilę wyszła i wróciła, widząc, że tym razem nie czytam, powiedziała: Masz szczęście, bo właśnie postanowiłam, że jeśli wrócę i będziesz czytać, to dostaniesz w twarz.*

Profesorze Schatz, czy może być jeszcze piękniej?

Następną książkę, — tę w której porozmawiamy o szablonach i obsłudze sytuacji wyjątkowych — zatytułowałem „Pasja C++".

INDEKS

!

sklejacz 124
#define 112-114
~ a const 47
#elif 120
#else 119
#endif 118
#error 121
#if 118
#ifdef 120
#ifndef 120
#include 123
#line 122
#pragma 125
#undef 115
% (operator modulo) 55
& operator pobrania adresu 152
&& operator iloczynu logicznego 61
, (przecinek) operator 71
\n 9
__cplusplus 126
__DATE__ 125
__LINE__ 125
__NAME__ 125
__TIME__ 126
| | operator sumy logicznej 61
łączność 74, 427

A

abstrakcyjna klasa 571-577, 706
adjustment 609
adres
~ funkcji 215
~ funkcji a jej nazwa 208
~ funkcji przeładowanej 246-249
~ tablicy 133
agregat 380
aktualny argument funkcji 81
analiza zachowań obiektów 702
anonimowa unia 321
anonimowy strumień 685, 688, 691-692
ANSI 11
ANSI C 5, 158
app 646
argument
~ aktualny funkcji 81
~ będący obiektem 282-285
~ będący tablicą 175
~ będący wskaźnikiem do funkcji 216
~ domniemany 87-89
~ formalny a aktualny 82
~ formalny funkcji 81
~ funkcji będący tablicą 132-135
~ funkcji, jego nazwa 77
~ identyczny czy różny dla przeładowania 238-245
~ nienazwany 90
~ przesyłanie przez wartość 81-82

~ przesyłany przez referencję 83-85
~ wskaźnikiem do const 178
~ wywołania funkcji 81
~ wywołania programu 223-226
argumentowość operatora 427
arytmetyczny operator 54-59
ASCII 37
ate 646
automatyczna rezerw. miejsca 682
automatyczny obiekt 97
awans 253

B

błędy pracy strumienia 650-659
o babci przypowieść 83
bad() 652
badbit 650
basefield 610
białe znaki 8, 624
biblioteczna funkcja 107-109
biblioteka
~ iostream 591
~ standardowa we/wy 590
~ stdio 590, 675-678
~ stream 590
binarne wczytywanie 638
binary 646
"bit po bicie" 516
bitowy operator 63-65
blok
~ instrukcji 17
~ lokalny 43
break 23-24
buforowanie strumienia 594, 623

C

C klasyczny 5
case 23
cerr 593
chwilowy obiekt 285
ciało
~ funkcji 76
~ klasy 260
cin 593
clear(int) 655
clog 593
close() 649
const 45-47
~ a #define 47
~ a przeładowanie 306
~ funkcja składowa 303-306
~ obiekt 158
~ przy wysyłaniu argumentu do funkcji 178
~ wskaźnik 192
~ wskaźnik do takiego obiektu 193
continue 27
cout 593
cykl życiowy obiektu 709
czas życia 183
~ a zakres ważności 42-43
~ obiektów globalnych 343
~ obiektów lokalnych 343
~ obiektów new 344
~ obiektu 96-101
~ obiektu - definicja 42
czysto wirtualna funkcja 574

D

długość a rozmiar stringu 138
dana składowa 261
dane składowe publiczne - odwołanie się 277
debugger 8, 48, 93, 623, 661
dec 608-609, 622, 627
default 23
definicja 102, 105
~ „w biegu" 34
~ a deklaracja 13, 31
~ funkcji 76
~ funkcji zaprzyjaźnionej w klasie 312
~ klasy 260
~ klasy lokalna 331-333
~ klasy zagnieżdżona 328-335
~ obiektu klasy 268
~ po łacinie 31
~ przeładowanego operatora 425
~ wskaźnika 151
definiowanie stałych za pomocą #define 114
deklaracja 102, 105
~ a definicja 13, 31
~ dostępu 503
~ funkcji 76
~ funkcji – nieobowiązkowa 86
~ po łacinie 31
~ przyjaźni 308
~ zapowiadająca 311, 523

dekrementacji operator 57
delete 182
~ a NULL 348
destrukcja obiektu złożonego 359
destruktor 291-294, 336-376
~ a const i volatile 348
~ a dziedziczenie 505
~ a przeładowanie 347
~ a unia 348
~ jawne wywołanie 348
~ kolejność pracy 358
~ kolejność wywołania 509-514
~ wirtualny 578-582
~ wywołanie z this 349
~ zastosowania 294, 347
Deutsche Oper (dziedziczenie prywatne) 534
do... while... 19
domeny zastosowania wskaźników 159
dominacja 549
domniemany
~ argument 87-89
~ argument a przeładowanie 232
~ konstruktor 349
dopasowanie
~ a konwersja 415-419
~ etapy 251-255
~ funkcji przeładowanych 250
~ z awansem 253
dorzecze, schemat konwersji 420
dosłowne stałe 35-39
dostęp
~ a dwuznaczność 419
~ do skł. odziedziczonych 498-503
~ do składników klasy 264-266
~ private 266
~ protected 266
~ public 266
~ wybiórczo 503
dostępu
~ deklaracja 503
~ specyfikator 496
drukowanie z programu 691-692
duża tablica, przykład 665-671
dwuargumentowy operator 54
dwuznaczność a dostęp 419
dynamiczna alokacja tablicy 181-191
dyrektywa
~ preprocesora 110
~ pusta # 125
dziedziczenie 495-551
~ a operatory we/wy 604
~ a wieloznaczność 526
~ a zawieranie 529-530
~ kaskadowe 507
~ kilkupokoleniowe 506
~ operatorów 463
~ prywatne 534
~ prywatne (Deutsche Oper) 534
~ prywatne, kiedy 502, 507
~ wielokrotne 522-528
~ wirtualne 542-551

E

ekran 591
eksplozja, schemat konwersji 420
else 16-17
endl 623
ends 624
enkapsulacja 263, 502, 507, 708
enum 32, 52-53
~ a przeładowanie 238
EOF 634, 640
eof() 651
eofbit 650
etykieta 26, 43
~ private 264-266
~ protected 264-266, 500
~ public 264-266
~ zakres ważności 43, 96
exception handling 3
EXCLUSIVE OR operator 66
extensibility 561, 697
extern 31, 102, 106

F

fałsz-prawda 15
fail() 652
failbit 650
fill(int) 619, 625
fixed 608-609
flagi stanu
~ błędu strumienia 650
~ formatowania 608-611
flags(long) 616
floatfield 611
flush 623

for 20-21
formalny argument funkcji 81
format operacji we/wy 607-611
formatowanie wewnętrzne 679-690
free store 185, 379
funkcja 75-109, 694
~ adres f. przeładowanej 246-249
~ argument aktualny 81
~ argument będący wskaźnikiem 172-180
~ argument formalny 81
~ argument wskaźnikiem do const 178
~ biblioteczna 107-109
~ czysto wirtualna 574
~ deklaracja 76
~ domniemane przesyłanie obiektów 285
~ dwa sposoby wysyłania jej tablicy 175
~ inline 91-94
~ jej definicja 76
~ jej nazwa 76, 208
~ konwertująca 404-409
~ main 6
~ nawiasy w wywołaniu 208
~ operatorowa jako przyjaciel 433
~ operatorowa jako składowa 430-432
~ outline 93
~ przeładowana a wirtualna 570
~ przeładowana dopasowywanie 250
~ przeładowanie nazwy 228
~ przekazywanie jej tablicy 132-135
~ przesyłanie argumentu przez wartość 81-82
~ przesyłanie obiektów przez referencję 285
~ rozmieszczenie tychże w kilku plikach 102-106
~ składowa 261-262, 269-275
~ składowa - definiowanie 271
~ składowa - wywołanie dla obiektu 270
~ składowa - wywołanie dla referencji 271
~ składowa - wywołanie dla wskaźnika 271
~ składowa const 303-306
~ składowa inline 273
~ składowa static 431
~ składowa statyczna 299-301
~ składowa statyczna- a this 301
~ składowa statyczna- wywołanie 301
~ składowa volatile 303-306
~ składowa, wskaźn. do niej 394-395
~ sygnatura jej 562
~ wirtualna 552-589, 723
~ wirtualna inline 570
~ wirtualna w klasie pochodnej 563
~ wirtualna, a przyjaźń 564
~ wirtualna, a static 564
~ wirtualna, dostęp do niej 563
~ wskaźnik do niej 206-222
~ wysłanie do niej elementu tablicy 136
~ wysłanie do niej stringu 199
~ wysyłanie do niej obiektu 282-285
~ z wielokropkiem, dopasowanie 256
~ zaprzyjaźniona 307-317
~ zaprzyjaźniona a wirtualność 564
~ zaprzyjaźniona zdefiniowana w klasie 312
~ zwracanie rezultatu 78-79
funkcja we/wy
~ bad() 652
~ clear(int) 655
~ close() 649
~ eof() 651
~ fail() 652
~ fill(int) 619, 625
~ flags(long) 616
~ gcount() 640
~ get(char &) 633
~ get(char*, int, char ='\n') 634
~ get(void) 634
~ getline(char *, int, char '\n') 636
~ good() 651
~ ignore() 639
~ open(char*, int, int) 646
~ peek() 641
~ precision(int) 619
~ put(char) 643
~ rdstate() 654
~ read(char*, int) 638
~ setf 627
~ setf(long) 614
~ str() 684
~ tellg() 664
~ tellp() 664
~ unsetf 627
~ unsetf(long) 614
~ width(int) 616, 624
~ write(const char*, int) 643

G

gcount() 640
get(char &) 633
get(char*, int, char ='\n') 634
get(void) 634
getline(char *, int, char '\n') 636
globalne
~ nazwa 43
~ obiekt 96
~ wskaźnik 195
~ zmienna 694
globalny obiekt
~ wstępna inicjalizacja 101
good() 651
goodbit 650
goto 25-26, 43, 694
graf
~ dziedziczenia 506
~ współpracy klas 707, 721

H

heap 185
hex 608-609, 622, 627
hierarchia 508, 535, 703, 706, 719
hybrydowość 2, 696

I

identyfikacja obiektów 703
identyfikacja zachowań systemu 703
if 16-17
ignore() 639
in 646
inicjalizacja 455
~ a klasy podst. wirtualne 546
~ definicja 46
~ konwersja w niej 413
~ obiektu 287
~ przy dziedziczeniu 515-521
~ struktury 383
~ tablicy 131, 386
~ tablicy obiektów 380-387
~ unii 321
~ zbiorcza 132
inicjalizator kopiujący 361-376
inkrementacji operator 57
inline 91-94, 283
~ a makrodefinicja, porównanie 116
~ a wirtualność 570
~ funkcja składowa 273
~ takiż przyjaciel 603
instrukcja
~ blok tychże 17
~ sterująca 15-29
internal 608-609
ios::adjustment 609
ios::app 646
ios::ate 646
ios::badbit 650
ios::basefield 610
ios::beg 664
ios::binary 646
ios::cur 664
ios::dec 608-609
ios::end 664
ios::eofbit 650
ios::failbit 650
ios::fixed 608-609
ios::floatfield 611
ios::goodbit 650
ios::hex 608-609
ios::in 646
ios::internal 608-609
ios::left 608-609
ios::nocreate 646
ios::noreplace 646
ios::oct 608-609
ios::out 646
ios::right 608-609
ios::scientific 608-609
ios::seek_dir 664
ios::showbase 608-609
ios::showpoint 608-609
ios::showpos 608-609
ios::skipws 608-609
ios::stdio 608-609
ios::trunc 646
ios::unitbuf 608-609
ios::uppercase 608-609
iostream
~ biblioteka 591
~ klasa 593
istream 593

J

jawna konwersja, jej zapis 413-414
jawne wywołanie konstruktora 345
jednoargumentowy operator 57
język obiektowo orientowany 693

K

karty modelujące 704
kaskadowe dziedziczenie 507
kasowanie tablicy w zapasie pamięci 380
kilkupokoleniowe dziedziczenie 506
klamry 28-29
klasa 259-306
~ a obiekt - różnica 267-269
~ a typ 260
~ abstrakcyjna 571-577, 706
~ ciało 260
~ definicja 260
~ dominuje 549
~ lokalna 331-333
~ najbardziej pochodna 547
~ o zagnieżdżonej definicji 328-335
~ ogólna 509
~ pochodna 496
~ podstawowa 496
~ podstawowa bezpośrednia 506
~ podstawowa pośrednia 506
~ podstawowa wirtualna 542-551
~ prywatna 359
~ składniki 261-262
~ wskaźnik do jej obiektów 378
~ zaprzyjaźniona 315
klasyczny C 5
klawiatura 591
kolejka 509
komentarze 11
komentarze zagnieżdżane 11
komórka pamięci adresowana wskaźnikiem 181
kompilacja warunkowa 118-120
kompilator 8
komputer steruje pomiarami 429
konkatenacja stringów 204
konstrukcja obiektu z obiektami składowymi 353-358
konstruktor 286-290, 336-376
~ 2 etapy pracy 351
~ a const i volatile 337
~ a dziedziczenie 504
~ a static 337
~ a unia 337
~ a virtual 337
~ do konwersji 401-403
~ domniemany 341, 349, 478, 548
~ domniemany dla elementów tablicy 386
~ i new 344
~ jawne wywołanie 345
~ jego adres 337
~ klasy pochodnej 510, 524
~ kolejność wywołania 509-514
~ kopiujący 361-376, 455
~ kopiujący dla obiektów const 370
~ kopiujący generowany automatycznie 372
~ kopiujący klasy pochodnej 517
~ kopiujący użyty niejawnie 362
~ na cudzej liście inicjalizacyjnej 383
~ nie-publiczny 359-360
~ przeładowanie nazwy 290
~ przeładowany 337
~ pułapka przy domniemanym 342
~ wirtualności jego symulacja 583-587
konstruowanie obiektu złożonego 359
kontrakt 708
konwersja 59, 399-421
~ a dopasowanie 415-419
~ argumentów operatora 535
~ argumentu funkcji 535
~ arytmetyczna 255
~ dziesiątkowa 610
~ jawnie 413-414
~ kaskadowo 416
~ konstruktorem 401-403
~ niejawna 485
~ niejawnie 401, 412
~ operator tejże 404-409
~ ósemkowa 610
~ przy inicjalizacji 535
~ referencji 255
~ referencji do klasy pochodnej 531-541
~ rezultatu funkcji 535
~ rzutowaniem 406, 414
~ standard. wskaźnika do skł. klasy 540
~ standardowa 254
~ standardowa przy dziedziczeniu 531-541
~ szesnastkowa 610
~ trywialna 252
~ typów całkow. i zmiennoprzecink. 255
~ typów zmiennoprzecinkowych 254
~ typu całkowitego 254
~ wskaźnika do klasy pochodnej 531-541

~ wskaźników 255
~ wywołanie funkcji 406, 414
~ zachodzi niejawnie gdy 412
kopiujący konstruktor 361-376, 455
kwalifikator zakresu 497

L

l-value 155
l-wartość 155, 466
left 608-609
liczby zespolone 399, 422-423
likwidacja obiektu 291, 347
liniowe programowanie 694
linker 8, 103, 182, 235, 272, 283, 317
linkowanie z innymi językami 235
lista
~ inicjalizacyjna konstruktora 350-352, 509-514
~ inicjalizacyjna, wywołania konstruktorów 383
~ inicjalizatorów a l. inicjalizacyjna 381
~ pochodzenia klasy 496, 503, 523
~ wyliczeniowa 53
logiczny operator 60-62
lokalne -
~ definicja klasy 331-333
~ nazwa typu (typedef) 334-335
~ obiekt a wstępna inicjalizacja 101
~ obiekt automatyczny 343
~ obiekt statyczny 343
~ zakres ważności 42
~ zmienna 694

M

main 6
makrodefinicja 115-117
makrodefinicja i nawiasy 117
manipulator 613, 621-630
~ bezargumentowy 622
~ dec 622
~ definiowanie przez użytkownika 628
~ endl 623
~ ends 624
~ flush 623
~ hex 622
~ oct 622
~ parametryzowany 624
~ precision(int) 626
~ predefiniowany 621
~ resetiosflags(long) 627
~ setbase(int) 627
~ setfill(int) 625
~ setiosflags(long) 627
~ setw 624
~ ws 624
model
~ składanie 709
~ widoczne własności 710
~ zachowania 710
modelowanie 2, 698, 700

N

nagłówkowy plik 103
najbardziej pochodna klasa 547
naukowa notacja 611
nawias pusty w deklaracji funkcji 77
nazwa
~ argumentu funkcji 77
~ funkcji 76, 208, 215
~ nie jest obiektem 167
~ obiektu 183
~ statyczna globalna 106
~ tablicy 133, 379
~ tablicy a wskaźnik 166
~ typu lokalna 334-335
~ w C++ 12, 30
~ w funkcji zakres ważności 95
~ w klasie, zakres ważności 262
~ zasłanianie 44, 278-280
negacji operator ! 63
new 182
~ i konstruktor 344
~ nadanie obiektowi wartości w momencie stworzenia 185
~ wstępna zawartość tak stworzonego obiektu 183
new-line 9
niebuforowany strumień 594
nieformatowane operacje we/wy 631-632
niejawna konwersja 485
nienazwany argument 90
nieruchomy wskaźnik 192
nieskończona pętla 21
nocreate 646
noreplace 646
notacja
~ liczb 611
~ naukowa (wykładnicza) 597, 611

nowa linia 9
NULL 39, 136
~ a wskaźnik 195
NULL adres 171
numeracja elementów tablicy 129

O

obiekt
~ a klasa - różnica 267-269
~ automatyczny 97
~ automatyczny a destruktor 347
~ chwilowy 285, 402
~ chwilowy a destruktor 347
~ const 45, 158
~ definiowanie 268
~ globalny 96, 343
~ globalny, definiowanie 343
~ inicjalizacja 287
~ klasyfikacja go 703
~ konstruowany new 344
~ likwidacja 347
~ lokalny automatyczny 343
~ lokalny statyczny 343
~ lokalny, definiowanie 343
~ new a destruktor 348
~ przesyłany przez wartość 537
~ register 48
~ składnikiem klasy 353-358
~ stały 45
~ stały a konstruktor 350
~ static 98
~ statyczny a destruktor 347
~ statyczny globalny 106
~ statyczny lokalny 98
~ volatile 49
~ w środku innego obiektu 353-358
obiektowe programowanie 695
obiektowo orientowana technika 2
obiektowo orientowane programowanie 696-698
obiekty tworzone dzięki new 183
oct 608-609, 622, 627
odejmowanie dwóch wskaźników 168
odniesienie się do obiektu 152
ogranicznik stringu 39
open(char*, int, int) 646
operacja rzutowania 157
operacje wejścia/wyjścia 7, 590-692
~ błędy 650-659
~ na plikach 644-649
~ nieformatowane 631-632
operator 54-74, 464-467, 666
~ ! negacji 63
~ % modulo 55
~ & (adres) 152
~ & | ~ ^ 66
~ && oraz | | 61
~ (), przeładowanie 468-469
~ * (odniesienie się do obiektu) 152
~ , (przecinek) 71, 144
~ -> 261
~ ->, przeładowanie 470-476
~ . (kropka) 261
~ << 64
~ >> 65
~ łączność 427
~ łączność operatorów 74
~ argumentowość 427
~ arytmetyczny 54-59
~ bitowego iloczynu 66
~ bitowej negacji 66
~ bitowej sumy 66
~ bitowy 63-65
~ bitowy a logiczny - porównanie 66
~ dekrementacji 57
~ delete 182
~ delete - a dwukrotnie kasowanie 187
~ delete, przeładowanie 479
~ dwuargumentowy 54
~ dwuargumentowy, przeładowanie 438-440
~ exclusive or (XOR) 66
~ i konwersja 412
~ inkrementacji 57
~ jako funkcja globalna 436
~ jako funkcja składowa 437
~ jednoargumentowy 57
~ jednoargumentowy + , - 57
~ jego priotytet 71-73
~ konwersji 404-409
~ logiczny 60-62
~ logiczny a bitowy - porównanie 66
~ new 182
~ new, przeładowanie 477-478
~ postdekrementacji 59
~ postinkrementacji 59
~ predefiniowany 434

~ preinkrementacji 59
~ priorytet 426
~ przeładowany, definicja 425
~ przeładowany, lista 425
~ przemienność przy przeładowaniu 440
~ przesunięcia bitów w lewo 64
~ przesunięcia bitów w prawo 65
~ przypisania 59, 455, 463
~ przypisania (dodatkowe) 67
~ przypisania a dziedziczenie 505
~ przypisania klasy pochodnej 517
~ przypisania private 463
~ relacji 60
~ różnicy symetrycznej 66
~ rzutowania 70
~ sizeof 69
~ tworzący typ pochodny 41
~ wkładania do strumienia 594, 600-606
~ wyjmowania ze strumienia 594, 600-606
~ zakresu 45
OSIRIS - urządzenie pomiarowe 711
ostream 593
out 646
outline funkcja 93
overload, słowo kluczowe 228

P

parametr aktualny funkcji 81
parametr formalny funkcji 81
peek() 641
pętla nieskończona 21
plik dyskowy 591
plik dyskowy - operacje we/wy 644-649
plik nagłówkowy 103
plik, tryby otwarcia 646
pochodny typ 40-41
podprogram 75
podsystemy 708
pola bitowe 152, 323-327
pole justowania 609
pole notacji liczb 611
pole podstawy konwersji 610
polimorfizm 559-561, 696-697
~ anulowany 527
porównywanie wskaźników 169
prawda-fałsz 15
precision(int) 619, 626
predefiniowany
~ manipulator 621
~ strumień 593
preprocesor 110-127
~ dyrektywa 110
priorytet
~ operatora 71-73, 426
~ operatorów we/wy 598-599
private 264-266
procedura 694
proceduralne programowanie 694
program
~ który drukuje 691-692
~ OO, projektowanie 693-735
~ składający się z kilku plików 102-106
~ wywołanie go z argumentami 223-226, 688
~ zawiesza się 168
programowanie
~ liniowe 694
~ obiektowe 695
~ obiektowo orientowane 696-698
~ proceduralne 694
~ z ukrywaniem danych 695
projektowanie OO 693-735
projektowanie programu OO 699-700
protected 264-266, 500
prywatna klasa 359
przeładowanie
~ operatora 464-467
~ i zasłonięcie 281
~ a const 306
~ a przyjaźń 312
~ a volatile 306
~ a wirtualność 570
~ a zakres ważności nazwy funkcji 236-237
~ nazw funkcji a technika OO 233-234
~ nazwy funkcji 228
~ operatora 422-493
~ operatora [] 666
~ operatora << 486-491
~ operatora delete 479
~ operatora dwuargumentowego 438-440
~ operatora jednoargumentowego 435-437
~ operatora new 477-478
~ operatora postdekrementacji 480-481
~ operatora postinkrementacji 480-481
~ operatora przypisania 451-463
~ operatora() 468-469
~ operatora-> 470-476

~ zaoszczędzone dzięki konwersjom 402
przeładowanie nazw funkcji 227-257
przeładowanie nazwy funkcji
~ a domniemany argument 232
przeładowany operator
~ definicja 425
~ lista 425
przecinek 71, 144
przecinki (dwa obok siebie) 89
przesunięcie bitów
~ w lewo 64
~ w prawo 65
przez wartość przesłanie obiektu 537
przyjaźni deklaracja 308
przylegające stringi 40
przypisania (dodatkowe) opratory 67
przypisania operator 59
przypisanie 455
~ "bit po bicie" 516
~ "składnik po składniku" 515
~ definicja 46
~ przy dziedziczeniu 515-521
public 264-266
put(char) 643

R

rdstate() 654
read(char*, int) 638
referencja 41, 152
~ a przeładowanie 242
~ argumentem funkcji 83-85
~ do klasy pochodnej 531-541
register 48
relacji operator 60
resetiosflags(long) 627
return 76, 78-79, 400
~ mechanizm 173
reusability 537, 697, 705
rezerwacja obszarów pamięci 181-191
rezultat zwracany przez funkcję 78-79
right 608-609
rozbudowalność C++ 697
rozmiar a długość stringu 138
rozmiar tablicy
~ a długość stringu 687
~ określanie w definicji 128
rozszerzalność C++ 561, 697
rzutowanie 70

S

słowa kluczowe C++ 12
scientific 608-609
seek_dir 664
sekwencja działań obiektu 709
sekwencje trzyznakowe 111
set_new_handler 190
setbase(int) 627
setf(long) 614, 627
setfill(int) 625
setiosflags(long) 627
setw 624
showbase 608-609
showpoint 608-609
showpos 608-609
signed 32
size_t 477
sizeof operator 69
sizeof(wskaźnik) 618
składnik
~ będący obiektem innej klasy 263
~ const w klasie 463
~ dana 261
~ dana statyczna 295-298
~ funkcja 262
~ klasy 261-262
~ klasy, dostęp 264-266
~ odziedziczony 496
~ odziedziczony, dostęp 498-503
~ referencja w klasie 463
~ statyczny miejsce definicji 298
~ statyczny, deklaracja a definicja 296
~ statyczny, jego typ 299
~ statyczny, trzy sposoby odniesienia się 296
~ statyczny, wskaźnik doń 398
~ statyczny, zastosowanie 302
"składnik po składniku" 515
skipws 608-609
spacje 596, 598
spaghetti kod a la 694, 708
specyfikator dostępu 496
spis relacji klas 707, 721
stała
~ za pomocą #define 114
stała dosłowna
~ całkowita 35
~ string 39

~ zmiennoprzecinkowa 36
~ znakowa 37
stałe
~ dosłowne 35-39
~ tekstowe 39
stały
~ obiekt 45
~ wskaźnik 192, 379
standardowa biblioteka 107
standardowa biblioteka we/wy 590
static
~ a funkcja wirtualna 564
~ funkcja 299-301
~ funkcja składowa 431
~ obiekt 98, 101
~ składnik w klasie 295-298
~ string 206
statyczna
~ funkcja składowa 299-301
~ funkcja składowa - wywołanie 301
statyczny
~ obiekt globalny 106
~ obiekt lokalny 98
~ składnik miejsce definicji 298
~ składnik, jego typ 299
~ składnik, zastosowanie 302
stdio
~ biblioteka 590, 675-678
~ flaga ios:: 608-609
sterowanie formatem operacji we/wy 607-611, 614
stos 80
str() 684
stream biblioteka 590
streampos 663
string 35, 39, 136
~ argumentem funkcji 199
~ bardzo długi 40
~ długość a rozmiar 138
~ konkatenacja 204
~ ogranicznik 39
~ w cudzysłowie jest static 206
~ wariacje na temat 199-205
~ zapis w kilku linijkach 40
stringi
~ przylegające 40
struktura 318, 503, 524
~ inicjalizacja 383
strumień 592
~ anonimowy 685, 688
~ buforowany 594
~ niebuforowany 594
~ otwieranie 646
~ predefiniowany 593
~ predefiniowany, domniemania 595-597
~ tryb pracy 646
strzał na oślep 194-195
switch 22-23
sygnatura funkcji 562
symetria operacji we/wy 604
system 702

T

tablica 40, 128-148
~ bardzo duża, przykład 665-671
~ dwa sposoby wysyłania jej do funkcji 175
~ dynamiczna alokacja 186
~ inicjalizacja 131
~ inicjalizacja zbiorcza 132
~ jej adres 133
~ jej elementy 129-130
~ jej nazwa 133, 379
~ new, kasowanie jej 380
~ numeracja elementów 129
~ obiektów definiowana new 379
~ obiektów inicjalizacja 380-387
~ obiektów jakiejś klasy 377-387
~ określanie rozmiaru w definicji 128
~ przekazywanie do funkcji 132-135
~ rezerwowana dynamicznie 181-191
~ rozmiar 135
~ rozmiar a długość stringu 687
~ wielowymiarowa 144-148, 469
~ wielowymiarowa a przeładowanie 240
~ wskaźników 197-198
~ wskaźników do danych składowych 396
~ wskaźników do funkcji 220
~ wskaźników do funkcji składowych 397
~ wymaganie konstr. domniemanego 386
~ wysłanie do funkcji jednego jej elementu 136
~ znakowa 136-143
technika obiektowo orientowana 2, 693
technika OO
~ a przeładowanie nazw funkcji 233-234
technika proceduralna 710
tellg() 664

tellp() 664
templates 3
this 276
~ a funkcja składowa statyczna 301
~ a funkcja zaprzyjaźniona 311
~ dla destruktora 349
~ typ tego wskaźnika 277
trunc 646
tryby otwarcia pliku 646
trywialna konwersja 252
trzyznakowe sekwencje 111
typ 30-53
~ a klasa 260
~ definiowany przez użytkownika 259-260
~ fundamentalny 32-34
~ pochodny 40-41, 261
~ streampos 663
~ systematyka 32
~ void 42
~ void* 42
~ wbudowany 32
~ wskaźnika do funkcji 214
~ wyliczeniowy enum 52-53
~ zdefiniowany przez użytkownika 32
typedef 50-51, 415
~ a przeładowanie 238
~ lokalne 334-335

U

ukrywanie danych, programowanie 695
ukrywanie informacji 264-266
unia 319-322
~ anonimowa 321
~ inicjalizacja 321
~ rozmiar 319
unitbuf 608-609
unsetf(long) 614, 627
unsigned 32
uppercase 608-609
ustawianie wskaźnika 152-154

V

virtual
~ funkcja a klasa 557
void 42
void* wskaźnik 156-158
volatile 49
~ a przeładowanie 306
~ funkcja składowa 303-306

W

warunkowa kompilacja 118-120
warunkowe wyrażenie 68
wbudowany typ 32
wejścia/wyjścia oeracje 590-692
while 18
wiązanie
~ późne 565
~ wczesne 565
~ wczesne a wirtualność 566-567
widmo 429
widoczne własności 710
width(int) 616, 624
wielokrotne dziedziczenie 522-528
wielowariantowy wybór 18
wielowarintowy wybór 22
wielowymiarowa
~ tablica 144-148, 469
~ tablica a przeładowanie 240
wieloznaczność
~ a dziedziczenie wirtualne 544
~ przy dziedziczeniu 526
~ przy konwersji 411, 418, 421
wirtualna
~ funkcja 552-589, 723
~ klasa podstawowa 542-551
wirtualne dziedziczenie 542-551
wirtualność
~ a operatory we/wy 606
~ a przeładowanie 570
~ a wczesne wiązanie 566-567
wolny format zapisu 7
write(const char*, int) 643
ws 624
wskaźnik 40, 149-226
~ a nazwa tablicy 166
~ argumentem funkcji 172-180
~ arytmetyka 167
~ będący obiektem statycznym 195
~ definiowanie 151
~ do const 193
~ do funkcji 206-222
~ do funkcji - definiowanie i odczytywanie deklaracji 209
~ do funkcji - typ 214
~ do funkcji argumentem innej f. 216
~ do funkcji składowej 394-395
~ do funkcji, tablica tychże 220

~ do klasy pochodnej 531-541
~ do NULL 195
~ do obiektów klasy 378
~ do składnika-danej 390-393
~ do składników klasy 388-398
~ do składników statycznych 398
~ do stałej 193
~ dodawanie i odejmowanie liczby całkowitej 167
~ domeny zastosowania 159
~ dostęp do komórek pamięci 181
~ globalny 195
~ konwersja 255
~ odejmowanie dwóch od siebie 168
~ pisania i czytania 662-664
~ porównywanie 169
~ poruszanie nim 159
~ sposoby ustawiania 196
~ stały 192, 379
~ stały, inicjalizacja 193
~ tablica tychże 197-198
~ this 276
~ this, jego typ 277
~ typu void 156-158
~ ustawianie 152-154
~ w zastosowaniu do tablic 159-171
~ zręczny 471, 481
~ zwykły, we wnętrzu obiektu 389
wskaźnik czytania get 663
wskaźnik pisania put 663
wybór wielowariantowy 18, 22
wyliczeniowy typ enum 52-53
wyrażenie warunkowe 68

Z

zagnieżdżanie komentarzy 11
zagnieżdżanie zakresów 496
zakres leksykalny 603
zakres ważności
~ a czas życia 42-43
~ definicja 42
~ definicji klasy wewnętrznej 328
~ etykiety 96
~ funkcja 43
~ klasa 44, 264
~ lokalny 42
~ nazw w funkcji 95
~ nazw w klasie 262
~ nazwy funkcji a przeładowanie 236-237
~ nazwy obiektu 96-101
~ obiektów globalnych 343
~ obiektów lokalnych 343
~ obiektów new 344
~ plik 43
~ zagnieżdżanie 496
zakres ważności nazwy 183
zakresu operator 45
zapas pamięci 185, 379, 477
zapas pamięci - wyczerpanie 189
zapowiadająca deklaracja 311
zaprzyjaźniona klas 315
zasłanianie nazw 44, 278-280, 332
zasłanianie składnika 497
zasłonięcie i przeładowanie 281
zasłonięta nazwa globalna - dostęp 280
zawieranie
~ obiektów a dziedziczenie klas 529-530
~ obiektu 703, 706, 720
zawieszający się program 168
zbiorcza inicjalizacja tablicy 132
zespolone liczby 399, 422-423
zmiana formatu operacji we/wy 614
zmienna 12
~ automatyczna 97
~ statyczna globalna 106
~ statyczna lokalna 98
znaki wypełniające 625
zręczny wskaźnik 471, 481

Ponieważ książka ta drukowana jest na arkuszu wydawniczym mieszczącym 16 stron, dlatego kilka ostatnich stron zwykle zostaje pustych. Postanowiliśmy więc zamieścić tu fragmenty listów do autora. Są one wypowiedziami o poprzednich wydaniach "Symfonii," a opisywane w nich usterki Symfonii staraliśmy się naprawić. Liczymy, że nam się to udało.

W niektórych listach postanowiliśmy cytowane nazwiska zastąpić gwiazdkami. Dobrze wychowani ludzie zrozumieją dlaczego nie wypadało nam inaczej.

Oficyna Kallimach

✉

Jestem w połowie drugiego tomu. Jestem naprawdę zadowolony z zakupu tej książki, gdyż wydaje mi się, że my Polacy mamy trochę inne podejście do różnych problemów. Co mam na myśli? Otóż w żadnej z angielskich książek o komputerach (a mam ich tu 53) nie znalazłem tak prostego i przejrzystego wytłumaczenia "problemów językowych". Myślę, że także poczucie humoru mamy nieco inne i być może dlatego Twoja książka przypadła mi do gustu. Dobrze wystarczy... nie chcę by Ci woda sodowa do głowy uderzyła (ale poważnie naprawdę fajnie mi się ją czyta).

✉

Mój kolega powiedział, że Symfonię odkłada zawsze na wieczór, bo wie, że przy tej książce nie zaśnie.

✉

Książka rewelacyjna. Najlepsza rzecz to pisownia fonetyczna.

✉

*W porównaniu z autorem ***, to książka jest naprawdę dobra. Nie jest pisana "przez prof. dla prof.".*

✉

Książka extra, tylko korekta lekko zawiodła!!! Proszę o coś podobnego o UNIX!!!

Pozdrowienia z zajęć z Unix-a!

laboratorium dydaktyczne (Politechnika Wrocławska)

✉

Doszłam do rozdziału 14 (konstruktory i destruktory) i jak dotąd to oprócz błędów literowych (korekty oczywiście), innych nie zauważyłam, ten o którym pisałam ostatnio to zauważył jeden z kolegów, może on napisze osobiście, nie wiem. Błąd dotyczył zdaje się konstruktorów obiektów tworzonych new i ponoć z książki wynikało, że czegoś nie wolno robić, a w rzeczywistości nawet należało, może źle zrozumiał? Przez ten tydzień nie udało mi się go spotkać, ale przysięgam, że jak tylko go dorwę, to albo wytłumaczy się dokładniej albo odszczeka kalumnie (żartuję oczywiście, aż taka groźna nie jestem). Napiszę jak najszybciej się da.

W międzyczasie drugi z kolegów (po przejrzeniu moich egzemplarzy książki) kupił też "Symfonię C++" i mówi że ma kłopoty z konstruktorem kopiującym, przypomniałam mu o adresie i prośbie o uwagi więc może też napisze...

No, z cudzymi uwagami koniec, więc:

❖) do przyszłego wtorku mam zrobić symulację sklepu z 4 kasami i kolejkami równomiernie rozstawionych klientów, z obiektami i czymś co nasi mili prowadzący określili jako konstruktor nawiasowy i bezparametrowy, poprzednio zaś kazali zadeklarować w klasie część funkcji jako otwarte (?), nie wspomnę już o tym jak często używają słowa inicjacja...

"Symfonię" czytam więc nie tylko przy herbatce (choć z początku to głównie tak), ale i przy komputerze, przy "odrabianiu lekcji" zaglądam do niej często! Trochę denerwujące jest to, że spis treści i indeks są w tomach 1 i 3, więc gdy chcę poczytać na nudnym wykładzie to muszę targać ze sobą trzy tomy. Dobrze, że nie są zbyt ciężkie...

Może mam słaby zmysł krytyczny, ale z książki nie wyrzuciłabym nic, a nawet jeszcze dodała (szczególnie Jasia z Małgosią), a ideę nie używania zbyt dużo trudnych wyrazów w jednym zdaniu to bym z chęcią rozpowszechniła...

❖) Grupę mamy świetną tylko przedmioty dzikie, czy naprawdę bez funkcji samodwoistych i postaci dysjunkcyjnych nie da się żyć????? I czemu tego nie można w ludzkim języku przekazać, po co to i z czym to jeść?

❖) Język polski w książce bez zarzutu!! Nie to, co nasi prowadzący....

❖) Komunikatywność (tak mówiła moja polonistka z L.O.) na 6!!!

Pozdrawiam i przepraszam za tak kiepski styl!!! Pisanie listów jest jednak też sztuka!!!!!!!

✉

Szanowny Panie!

*Nie przeczytałem jeszcze w całości Pana książki, jednak już jestem w stanie powiedzieć, iż jest to chyba najlepsza pozycja dotycząca C++ na polskim rynku. Dotychczas spotykałem prawie wyłącznie książki pana ***, który pisze swe "dzieła" w sposób : "patrzcie jaki ja jestem mądry!". "Symfonia C++" jest natomiast pozycją w stylu: "patrzcie jakie to proste". Zamieszczone przykłady są rozłożone na czynniki pierwsze i dokładnie omówione.To jest przejrzyste i przemawia do czytelnika. Nie jestem zbyt dobrym programistą, rzekłbym nawet że słabym, jednak wydaje mi się, iż po przeczytaniu "Symfonii" powinienem się podciągnąć.*

✉

Szanowny Panie Jurku!!!

Prawdę mówiąc po ukazaniu się Symfonii na półkach księgarskich - nie przyjrzałem się bliżej jej zawartości, sądząc, iż jest to kolejna książka, pisana w stylu, który Pan określił, jako: "popatrzcie, jaki jestem mądry"...

Dlatego też zostałem mile zaskoczony, gdy jeden z kolegów wyprowadził mnie z błędu - pożyczając mi "Symfonię..." na parę dni...

Jest to książka napisana naprawdę w dobrym stylu, którą przyjemnie się czyta - a co najważniejsze - zawiera istotne wiadomości, przystępnie wyjaśnione czytelnikowi.

Ponieważ książka ta drukowana jest na arkuszu wydawniczym mieszczącym 16 stron, dlatego kilka ostatnich stron zwykle zostaje pustych. Postanowiliśmy więc zamieścić tu fragmenty listów do autora. Są one wypowiedziami o poprzednich wydaniach "Symfonii," a opisywane w nich usterki Symfonii staraliśmy się naprawić. Liczymy, że nam się to udało.

W niektórych listach postanowiliśmy cytowane nazwiska zastąpić gwiazdkami. Dobrze wychowani ludzie zrozumieją dlaczego nie wypadało nam inaczej.

Oficyna Kallimach

✉

Jestem studentem Politechniki Poznańskiej. Przeczytałem Pana książkę "Symfonia C++", i muszę powiedzieć że jestem nią zachwycony. Na polskim rynku istnieje obfita ilość różnorakich pozycji na temat języka C++, lecz najbardziej przypadł mi do gustu Pana styl programowania.

✉

Ostatnio polecam studentom, których uczę programowania twoją książkę Symfonia C++. Kilka osób ją kupiło (nie wiem jak im się podoba). Z mojego punktu widzenia ta książka ma istotną wadę: Brak w niej zadań do samodzielnego rozwiązania. Czy planujesz może dodanie takich zadań w przyszłych wydaniach? Ja chwilowo biorę zadania z książki Kernighana The C Programming Language, myślę, że wiele z nich można wprost dołączyć do twojej książki (pozostaje problem praw autorskich).

✉

Symfonia jest napisana prostym, zrozumiałym językiem, co czyni ją niezwykle przystępną dla niemal każdego miłośnika techniki komputerowej. Ważniejsze fragmenty powtarzane są wielokrotnie, co początkowo - zwłaszcza przy łatwiejszych partiach wykładanego materiału - może wydać się nieco nużące, ale szybko okazuje się jedną z najmocniejszych stron opisywanej pozycji. Wymieniając jej zalety nie sposób też pominąć jeszcze jednej - Otóż czytając książkę i śledząc przytoczone fragmenty szybko daje się zauważyć, iż autor DOSKONALE wie o czym pisze, co wzbudza zaufanie u "uczniów-czytelników". Bez zarzutu jest również forma graficzna książki.

Hmmm, pora chyba przejść do wad, które nazwałbym raczej małymi potknięciami. Jednym z najczęstszych są wszechobecne literówki, szerzej - błędy wynikłe z niezbyt chyba starannej edycji tekstu. Pewien "pośpiech" - jak mniemam - "zaowocował" przestawionymi literkami, częstymi powtórzeniami tych samych wyrażeń, niedokończonymi zdaniami...

No cóż, wiem, iż takie błędy można "poprawić" sobie samemu, ale czasem niestety mogą one wręcz uniemożliwić pełne i jasne zrozumienie zasad rządzących piękną krainą C++.

✉

Książka jest niezła, może trochę przegadana, ale to nie umniejsza jej wartości. Prosty język to zaleta, a nie wada. Kursy programowania pisane przez osoby, które osobiście pisały duże projekty w danym języku są (na ogół) lepsze od wyrobów tzw. uczonych informatyków.

Z językiem programowania jest trochę tak, jak z językiem żywym, 99% ludzi chce po prostu mówić/programować nie wgłębiając się w gramatyczne/formalne niuanse tak podniecające dla uczonego lingwisty/informatyka.

[asystent prowadzący zajęcia z C++]

✉

Muszę przyznać, że Twój entuzjazm do C++ daje się odczuć i w pewnym stopniu oddziaływuje to na czytelniku. Właśnie wczoraj czytałem rozdział o przeładowywaniu operatorów. Muszę przyznać, że po przeczytaniu angielskich książek, byłem pod wrażeniem, że jest to trudne i nie widziałem dla tego żadnego zastosowania. Masz dar tłumaczenia i upraszczania trudnych pojęć. Nawet sam sobie zacząłem kombinować jak można jeszcze wykorzystać ten wspaniały pomysł. Pomyślałem sobie, że mogę napisać structure "linked lists" i przeładować + i - żeby mi np. odpowiadały procedurom: insert i delete. Chyba tylko przez przypadek czytałem też wczoraj o bibliotece Borland'a Turbo Vision, i coś mnie tchnęło, żeby sprawdzić jak oni napisali menu w tym pakiecie. :-) Tak Właśnie oni to użyli: przeładowali + operator i każde nowe menu można dodać poprzez napisanie np. stary item + nowy item. Niesamowita dogodność i ma zastosowanie w profesjonalnym pakiecie!!!

Co do Twojej książki to wcale się nie dziwię, że posługują się nią studenci. Jest ona bardzo dokładna i opisuje wszystkie elementy języka. Na pewno cieszy się popularnością studentów w Polsce, bo po porównaniu jej ze studenckimi skryptami jest "trochę" lżejsza w czytaniu.

Jestem na informatyce na uniwersytecie, koło którego pracuję. (Tutaj mała dygresja - mamy tutaj tych samych profesorów, którzy mają duży kontakt z University of Waterloo, który jest domem jednego z najlepszych kompilatorów do C++ do PC Watcom C++. Może kiedyś uda mi się tam dostać pracę).

Pozdrowienia z Kanady

✉

Podobno chciałbyś dopisać nowe rozdziały do Twojej już doskonałej książki. Właśnie próbuję nauczyć się Templates i nawet nie masz pojęcia jak mi brakuje Twojej spokojnej, przyjacielskiej porady, do jakiej Twoja książka mnie przyzwyczaiła. Jak widzisz popieram pomysł dopisania tych rozdziałów do książki całym sobą, i mam nadzieję, że nie są to tematy zbyt skomplikowane.

✉

W ostatnim mailu napisałeś, że nie planujesz wydania książki o programowaniu dla Windows. Jednak gorąco Cię namawiam, ponieważ już pierwszy rok (politbuda czy uniwerek) zaczyna programować w Windowsie. Innym argumentem jest to, iż na naszym rynku jest duży niedobór. Pojawiły się jedna, może dwie książki, które opisują to tak ogólnikowo, że nikt nie jest w stanie czegoś zrozumieć.

Ponieważ książka ta drukowana jest na arkuszu wydawniczym mieszczącym 16 stron, dlatego kilka ostatnich stron zwykle zostaje pustych. Postanowiliśmy więc zamieścić tu fragmenty listów do autora. Są one wypowiedziami o poprzednich wydaniach "Symfonii," a opisywane w nich usterki Symfonii staraliśmy się naprawić. Liczymy, że nam się to udało.

W niektórych listach postanowiliśmy cytowane nazwiska zastąpić gwiazdkami. Dobrze wychowani ludzie zrozumieją dlaczego nie wypadało nam inaczej.

Oficyna Kallimach

✉

Symfonia bardzo mi się podoba. Po pierwsze, niewiele jest książek, które są napisane lekkim, łatwym i przyjemnym językiem, i jeszcze do tego niosą w sobie solidny ładunek wiedzy. Może dla mnie, jako informatyka znającego C, była na początku trochę rozwlekła (ale wszak nie wszyscy C znać muszą), ale potraktowałam pierwszą część jako literaturę rozrywkową i było dobrze. Poważne zastrzeżenie mam do ilości przecinków (zbyt ich mało!), no, ale to są drobne niedociągnięcia. Drugą wadą to brak templates i exceptions, ale zdaje się, że będzie to uzupełnione.

Książka jest na 5+. Więcej takich...

✉

Niniejszym chciałbym wyrazić podziękowania za Pana książkę "Symfonia C++". Swoją przygodę z C++ i obiektowością właśnie od tej książki zacząłem. Teraz specjalizuje się w Object Oriented Analysis/Design. Moim podstawowym językiem programowania jest C++. Pomimo, że dosyć pewnie poruszam się w świecie obiektów, to jeśli chodzi o programowanie, Pana książka jest dla mnie cały czas użytecznym źródłem cennych informacji. Jak dotąd NIE SPOTKAŁEM lepszego podręcznika.

✉

"Symfonia C++" jest w naszej uczelni podręcznikiem podstawowym. Polecił mi ją mój promotor (jest on wspaniałym pedagogiem, a na dodatek maniakiem obiektowości i C++). Na dobre weszła do użytku (nie chcę się tu przechwalać) od momentu, gdy cały nasz (wówczas jeszcze czwarty) rok zaczął ją czytać. W tej chwili jest to standard wśród studentów. Wykładowcy, z wyjątkiem jednego, czytują ją ukradkiem jeśli w ogóle to robią. Po przestudiowaniu Symfonii bez trudu mogę sięgać do C++ - opis standardu - Stroustrupa. Symfonię nabyłem w dniu jej ukazania się w księgarni.

W Symfonii znalazłem kilka "literówek", osobiście nie zmieniał bym w niej niczego więcej. Najbardziej podoba mi się sposób, w jaki przekazuje Pan swoją wiedzę. Trudne poniekąd problemy obrazowane są w sposób PROSTY. Do tej pory spotkałem tylko jedną książkę, którą mógłbym porównać z Symfonia. Jest to "OOP" autorstwa P. Coad'a. Z niecierpliwością czekam na "templates" i "exceptions". Nieskromnie liczę na utrzymanie korespondencji z Panem w przyszłości.

✉

Właśnie wczoraj miałem oficjalną prezentację mojego programu dyplomowego w C++.

Teraz kończę pisać dokumentację i raport. Mój program, mam nadzieję, będzie dobrze służył w laboratorium materiałów kompozytowych. Analiza mikrograficzna to dość ciekawy problem. Tak prawdę mówiąc wciągnęło mnie to trochę.

Musiałem sam zaprojektować i zaimplementować cały system. Z perspektywy czasu wydaje mi się to niezbyt dużym przedsięwzięciem. W sumie jest to tylko program do oglądania obrazków i wykonywania na nich jakichś przekształceń. Nigdy jednak nie musiałem sam wykonywać etapów analiza/projektowanie/programowanie. Było to dla mnie duże doświadczenie.

Niektórzy zarzucali mi, że zbyt mało funkcji jest zaimplementowanych, że zbyt dużo czasu poświęcałem na dwa pierwsze etapy. Ja jednak jestem zdania, że lepiej napisać program łatwy do późniejszych modyfikacji i rozszerzalny, niż 'spaghetti' bez ładu i składu. Nie chciałem, żeby po moim wyjeździe następni programiści wieszali na mnie psy. Chyba się ze mną zgadzasz?

Być może minusem mojej prezentacji było to, że odbyła się nie po francusku, ale w języku angielskim. Czy zwróciłeś uwagę na lekką niechęć do tego języka wśród Francuzów (a może nie dali Ci tego odczuć).

W moim raporcie pozwoliłem sobie w bibliografii umieścić Symfonię jako jedną z najważniejszych pozycji, które pomogły mi w programowaniu. Potem byli Petzold i Coad. Wywołało to trochę zamieszania, bo przemilczałem B. Stroustrupa. Niestety Francuzi mają pecha, z tego co mi wiadomo nie ma Symfonii po francusku.

Uważam, że Symfonia powinna zostać przetłumaczona przynajmniej na angielski. Jest to podręcznik na tyle dobrze zrobiony, że jest tego wart. Muszę przyznać, że gdy go kupowałem miałem mieszane uczucia. Wydawało mi się, że kupuję kolejny opis słów kluczowych języka C++. Okazało się jednak, że ta książka uczy programowania, a co ważniejsze w łatwy sposób przybliża technikę OO.

Wielokrotnie rozmawiałem z Francuzami na temat programowania. Większość z nich posługuje się C++, jednak na temat OO niewiele mieli do powiedzenia. Czasem rzeczywiście jakoś intuicyjnie wyczuwają pewne sprawy, ale z reguły nie jest to wiedza świadoma. Mam tu oczywiście na myśli studentów informatyki. W większości książek, które czytali o C++ znajdowali jedynie opisy składni. To, co jest w Symfonii, stanowi pełny kurs języka wraz z praktycznym wykorzystaniem obiektowości. To bardzo istotne, żeby początkujący programistą mógł w pełni poczuć narzędzie i do czego może je zastosować.

Osobiście do przeczytania Symfonii zachęcił mnie fakt, że napisana została przez kogoś, kto C++ wykorzystuje w swojej pracy, a nie tylko prowadzi akademickie dysputy na temat programowania.

Wracając do tłumaczenia Symfonii na angielski na przykład, myślę że jest to zadanie dość karkołomne ze względów języko-

Ponieważ książka ta drukowana jest na arkuszu wydawniczym mieszczącym 16 stron, dlatego kilka ostatnich stron zwykle zostaje pustych. Postanowiliśmy więc zamieścić tu fragmenty listów do autora. Są one wypowiedziami o poprzednich wydaniach "Symfonii," a opisywane w nich usterki Symfonii staraliśmy się naprawić. Liczymy, że nam się to udało.

W niektórych listach postanowiliśmy cytowane nazwiska zastąpić gwiazdkami. Dobrze wychowani ludzie zrozumieją dlaczego nie wypadało nam inaczej.

Oficyna Kallimach

wych. Wydaje mi się, że ciężko będzie oddać klimat tej książki w obcym języku. Ale to już strata obcokrajowców. Niech uczą się polskiego.

Pozdrawiam serdecznie.

(student Ecole Catholique d'Arts et Metiers)

✉

Szanowny Panie Autorze,

1. Przeczytałem Pana podręcznik - Symfonia C++. Uważam, że jest świetny.

2. Chciałbym wykorzystać podręcznik do nauki języka C++ studentów Politechniki.

3. Czy i jaka byłaby szansa prosić Pana o udostępnienie za pomocą INTERNETu lub na dyskietce przykładów wydrukowanych w książce? W ten sposób studenci mogliby zaoszczędzić dużo czasu i poświęcić go na staranniejsze przestudiowanie Pana podręcznika.

✉

Jest Pan genialnym autorem najlepszego podręcznika do C++.

Jestem zainteresowany bieżącym wykorzystaniem podręcznika do prac studentów w ramach koła naukowego, a w następnym semestrze chcę przygotować wykład dot. programowania obiektowego w C++.

Dlatego jestem zainteresowany jak najszybszym uruchomieniem przykładów z książki. Część z nich udało mi się już uruchomić. Wszystkie działały bezpośrednio po przepisaniu, bez żadnych poprawek!!!

Będę szczęśliwy, jeżeli Będę mógł z Panem wymienić doświadczenia, a raczej nauczyć się od Pana zaawansowanych technik programowania obiektowego.

✉

PRZESYŁAMY GRATULACJE ZA NAPISANIE ŚWIETNEJ KSIĄŻKI SYMFONIA C++

STUDENCI Z GŁOGOWA

✉

"SYMFONIĘ C++" polecił nam wykładowca języka C++. Jednak z powodu nauki w systemie zaocznym korzystamy z niej głównie w domu. Niewątpliwie zaletą książki jest prosty, obrazowy,oparty na prostych przykładach sposób przekazu.

Uważamy, że jest to książka, od której trzeba zacząć, żeby zrozumieć C++.

Okładka jest ładna, kolorowa, przyciągająca wzrok (żeby tak jeszcze w twardej okładce). Książka jest trudno dostępna z powodu zbyt małego nakładu w stosunku do potrzeb.

✉

Naprawdę bardzo dobra książka. przeczytałem tchem od dechy do dechy (choć nie była to moja pierwsza książka o C++) i często używam jako reference do rzadziej używanych spraw. I wcale nie ma odpowiedzi na pytanie "How old is JB" ;-). Jedyna wada - spis treści tylko w pierwszym tomie.

✉

Dziękuję za odpowiedź. To wielki zaszczyt otrzymać list od samego Stwórcy Wielkiej Symfonii C++ na palce i klawiaturę.

Tak bez żadnego wazeliniarstwa - uważam ją za wspaniałą książkę do nauki programowania. Problemy w niej są wytłumaczone bardzo dokładnie i bez żadnych niedomówień, jak to się dzieje w przypadku innych książek. Chodzi mi o to , że jeśli są jakieś niuanse to są one wyeksponowane w przykładach.

Howgh. To tyle, co chciałem powiedzieć.

Kunta

P.S. Jeśli można to bardzo prosiłbym o informację na temat innych książek pańskiego autorstwa - jeśli temat byłby ciekawy to kupiłbym ją w ciemno.

✉

Mam jeszcze opinie subiektywne, dwie. Pierwsza to razi mnie określenie 'operator przeładowany'. Nie spotkałem takiego w polskiej literaturze i jakoś jestem przyzwyczajony do sformułowania 'operator przeciążony'.

Natomiast muszę gorąco pochwalić pomysł podawania wymowy angielskich terminów, mimo iż jestem programistą i używam niektórych z nich na co dzień, Musiałem zweryfikować kilka sposobów wymawiania, po uważnym przyjrzeniu się Twoim wskazówkom. Myślę, że obiekcje wyrażone co do tego we wstępie są nieuzasadnione.

Tyle pierwszych wrażeń, jeśli chcesz podzielę się dalszymi uwagami po przeczytaniu książki.

✉

Napisałem do pana dwa maile i nie dostałem odpowiedzi.
Po tym jak wszyscy w reklamują Symfonię zacząłem jej szukać w księgarniach. Przynajmniej lokalnie "Symfonia C++" nie jest uznawana za specjalnie chodliwy towar (lokalnie, w Toruniu, mam na myśli), sprowadzają tego 2-3 egzemplarze do jednej księgarni i podobno są kłopoty ze zbyciem (powtarzam co usłyszłem od księgarzy).

Tak czy inaczej bardzo chciałem tą książkę kupić. Chciałem, bo teraz już mi przeszło, tj. nie mam zamiaru tej książki szukać.

✉

Ponieważ książka ta drukowana jest na arkuszu wydawniczym mieszczącym 16 stron, dlatego kilka ostatnich stron zwykle zostaje pustych. Postanowiliśmy więc zamieścić tu fragmenty listów do autora. Są one wypowiedziami o poprzednich wydaniach "Symfonii," a opisywane w nich usterki Symfonii staraliśmy się naprawić. Liczymy, że nam się to udało.

W niektórych listach postanowiliśmy cytowane nazwiska zastąpić gwiazdkami. Dobrze wychowani ludzie zrozumieją dlaczego nie wypadało nam inaczej.

Oficyna Kallimach

Zdążyłem już prawie przeczytać całą Twoją książkę - najlepsze są przykłady. Cieszy mnie tym bardziej dlatego, że prowadzący wykład czasami (a może nawet często) z nich korzysta (zawsze jednak mówi, że ściągnął to z Symfonii). Pomimo, że wszystko jest opisane prosto, jak drut i nawet taki gnat, jak ja to kuma, to mam mały problemik. Nie z obiektami czy klasami, a tak trywialną rzeczą, jak typ wyliczeniowy. Jeżeli mój prowadzący nie będzie wiedział jak rozwikłać ten problemik, to chyba dam Ci znać. Jeszcze raz dziękuję za wspaniałą książkę.

✉

Sprawdzam, czy nie nawciskał Pan kitów z tym adresem poczty elektronicznej. Wydaje mi się, że ma Pan bardzo duże poczucie humoru, więc się Pan nie obrazi, że tak sobie poczynam troszkę bezczelnie - bym powiedziała. Zastanawiam się, czy Pan wie o jaką książkę chodzi. Wydaje mi się, że to nie jest tylko jedyna książka, którą udało się Panu z takim powodzeniem napisać.

Pozostawiam Pana w błogiej nieświadomości, o jaką książkę chodzi i co do płci autora tego listu (jestem dopiero początkującym internautą, więc nie wiem, co można się dowiedzieć tak ot sobie).

Już naprawdę kończąc powiem Panu jeszcze to, że z pewnością Pana książka może świadczyć o tym, że należy Pan do ludzi, od których wychodząc wychodzi się mówiąc: "Jakie to proste!", a nie "Jaki on mądry ".

Mam też kilka uwag krytycznych. Jeśli jest Pan tym zainteresowany to mogę się z nimi podzielić (raczej nie merytorycznych: literówki).

✉

...muszę powiedzieć, że jest to książka jakich mało. Dlatego że w sposób prosty mówi o rzeczach, które z początku są bardzo tajemnicze i niezrozumiałe. Ty potrafiłeś umiejętnie pogodzić trudny temat z łatwym, prostym i przyjemnym (ale nie - trywialnym) przedstawieniem.

Dużą zaletą tej książki jest to, że wprowadzasz analogie 'z życia wzięte', i przez to można wiele zrozumieć (podkreślam - **zrozumieć** *- to znaczy wiedzieć i umieć to zastosować) np. to z tą fotografią babci...*

Tak więc taka jest Twoja książka, ale jest w niej jeszcze coś, czego nie potrafię do końca dookreślić. A co to takiego, to jeszcze nie wiem.

✉

*W zeszłym semestrze nagle przypomniano sobie, że informatykom przydało by się C++. Mieliśmy wykłady z mgr ***. W literaturze którą podał nam na pierwszym wykładzie Symfonii nie było. Byłem na dwóch pierwszych wykładach zobaczyć co będzie mówione, ale mgr *** języka C++ chyba nie zna. Twierdził np. że struktura tym się różni od klasy, że nie może mieć metod(????!!!!!). Pisał przykłady, których żaden kompilator by nie skompilował i wygłaszał inne wspaniałe twierdzenia. Porzuciłem więc chodzenie na wykłady, ale zawsze wypytywałem co się tym razem działo. Po wykładzie o funkcjach wirtualnych kolega, który znał już C++ stwierdził, że teraz to on już sam nie wie, o co w funkcjach wirtualnych chodzi. W połowie semestru mgr *** jednak przypomniał sobie o Symfonii. Powiedział, że jest nienajgorsza.*

✉

Twoja książka jest niesamowita. Jeszcze nie czytałam takiej, jak ta - dziwna sprawa. Bo lubię czytać książki nie tylko o tematyce komputerowej, ale też beletrystycznej, psychologicznej i innej... Ale ta książka bije wszelkie rekordy. Jak zacznę ją czytać, to nie mogę się od niej oderwać. Po prostu. A poza tym to biorę aktywny udział w jej czytaniu - Myślę przy czytaniu tej książki, analizuję programy - na razie ciągle jeszcze na sucho. Ale uczę się, autentycznie się uczę.

Muszę jeszcze powiedzieć, że przykłady, które wybierasz na Twoje programy, mimo że w pewnym sensie standardowe, to jednak są bardzo interesujące (mówię to naprawdę szczerze). Z tego też powodu mogę powiedzieć, że czytanie Twojej książki jest to wielka przygoda. Rzadko kiedy udaje się nawiązać autorowi taki kontakt z czytelnikiem - Tobie się to udało na całej rozciągłości.

✉

Bardzo się cieszę, że mogę do pana się zwrócić poprzez internet. Chciałbym podziękować za Symfonię... jest to dla mnie odpowiednia pozycja i muszę stwierdzić, że jako osoba nie mająca do tej pory nic wspólnego z programowaniem w językach nowej generacji znajduję w niej to czego szukałem :) (tak przynajmniej mi się do tej pory wydaje)...

✉

Twoją Symfonię przeczytałem dawno temu, jej pierwsze wydanie! Bardzo mi się podobała, chociaż przez pierwszy tom wkurzałem się "czy facet musi bez przerwy przypominać w kółko to samo", ale w 2 i 3 tomie to pomogło. Jest ona napisana lopatologicznie i wiele rzeczy nie trzeba czytać po 10 razy żeby je zrozumieć. Ostatnio miałem przerwę w pisaniu w C dlatego zdecydowałem, że muszę mieć tę książkę WŁASNĄ!, ale brakuje mi pewnych rzeczy, no cóż nie można mieć wszystkiego w jednym :), chociaż niektórzy próbują robić 2w1 :)))

✉

Książka ta jest bardzo chwalona w konferencji o C++ w Fido. To wygląda tak, jakby wszyscy chcieli powiedzieć "wyrzućcie wszystkie książki do kosza, a kupcie Symfonię". Cała dyskusja zaczęła się od prośby ze strony któregoś z konferentów o

Ponieważ książka ta drukowana jest na arkuszu wydawniczym mieszczącym 16 stron, dlatego kilka ostatnich stron zwykle zostaje pustych. Postanowiliśmy więc zamieścić tu fragmenty listów do autora. Są one wypowiedziami o poprzednich wydaniach "Symfonii," a opisywane w nich usterki Symfonii staraliśmy się naprawić. Liczymy, że nam się to udało.

W niektórych listach postanowiliśmy cytowane nazwiska zastąpić gwiazdkami. Dobrze wychowani ludzie zrozumieją dlaczego nie wypadało nam inaczej.

Oficyna Kallimach

podanie tytułu książki do nauki C++. Ja sam polecałem Stroustrupa, ale szybko zrobiło się cicho na jej temat. Ludzie czytają Symfonię jak lekturę szkolną! I dobrze.

*Dawniej na półkach można było spotkać jedynie nędzne przekłady autorstwa pana ***. Teraz jest w czym wybierać, a poziom literatury jest na prawidłowym poziomie. Życzę powodzenia przy pisaniu kolejnych pozycji!*

✉

Symfonia jest po prostu genialna.

Zwracam się do Ciebie na ty, gdyż taki był 'rozkaz' we wstępie, który mimo ostrzeżeń przeczytałem :-)

Byłem na twojej stronie WWW i... szczęka mi opadła do podłogi - trochę już tych książek wydałeś :).

A propos: jesteś z wykształcenia fizykiem czy informatykiem?

Wydaje mi się że przydałoby Ci/nam się coś takiego co widziałem już niejednokrotnie na innych stronach. Jeśli sobie ktoś życzy to jego adres e-mail zostaje tam wpisany i automagicznie po naktualnieniu strony zostają rozesłane 'wici' w świat - dzięki temu zainteresowane osoby wiedzą kiedy np. wydasz nową książkę.

✉

Jestem doprawdy pełen podziwu dla Pana. Symfonia C++ jest najlepszym podręcznikiem do nauki języka, jaki kiedykolwiek czytałem. Znałem wcześniej kilka (Basic, Amos, E, Pascal, Prolog), ale nie zdarzyło mi się nigdy opanować podstaw języka programowania (składni, funkcji i instrukcji) nie pisząc ani jednego programu (!!!) Tak więc, dwa tygodnie temu, gdy zaczęły mi się zajęcia z grafiki komputerowej nie miałem pojęcia o C. Tydzień temu napisałem program, nad którym wykładowca stwierdził: "widzę, że jest pan zaawansowanym programistą...", a wczoraj, gdy zapytałem wykładowcę C++ jaka jest składnia listy inicjalizacyjnej konstruktora, on zapytał: "A co to takiego?"

Dziękuję za Symfonię (jestem w trakcie trzeciego tomu), jest Pan niesamowity!

✉

Jestem studentką drugiego roku informatyki Politechniki Wrocławskiej. Miałam już okazję korzystać z Twojej książki, ale był to egzemplarz wypożyczony z biblioteki. Ponieważ uważam ją za jeden z najlepszych podręczników do tego języka, bardzo chciałabym mieć ją na własność. Jeśli to jest możliwe z dedykacją. Proszę napisz, w jaki sposób da się to zrobić.

✉

Jako dodatek przesyłam mój pierwszy program (działa dobrze chyba tylko na pentium...). Zaznaczam, że jest to PIERWSZY mój program w C++ i tylko dwa razy zerkałem do podręcznika (operator podstawienia i konstruktor kopiujący)!

Jak udało Ci się napisać podręcznik, który tak łatwo wchodzi do głowy i zostaje w pamięci?!

✉

Bardzo jestem zadowolony z Twojej książki i polecam ją wszystkim swoim znajomym tu w USA.

✉

Pracuję na stanowisku adiunkta naukowo-badawczego na Uniwersytecie Łódzkim. Jestem matematykiem teoretykiem, ale coraz bardziej interesuję się informatyką. Po wstępnym zapoznaniu się z Turbo Pascalem, zabrałem się do "czytania" napisanej przez Pana książki "Symfonia C++". Jestem nią zachwycony. Ponieważ jestem osobą całkowicie niewidomą, więc nie mogę bezpośrednio samodzielnie korzystać z tekstu drukowanego. Do niedawna jedyną metodą pracy było korzystanie z pomocy lektora i robienie bardzo dokładnych notatek w brajlu. Jest to metoda skuteczna, ale bardzo czasochłonna(...).

✉

Kiedy będzie książka o template'ach? Chciałem tu skromnie zauważyć, że to ja głoszę wszem i wobec, że są dwie najlepsze książki pod względem fachowym i dydaktycznym: Teoria Pola (Landau i Lifszyc) i Symfonia C++ (Grębosz). Należy mi się wobec tego egzemplarz nowej as soon as possible.

(Wykładowca C++ na jednej z polskich politechnik).

✉

Przeczytałem już Twoją książkę do końca. Dużo się dowiedziałem, teraz po krótkiej kontemplacji zabiorę się za praktykę.

Generalnie, muszę Ci z radością pogratulować :-), książka jest fajna, daje się przeczytać (to jest niestety rzadkie ;-( ). Dobre jest w miarę szczegółowe komentowanie przykładów i to w formie nie komentarzy w tekście programów, które często bardziej zaciemniają, ale komentarza słownego później, odnoszącego się tylko do "bombek". Albo masz wrodzony talent, albo kosztowało Cię to masę pracy, trafne wyważenie tych przypisów. (...)

Życzę co najmniej 532 wydań i tłumaczeń na wszystkie języki nowożytne i ze cztery narzecza afrykańskie :-)

✉