Podstawy C++

Tomasz R. Werner

Warszawa 28 lutego 2023, 20:33

Podręcznik ten został przygotowany z myślą o wykładzie na temat języka C++ dla studentów znających już podstawy programowania w języku Java. Może być jednak, jak sądzę, przydatny również dla tych, którzy od C++ zaczynają naukę programowania. Materiał jest wystarczający na trzydziestogodzinny wykład semestralny; pewne fragmenty można pominąć lub zalecić studentom do samodzielnej nauki.

Spis treści

Spis tablic

— 1 —

Wprowadzenie

Materiał wykładu obejmuje podstawowe aspekty standardu języka C++. Pominięte zostały niektóre, bardziej techniczne szczegóły. Nie znajdzie też Czytelnik omówienia takich ważnych elementów programowania jak projektowanie interfejsów graficznych czy programowanie sieciowe – niestety te kwestie nie są na razie objęte standardem (choć w nowym standardzie, z roku 2011, pojawiło się już programowanie wielowątkowe czy wyrażenia regularne). Materiał zawarty w wykładzie powinien jednak stanowić wystarczającą podstawę do dalszej samodzielnej nauki tych i innych aspektów języka.

Wiele elementów języka omówionych jest we fragmentach tekstu poświęconych analizie podanych przykładów; stanowią one zatem bardzo ważną część wykładu i w żadnym przypadku nie powinny być przez Czytelnika pomijane!

C++ jest bardzo bogatym językiem programowania. Niestety jest też raczej językiem trudnym. Jest to, do pewnego stopnia przynajmniej, spowodowane założeniem zgodności C++ ze znacznie starszym i nieobiektowym językiem C. W zasadzie każdy prawidłowy program w C można skompilować za pomocą kompilatora C++ (oczywiście nie odwrotnie!). Od czasu do czasu będę więc wspominał o kompatybilności tych dwóch języków programowania.

Druga trudność, to hybrydowość C++: można stosować w nim zasady programowania obiektowego, ale można też pisać w nim całkiem złożone programy nie wiedząc co to są obiekty, dziedziczenie, hermetyzacja i inne tego rodzaju elementy programowania obiektowego. Warto też zwrócić uwagę, że w nowym standardzie pojawiło się sporo nowych konstrukcji, charakterystycznych dla języków funkcyjnych.

W roku 2011 zatwierdzony został nowy standard (uzupełniony w roku 2014, w tej chwili większość kompilatorów uwzględnia już standard z roku 2017) – wprowadza on bardzo wiele zmian zarówno w samym języku jak i, może przede wszystkim, w bibliotece standardowej. Zmian jest tak dużo, że omówienie ich wszystkich w tym wykładzie nie jest możliwe; niektóre jednak, które wydają się najważniejsze czy najbardziej użyteczne, będą przynajmniej pokrótce opisane.

1.1 Kompilatory

Język C++ należy do języków kompilowanych (tak jak Fortran, Ada, Pascal i wiele innych).

W odróżnieniu np. od Javy, wynikiem kompilacji nie jest niezależny od platformy kod bajtowy (ang. *byte-code*), ale **plik wykonywalny** (ang. *executable*), czyli taki, który zawiera kod programu w języku maszynowym odpowiednim dla platformy (czyli systemu operacyjnego i procesora naszego komputera). Tak więc dla uruchomienia programu nie jest konieczne specjalne środowisko uruchomieniowe (jak maszyna wir-

tualna Javy czy interpreter Pythona) — wykonanie odbywa się bezpośrednio pod kontrolą systemu operacyjnego, choć zwykle do jego działanie są konieczne pewne pliki biblioteczne (w wypadku platformy **.NET** wygląda to nieco inaczej). Ponieważ w czasie wykonania nie jest już dokonywana żadna dodatkowa translacja, programy w C/C++ są zwykle szybkie (programy w C są prawie tak szybkie jak programy w Fortranie, programy w C++ są zwykle, choć nie zawsze, nieco wolniejsze).

Tak naprawdę kompilacja jest tylko jednym z etapów procesu wytwarzania pliku wykonywalnego. Dla uproszczenia jednak nazywać tu będziemy kompilacją cały proces prowadzący od plików źródłowych do pliku wykonywalnego.

W rzeczywistości potrzebny jest jeszcze na przykład preprocesor i tzw. linker (zwany też konsolidatorem albo programem łączącym). W dużym skrócie, rolą linkera jest połączenie w jeden plik wykonywalny wielu plików otrzymanych w wyniku kompilacji poszczególnych plików źródłowych, których może być wiele i mogą być kompilowane osobno i w różnym czasie.

Zaletą tego podejścia jest duża szybkość wykonywania kodu zawartego w pliku wykonywalnym. Kompilacji oczywiście nie trzeba powtarzać przed każdym uruchomieniem programu: raz utworzony plik wykonywalny można zlecać systemowi operacyjnemu do wykonania, czyli **uruchamiać**, dowolną liczbę razy. Wadą języków w pełni kompilowanych, jak C++, jest, jak wspomnieliśmy, uzależnienie kodu wykonywalnego od platformy. Oczywiście sam tekst programu (źródło, ang. *source*) *jest* niezależny od platformy, jeśli trzymamy się w nim ściśle standardu języka C++ i nie stosujemy żadnych specyficznych dla danego kompilatora rozszerzeń.

Co dla nas z tego wszystkiego wynika to to, że potrzebujemy kompilatora (wraz z linkerem, preprocesorem, bibliotekami itd.). Dla tych, którzy nie mają zainstalowanego na swoim komputerze kompilatora C++ (albo mają, ale o tym nie wiedzą), podamy więc parę szczegółów dotyczących jego instalacji. Do jej sprawdzenia możemy potem użyć następującego programu:

P1: ***testInst.cpp*** Test instalacji

```
/*
 * Test instalacji. Program powinien wypisać nazwy
 * 4 języków programowania w porządku alfabetycznym.
 */
#include <vector>
#include <algorithm>
#include <iostream>
#include <string>
using namespace std;

int main() {
    vector<string> vs{"Python", "Haskell",
```

```
13                          "C++",     "Java"};
14     sort(vs.begin(),vs.end());
15     for (const auto& e : vs) cout << e << " ";
16     cout << endl;
17 }
```

którego uruchomienie powinno spowodować wypisanie na ekranie nazw czterech języków programowania w porządku alfabetycznym. Program może na razie być niezrozumiały, ale chodzi tu raczej o to by sprawdzić, czy Twoja instalacja C++ (kompilator, linker, biblioteki) jest prawidłowa. Spróbuj zapisać ten program w oddzielnym katalogu, skompilować go i uruchomić.

Przy okazji podkreślmy, że

dla każdego projektu w C++ tworzymy odrębny katalog

o umiejętnie dobranej nazwie i przemyślanej lokalizacji w strukturze katalogów. Nieprzestrzeganie tego zalecenia wcześniej czy później (raczej wcześniej) doprowadzi do powstania chaosu niemożliwego do opanowania.

1.1.1 Linux

W najprostszej sytuacji są użytkownicy Linuksa: nie potrzebują bowiem niczego ponad to, co już prawdopodobnie mają, nawet jeśli nie byli tego świadomi. Należy tylko zadbać podczas instalacji samego systemu, aby zaznaczyć do instalacji pakiet „dla programistów”; jeśli tego nie zrobiliśmy, to można zainstalować go później jednym poleceniem za pomocą instalatora właściwego dla danej dystrybucji (dotyczy to również komputerów Mac).

Jeśli w aktualnym katalogu mamy plik źródłowy zawierający program w C++, np. wspomniany wyżej plik ***testInst.cpp***, to komenda

```
g++ -o testInst testInst.cpp
```

powinna spowodować uruchomienie kompilatora i linkera (konsolidatora) i, jeśli nie zostały wykryte żadne błędy, pojawienie się na dysku, w tym samym katalogu, pliku ***testInst***, który właśnie jest plikiem wykonywalnym. W świecie Linuxa pliki wykonywalne tradycyjnie nie mają żadnego rozszerzenia, choć oczywiście nic nie stoi na przeszkodzie, abyśmy takie rozszerzenie, np. ***.exe***, dodali: nazwa pliku wykonywalnego będzie taka, jaką podamy po opcji ’`-o`’ powyższej komendy. Jeśli opcję tę w ogóle pominiemy, to przyjęta zostanie domyślna nazwa ***a.out***.

Jeśli w programie używamy nowych konstrukcji językowych (ze standardów 2011/2014), może okazać się konieczne dodanie opcji ’`-std=c++14`’; dobrze jest też dodać opcje które wymuszą sprawdzanie zgodności ze standardem – na przykład:

```
g++ -o testInst -std=c++14 -pedantic-errors -Wall testInst.cpp
```

Program zawarty w pliku wykonywalnym uruchamiamy pisząc po prostu jego nazwę poprzedzoną znakami '`./`', a więc w naszym przypadku '`./testInst`', i wciskając klawisz `Enter`. Zatem przebieg takiej sesji mógłby być następujący:

```
cpp> g++ -o testInst -pedantic-errors -Wall testInst.cpp
cpp> ./testInst
C++ Haskell Java Python
```

Jeśli nasz program zapisany jest w wielu plikach, to podajemy je wszystkie; można używać znaków uniwersalnych, czyli tzw. dżokerów (ang. *wild cards*), takich jak np. gwiazdki

```
cpp> g++ -o testInst *.cpp
```

Plik wykonywalny powstaje, oczywiście, jeden. Dodatkowe informacje i opis wszystkich opcji kompilatora i linkera dostępne są, jak zwykle, na stronie pomocy *man* (komenda `man g++`) lub poprzez program ***info*** (`info g++`). W wielu edytorach można kompilację i uruchamianie programu przypisać do mnemoników klawiszowych; istnieją też graficzne programy bardzo ułatwiające pisanie programów w C++ (jak ***Eclipse***, ***Anjuta***, ***Geany***, ***KDevelop***, ***CodeWarrior***, ***Code::Blocks*** — można je łatwo znaleźć w internecie). Większość dobrych edytorów tekstu zapewnia podkreślanie składni, zwijanie i rozwijanie struktur, automatyczne wcinanie i inne udogodnienia do redagowania plików źródłowych C/C++.

1.1.2 Windows

Visual C++ i Studio .NET

Dobry kompilator C++ wchodzi w skład pakietu ***Visual Studio*** firmy Microsoft. Jest on wyposażony w bogaty interfejs graficzny, funkcje pomocy, debugger itd. Istnieje wersja darmowa, ***Visual Studio Express***, w zupełności wystarczająca do nauki programowania. Osobom posiadającym to narzędzie można tylko przypomnieć, aby zawsze przed przystąpieniem do pisania programu zadbali o utworzenie najpierw tzw. projektu, w osobnym katalogu, przeznaczonym tylko na pliki wchodzące w skład tego projektu. Tworząc nowy projekt zadbać trzeba o jego właściwy typ, w naszym przypadku będzie to '`Empty project`'; inaczej mogą pojawić się niewiele wyjaśniające komunikaty o błędach podczas kompilacji. Uruchamiać kompilator można zarówno korzystając z interfejsu graficznego jak i bezpośrednio z linii poleceń wywołując program ***cl*** (co jest skrótem od *compile and link*). Kompilator ma wiele opcji, z których dobrze będzie wybrać '`-Wall -Za -GR -EHsc`', które, między innymi, włączą komunikaty o błędach, a za to wyłączą niestandardowe rozszerzenia języka.

1.2 Literatura

Literatura dotycząca języka C++ jest bardzo bogata. Spośród wielu pozycji dostępnych na rynku polskim na pewno godne polecenia są książki:

C++ Primer *Stanley B. Lippman, Josée Lajoie, Barbara E. Moo*

The C++ Programming Language *Bjarne Stroustrup*

The C++ Standard Library, 2nd Edition *N.M. Josuttis*

Algorithms in C++ *R. Sedgewick*

Podobny materiał można oczywiście znaleźć w wielu innych książkach na temat języka C++.

Niewyczerpanym źródłem informacji o C/C++ (i wszelkich innych innych językach programowania), przykładów, specyfikacji, omówień, wykładów i wszelkich innych przydatnych materiałów jest oczywiście Internet.

— 2 —

Zaczynamy

W rozdziale tym przeanalizujemy strukturę najprostszych programów w C++, ze szczególnym uwzględnieniem podstawowych operacji wejścia i wyjścia, co pozwoli nam rozpocząć samodzielne pisanie programów w C++. Omówimy tu tylko podstawy — generalnie operacje wejścia/wyjścia są bowiem mocno nietrywialne...

PODROZDZIAŁY:

2.1 Najprostsze programy

Przejdźmy zatem do omawiania samego języka C++; jak zwykle zaczniemy od od najprostszego ***Hello, World***.

Znający Javę (czy PHP) z łatwością zauważą, że wywodzi się ona, przynajmniej jeśli chodzi o składnię, właśnie od C/C++. Java nie jest tu zresztą wyjątkiem: wiele innych języków nawiązuje swą składnią do C/C++. Tak więc znajomość C/C++ okaże się bardzo przydatna przy nauce również innych współczesnych języków. W C/C++ występuje też sporo konstrukcji specyficznych dla tego języka. Dotyczą one, między innymi, operacji wejścia/wyjścia. Zauważmy tu, że operacje WE/WY są z kolei różne dla C i C++. Posługiwać się będziemy raczej tymi zdefiniowanymi w C++, choć znajomość wersji z języka C jest bardzo przydatna, choćby przy czytaniu istniejących kodów.

Pliki źródłowe zawierające kod C++ mają tradycyjnie rozszerzenie '***.cpp***', ale spotyka się też rozszerzenia '***.C***' i inne. Pliki z kodem w czystym C mają zwyczajowo rozszerzenie '***.c***'. Specjalny rodzaj plików, tzw. pliki nagłówkowe, mają w C i C++ rozszerzenie '***.h***' lub, w C++, w ogóle nie mają rozszerzenia.

Rozpatrzmy zatem plik ***helloWorld.cpp*** zawierający zasłużony program ***Hello, World*** (B. Kernighan, 1973) w języku C++:

P2: ***helloWorld.cpp*** *Hello, World* w C++

```
1 #include <iostream>                    #\raisebox{0.5pt}{\textcircled{\footnot
2 using namespace std;                   ②
3
```

```
4 /*
5     Naukę każdego języka programowania
6     zaczynamy zawsze od Hello, World!!!!
7  */
8
9 int main() { // Komentarze jak w Javie
10     cout << "Hello, World!" << endl;     ③
11 }
```

Jak łatwo się domyślić, uruchomienie tego programu powinno spowodować wypisanie na ekranie słów 'Hello, World!'.

Znający Javę zauważą, że struktura tego programu jest różna od analogicznego programu w tym języku. Pierwsza rzucająca się w oczy różnica to fakt, że nie występują tu w ogóle klasy. W szczególności, funkcja **main** *nie jest zawarta w żadnej klasie: jest funkcją* globalną.

Ogólnie, program w C++ składa się z jednego lub kilku (zapisanych w osobnych plikach) *modułów*. Każdy moduł może zawierać dyrektywy preprocesora (jeśli są potrzebne), deklaracje i/lub definicje zmiennych i funkcji oraz, oczywiście, komentarze. Dyrektywy preprocesora poprzedzone są znakiem '#', jak dyrektywa `#include <iostream>` w powyższym kodzie, i *nie* są przekazywane kompilatorowi — służą jedynie do wstępnego przetworzenia *tekstu* programu. Zadanie to wykonuje wspomniany już **preprocesor**. Żadnej linii rozpoczynającej się od znaku '#' kompilator w ogóle nie zobaczy!

Dokładnie jeden moduł musi zawierać funkcję o nazwie **main**. Wykonanie programu polega zasadniczo na wykonaniu tej właśnie funkcji. Legalnym programem jest więc np.

```
int main() {
    return 0;
}
```

lub nawet

```
int main(){}
```

Zajmijmy się jednak programem z pliku ***helloWorld.cpp*** (str. 7). Na jego przykładzie przyjrzymy się podstawowym elementom programu w C++:

`#include...` (①) Włączenie do programu zawartości pliku ***iostream*** (plik włączany w ten sposób nazywamy **plikiem nagłówkowym**, ang. *header file*). Dzięki temu w programie dostępne będą definicje (lub tylko deklaracje – więcej o tym powiemy później) narzędzia (w postaci najrozmaitszych klas, funkcji, stałych itd.), służące do wykonywania operacji wejścia/wyjścia (w skrócie, *operacje we/wy*), a więc np. wczytywania z konsoli i wypisywania na ekranie danych. Zauważmy, że instrukcja `#include` powoduje rzeczywiste włączenie pliku: równie dobrze moglibyśmy w tym miejscu wpisać jego treść bezpośrednio do naszego programu. Jest to więc zupełnie co innego niż instrukcja **import** w Javie, gdzie

chodzi raczej o rozszerzenie przeszukiwanej przestrzeni nazw.
Sam plik ***iostream*** znajduje się w znanym kompilatorowi katalogu: użycie nawiasów kątowych (`<...>`) oznacza właśnie, że nie jest to nasz własny plik, ale plik ze znanego kompilatorowi specjalnego katalogu bibliotecznego, dostarczonego wraz z kompilatorem przez producenta. Gdyby chodziło o dołączenie naszego własnego pliku, co też jest możliwe, użylibyśmy podobnej instrukcji, ale zamiast nawiasów kątowych umieścilibyśmy cudzysłowy. Tak więc `#include <bib.h>` włącza standardowy plik nagłówkowy o nazwie ***bib.h*** (dostarczany zwykle wraz z kompilatorem), natomiast podobna dyrektywa `#include "bib.h"` dołączyłaby plik nagłówkowy ***bib.h*** z katalogu bieżącego, a tylko jeśli w katalogu tym takiego pliku by nie było, poszukiwany byłby w katalogu standardowym. Zauważmy, że instrukcja `#include` *nie* jest przeznaczona dla kompilatora. Włączenie pliku wykonywane jest przez preprocesor, który zajmuje się wyłącznie przetwarzaniem *tekstu* naszego programu przed jego właściwą kompilacją — to, co zobaczy kompilator, to tekst naszego programu przetworzony przez preprocesor. Inne przydatne instrukcje (dyrektywy) preprocesora poznamy w rozdziale 3.

`using namespace std;` (②) Linia ta oznacza, że nazwy (klas, obiektów, funkcji) z przestrzeni nazw std mają być włączone (importowane) do bieżącej (domyślnej) przestrzeni nazw. To właśnie w przestrzeni std zadeklarowane są nazwy obiektów dostarczone przez dyrektywę preprocesora `#include <iostream>`: jak widać, samo dołączenie pliku ***iostream*** nie wystarcza — obiekty tam zdefiniowane są „zamknięte” w przestrzeni nazw std.
Dyrektywy **using** moglibyśmy tu nie zastosować, ale wtedy musielibyśmy pisać na przykład `std::cout`, `std::endl` zamiast po prostu `cout` i `endl`.
Innym rozwiązaniem, bardziej polecanym, byłoby użycie tzw. deklaracji użycia, za pomocą których możemy zaimportować nie wszystkie nazwy z przestrzeni std, ale tylko te, których potrzebujemy; na przykład **`using`** `std::cout;`.
O przestrzeniach nazw powiemy więcej w rozdz. 23.2 na stronie 510.

Komentarze (linie 4-7 i 9) Komentarze mogą być wieloliniowe, jeśli ograniczone są dwuznakami '`/*`' i '`*/`', lub jednoliniowe: od dwuznaku '`//`' włącznie do końca bieżącej linii. Komentarzy w pierwszej z tych form nie można zagnieżdżać (choć niektóre kompilatory na to pozwalają). Komentarze zostaną z tekstu programu wycięte (i zastąpione jednym znakiem odstępu) już na wstępnym etapie przetwarzania i nie mają żadnego znaczenia na dalszych etapach kompilacji.

Funkcja main (linie 9-11) Funkcja **main** musi zwracać („obliczać”) wartość typu **int** (czyli liczbę całkowitą), co zaznaczamy pisząc nazwę typu **int** *przed* nazwą funkcji. Zauważmy, że **main** jest *zawsze* funkcją *globalną*, tzn. nie jest i nie może być zanurzona w żadnej klasie (jak funkcja o tej samej nazwie i podobnym przeznaczeniu w Javie). Wartość typu **int** zwracana jest do systemu i zwyczajowo powinna wynosić 0, jeśli program kończy się pomyślnie. Niezerowa wartość zwrócona oznacza zwykle niepowodzenie, którego naturę można zakodować tą wartością i wykorzystać na przykład w skrypcie powłoki, z którego uruchamialiśmy nasz program. Jeśli zwracamy z funkcji **main** wartość niezerową, to powinna to być wartość dodatnia mniejsza od 256 (system może traktować w specjalny

sposób wartości ujemne). Normalnie, jeśli funkcja deklaruje, że zwraca wartość (czyli jest funkcją rezultatową), to *musi* to zrobić (za pomocą instrukcji **return**). Pod tym względem **main** jest wyjątkiem: jeśli nie ma w treści instrukcji **return**, kompilator doda ją sam.
Funkcja **main** pełni rolę punktu wejścia do programu; wyjście sterowania z tej funkcji powoduje zakończenie programu, choć, jak się przekonamy, nie natychmiastowe — najpierw „zwijany" jest stos (nie mówimy tu o programach wielowątkowych, które zachowują się nieco inaczej).
Funkcja **main** może mieć parametry, poprzez które system przekazuje do programu argumenty wywołania. W C++ pierwszym z tych argumentów, o numerze zero, jest zawsze nazwa samego wywoływanego programu. Przekazywanie argumentów wywołania wyjaśnimy bliżej niebawem. Jeśli nie zamierzamy korzystać z argumentów wywołania, to funkcję **main** można zadeklarować z pustą listą parametrów (ale nawiasy są wymagane). Można też jawnie wskazać, że funkcja jest bezparametrowa poprzez użycie słowa kluczowego **void**: '`int main(void)...`'.

Blokowanie (linie 9-11) Ujęcie kilku instrukcji w nawiasy klamrowe ('{' and '}') powoduje, że cały ten blok może być traktowany jako jedna instrukcja wszędzie tam, gdzie składnia języka wymaga pojedynczej instrukcji, a chcielibyśmy umieścić ich tam wiele (nazywamy to **instrukcją złożoną**). Blokowanie instrukcji ma też wpływ na widoczność i zasięg zmiennych, o czym powiemy w dalszym ciągu.
Ciało (treść) funkcji ma formę instrukcji złożonej, a zatem ujęte jest w nawiasy klamrowe i następuje po nagłówku funkcji określającym, między innymi, typ wartości zwracanej oraz liczbę i typ parametrów funkcji.

Instrukcje (③) Jest to jedyna instrukcja tego programu. Jak widzimy, instrukcje kończą się średnikiem. Format zapisu programu jest „wolny"; oznacza to, że dowolna niepusta sekwencja białych znaków, włączając znak nowej linii, jest równoważna jednemu odstępowi. Pozwala to na pewną swobodę w zapisie programu: ze swobody tej należy korzystać, aby pisany kod był jak najbardziej czytelny (oczywiście żadnych spacji nie może być wewnątrz słów kluczowych i nazw).
Ta konkretna linia powoduje wyprowadzenie na ekran napisu podanego w cudzysłowach. Wykorzystany jest tu mechanizm wyprowadzania danych właściwy dla C++ (a nieobecny w C). Właśnie po to, by móc użyć tego mechanizmu, musieliśmy w linii pierwszej dołączyć plik nagłówkowy ***iostream***.
Operacje wejścia i wyjścia, a więc wprowadzania danych, na przykład z klawiatury, i ich wyprowadzania, na przykład na ekran komputera, to temat długi i dość zagmatwany (zresztą nie tylko w C/C++). Zajmiemy się nim w swoim czasie (patrz rozdz. 16 na str. 323) bardziej szczegółowo, a poniżej zamieścimy parę wstępnych informacji na ten temat, aby móc używać podstawowych operacji we/wy w naszych programach.

Zauważmy, że w C++, jako języku w zasadzie proceduralnym, instrukcje wykonywane są w takiej kolejności w jakiej pojawiają się w programie (choć, jak się przekonamy, powtórzenia i skoki są możliwe).

Zanim przejdziemy do bardziej szczegółowego omówienia tych i innych elementów programu w C++, kilka uwag wstępnych na temat wprowadzania i wyprowadzania danych.

2.2 Proste operacje We/Wy

Po dołączeniu do naszego programu pliku nagłówkowego ***iostream*** zdefiniowane są (lub tylko zadeklarowane, ale to na razie nie jest dla nas istotne), między innymi, identyfikatory cout i cin. Są to nazwy obiektów reprezentujących standardowe strumienie: **wejściowy** (ang. *standard input stream, stdin*) i **wyjściowy** (ang. *standard output stream, stdout*).

2.2.1 Wyprowadzanie danych

Obiekt cout, jak powiedzieliśmy, reprezentuje standardowy strumień wyjściowy. Jest to predefiniowany obiekt klasy **ostream**, której definicja jest widoczna dzięki dołączeniu pliku nagłówkowego ***iostream***. Strumień wyjściowy to strumień danych wyprowadzanych z programu do „świata zewnętrznego", który to świat jest w tym wypadku tożsamy z ekranem komputera. Skrót cout należy rozumieć jako *console output* i wymawiać *si-aut* (porównując z Javą, najbliższym odpowiednikiem cout jest obiekt System.out). Ponieważ cout jest obiektem, można na jego rzecz wywoływać metody (funkcje) zdefiniowane w jego klasie. Metody te poznamy w rozdz. 16 na str. 323. Na razie nie musimy ich znać, aby zacząć obiektu cout używać: dla wygody programisty został bowiem dostarczony mechanizm **wstawiania do strumienia** za pomocą operatora '<<' (dwa znaki mniejszości, bez odstępu pomiędzy nimi). Konstrukcja jest taka jak w linii ③ programu z pliku ***helloWorld.cpp*** (str. 7). Piszemy więc

```
cout << "Hello, World" << endl;
```

aby wstawić napis, podany tu dosłownie jako literał, do strumienia „płynącego" na ekran, reprezentowany tu przez cout. Następnie, po kolejnej parze znaków '<<', do tego samego strumienia wstawiamy coś, czego identyfikatorem jest endl. Ta nazwa nie jest ujęta w cudzysłowy, bo nie chcemy zobaczyć na ekranie napisu "endl". Jest to tzw. manipulator (zdefiniowany również dzięki dołączeniu pliku nagłówkowego ***iostream***). Jego wstawienie do strumienia wyjściowego powoduje dodanie znaku nowej linii (i opróżnienie bufora). Gdybyśmy endl nie dodali, to znak nowej linii nie byłby wyprowadzony i następna operacja pisania na ekran kontynuowana byłaby w tym samym wierszu. Podobny efekt moglibyśmy osiągnąć dodając znak nowej linii do samego napisu: '"Hello, World\n"'. Dwuznak '\n' pojawiający się wewnątrz napisu oznacza właśnie znak nowej linii. Efekt byłby podobny, ale nie identyczny: wstawienie manipulatora endl powoduje dodatkowo natychmiastowe opróżnienie bufora wyjściowego i wyprowadzenie napisu na ekran. Bez niego natomiast napis nie musi być wyprowadzony natychmiast: system może zaczekać, aż w buforze wyjściowym uzbiera się więcej danych i wyprowadzić je dopiero wtedy łącznie.

W tym przykładzie wstawiliśmy do strumienia dwa elementy (obiekty): napis i manipulator. Moglibyśmy wstawić ich więcej. Zauważmy jednak, że wstawianie danych

do strumienia wyjściowego odbywa się pojedynczo: każda dana musi być poprzedzona znakami '<<'. A zatem np. *nie* można oddzielać poszczególnych elementów przecinkami

```
cout << "Hello, World" , endl; // NIE!!!
```

Prawidłowa jest natomiast konstrukcja

```
cout << "Pierwsza linia,"    << endl
     << "druga linia,"       << endl
     << "i w koncu trzecia." << endl;
```

która jest, zauważmy, jedną instrukcją, wyprowadzającą na ekran trzy napisy i trzy znaki końca linii.

Znający Javę pamiętają, że jest tam dozwolona konstrukcja

```
int k = 7;
double x = 8.6;
System.out.println(k);
System.out.println(x);
```

Te instrukcje spowodują wypisanie na ekran liczb 7 i 8.6, mimo że typ argumentu jest w obu przypadkach inny, a w żadnym nie jest odniesieniem do obiektu klasy **String**. Dzieje się tak dzięki przeciążeniu metody **println**. Podobny mechanizm działa w C++ dla operatora '<<'. Wstawiać do strumienia możemy nie tylko napisy, ale zmienne typów wbudowanych lub innych, dla których działanie operatora '<<' zostało zdefiniowane, jak np. obiekty klasy **string** w przykładzie poniżej:

P3: *jan.cpp* Operator '≪'

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    double waga = 76.5;
    int  wzrost = 182;
    string imie = "Jan";
    cout << imie   << " wazy " << waga << " kg i ma "
         << wzrost << " cm wzrostu."  << endl;
}
```

Programik powyższy wypisuje, jak łatwo się domyślić, napis

```
Jan wazy 76.5 kg i ma 182 cm wzrostu.
```

Klasa **string**, użyta w tym przykładzie, zostanie omówiona w rozdz. 17.2 na stronie 366. Inne nowe elementy tego przykładu omówimy bardziej szczegółowo za chwilę.

2.2.2 Wprowadzanie danych

Wprowadzanie danych odbywa się za pomocą podobnego mechanizmu. Tym razem obiektem reprezentującym strumień informacji wchodzących do programu ze świata zewnętrznego jest obiekt cin z klasy **istream**, dostępnej dzięki dołączeniu pliku ***iostream***. Źródłem tej informacji jest domyślnie klawiatura komputera. Nazwa cin pochodzi od *console input* (i wymawiać ją należy *si-in*). Jak cout, również cin jest obiektem, a zatem można na jego rzecz wywoływać metody zdefiniowane w jego klasie. Na razie jednak będziemy korzystać głównie z mechanizmu **wyjmowania ze strumienia** za pomocą operatora '>>' (dwa znaki większości, bez żadnego odstępu pomiędzy nimi).

P4: ***czyt.cpp*** Operator '≫'

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 int main() {
6     string imie;
7     int wzrost;
8     double waga;
9     cout << "Podaj imie, wzrost i wage: ";
10    cin  >> imie >> wzrost >> waga;
11    cout << imie << ", masz " << wzrost  << " cm wzrostu "
12         << "i wazysz " << waga << " kg" << endl;
13 }
```

Wykonanie programu miało następujący przebieg:

```
cpp> ./czyt
Podaj imie, wzrost i wage: Tomasz 182 76.5
Tomasz, masz 182 cm wzrostu i wazysz 76.5 kg
cpp>
```

Zauważmy tu następujące fakty:

- Kierunek „strzałek" wskazuje na kierunek przepływu informacji, tak więc dla cout informacja płynie w lewo, a więc *do* cout, czyli w kierunku ekranu; dla cin płynie natomiast *od* cin, czyli z klawiatury.
- Każdy obiekt wyjmowany jest ze strumienia osobno; *nie* można ich np. po prostu oddzielać przecinkami.
- Wartości danych wpisywane z klawiatury muszą zgadzać się co do typu z tym, czego oczekuje program: w powyższym programie oczekiwane są dane typu **string** (napis), **int** (liczba całkowita) i **double** (liczba rzeczywista, być może z nieznikającą częścią ułamkową) i w takiej kolejności należy je wpisać. Jeśli

np. spróbujemy drugą daną (wzrost) wpisać w postaci liczby z kropką dziesiętną, to wczytywanie zakończy się błędem (którego możemy nie zauważyć, jeśli sami tego jakoś nie sprawdzimy!).

- Poszczególne elementy danych wejściowych wpisujemy oddzielając je dowolnie długą niepustą sekwencją białych znaków (znaki odstępu – SP, tabulacji – HT, nowej linii – LF, powrotu karetki – CR). Białe znaki przed pierwszą daną są ignorowane, każda następna niepusta sekwencja białych znaków jest traktowana jako separator oddzielający dane. W szczególności wynika stąd, że tą metodą nie da się wczytać napisów, które zawierają białe znaki, np. odstęp. Gdybyśmy, na przykład, utworzyli zmienną typu **string** o nazwie imieNazwisko i usiłowali wczytać jej wartość w ten sposób:
  ```
  cin >> imieNazwisko;
  ```
  to po wpisaniu na klawiaturze np. '`Jan Kowalski`' wartością zmiennej byłby napis '`Jan`'; napis '`Kowalski`' pozostałby w buforze klawiatury nie „wyczytany" i zostałby wczytany przez następną instrukcję wejścia (powodując najprawdopodobniej błąd).

Operacje czytania i pisania, a więc operacje wejścia i wyjścia, należą do najbardziej zawiłych prawie w każdym języku programowania. W szczególności dotyczy to czytania z klawiatury, ze względu na nieprzewidywalność zachowań użytkowników...
Szczegóły odłożymy do rozdz. 16 na stronie 323. W szczególności, w podrozdz. 16.7.2 na stronie 351 pokażemy jak wygodnie przeczytać (lub zapisać) plik tekstowy.

2.3 Funkcje

Funkcje są podstawowym elementem programów (nie tylko w C/C++). Więcej o nich powiemy w rozdz. 11, str. 155. Tu podamy tylko bardzo skrótowe wprowadzenie, by móc ich używać w dalszych przykładach.

Funkcje pełnią w programach rolę podobną do funkcji matematycznych. Definicja funkcji, jak w matematyce, jest rodzajem przepisu opisującego, jak na podstawie danych wejściowych obliczyć pewną wartość. Sama definicja nie powoduje żadnych obliczeń: po zdefiniowaniu można jednak funkcji użyć, zwykle wielokrotnie, dla różnych konkretnych wartości danych wejściowych (argumentów) — dla każdych takich konkretnych danych zostanie wtedy zastosowany przepis podany w definicji i obliczona wartość wyniku. Na przykład definicja funkcji $f(a, b, x) = ax + b$ oznacza, że za każdym razem gdy użyjemy tej funkcji, będziemy musieli podać trzy liczby jako argumenty, a funkcja zwróci wynik równy iloczynowi pierwszej i trzeciej z tych liczb zwiększonemu o drugą z nich. Zauważmy, że gdy tej funkcji używamy (wywołujemy ją), to piszemy na przykład $f(3, 5, w)$ — taki zapis oznacza *zastosuj przepis podstawiając wszędzie 3 za a, 5 za b, a wartość symbolu w za x*. Same nazwy a, b i x w wywołaniu funkcji nie pojawiają się. [To jest cecha C/C++; są języki, w których istnieje możliwość użycia nazw parametrów formalnych funkcji — jest to na przykład bardzo użyteczne w językach takich jak Python, Ada czy Fortran 90/95].

Podobnie jest w programowaniu. Rozważmy następujący program:

P5: *funk.cpp* Funkcja

```
1 #include <iostream>
2 using namespace std;
3
4 double linear(double a, double b, double x) {   ①
5     return a*x + b;
6 }
7
8 int main() {
9     double c = 2, z = 3;
10    double result = linear(c,5,z);                ②
11    cout << "Result = " << result << endl;
12 }
```

Definicja funkcji **linear** zaczynająca się w linii ① mówi tyle:

- definiujemy funkcję o nazwie **linear**;
- funkcja ta będzie potrzebować trzech liczb typu **double** jako danych wejściowych (argumentów). Typ **double** opisuje w przybliżeniu liczby rzeczywiste znane z matematyki — szerzej o tym powiemy w rozdz. 4, str. 27. W definicji funkcji argumenty te będą występować pod nazwami a, b i x. Jak widzimy, podobnie jak w matematyce, listę parametrów funkcji piszemy w nawiasach za nazwą tej funkcji;
- typem wyniku funkcji będzie liczba rzeczywista (**double**) — tę informację podajemy *przed* nazwą funkcji;
- sama definicja, za nazwą i listą parametrów, ujęta jest w nawiasy klamrowe. W naszym przypadku definicja składa się z jednej linii, ale mogłoby być ich więcej;
- instrukcja **return** mówi, że działania funkcji tu się kończy, a jej wynikiem jest wartość wyrażenia, które pojawia się za słowem **return**.

W linii ② naszego przykładu wywołujemy funkcję **linear**. Jak widzimy, „posyłamy" do funkcji wartość zmiennej c (czyli 2) jako pierwszy argument, wartość 5 jako drugi, a wartość z (czyli 3) jako trzeci. Te trzy wartości zostaną podstawione za a, b i x podczas wykonania funkcji; zwrócony przez funkcję wynik (w naszym przypadku liczba 11) będzie wartością wyrażenia '`linear(c,5,z)`' i zostanie przypisany do zmiennej o nazwie result, która następnie jest wypisywana.

2.4 Argumenty wywołania

Jak wspomnieliśmy, do programu można przekazać dane za pomocą argumentów wywołania. Opiszemy to pokrótce teraz, aby Czytelnik mógł używać tego mechanizmu w pisanych przez siebie programach. Pełne jego zrozumienie możliwe będzie jednak

dopiero po zapoznaniu się z tablicami, wskaźnikami i funkcjami w następnych rozdziałach.

Aby korzystanie z argumentów wywołania było możliwe, funkcja **main** programu powinna być zdefiniowana z nagłówkiem

```
int main(int argc, char **argv)
```

lub, równoważnie,

```
int main(int argc, char *argv[])
```

Pierwszy parametr, o zwyczajowej nazwie argc, jest liczbą argumentów podanych podczas wywołania programu. Pierwszym z tych argumentów, o numerze 0, jest zawsze nazwa wywoływanego programu. Tak więc argc jest zawsze co najmniej jeden. Zmienna argv wygląda trochę tajemniczo, ale tak naprawdę jest tablicą napisów zawierających kolejne argumenty wywołania: argv[0], argv[1], ..., argv[argc-1]; indeksowanie rozpoczyna się od zera, dlatego ostatnią wartością indeksu jest argc-1 a *nie* argc. Tablica ta jest podobna do tablicy, nazywanej zwykle args, w Javie. Ponieważ, jak się przekonamy, tablice w C/C++ nie są obiektami i nie zawierają odpowiednika zmiennej obiektowej length z Javy, więc trzeba było przekazać tablicę napisów reprezentujących argumenty oraz, osobno, rozmiar tej tablicy (poprzez parametr argc). Więcej szczegółów na temat tablic napisów podamy w rozdz. 5.5.2.

Tak więc, na przykład, program wypisujący swoje argumenty wywołania mógłby mieć postać:

P6: ***argumenty.cpp*** Argumenty wywołania

```
#include <iostream>
using namespace std;

int main(int argc, char **argv) {
    cout << "Nazwa programu:   " << argv[0]  << endl
         << "Ilosc argumentow: " << argc     << endl;
    for (int i = 1; i < argc; i++)
        cout << "Argumentem nr " << i       << " jest "
                                 << argv[i] << endl;
}
```

którego uruchomienie wyglądało następująco:

```
cpp> g++ -o argumenty argumenty.cpp
cpp> argumenty 12 a "b c" 'd e'
Nazwa programu: argumenty
Ilosc argumentow: 5
Argumentem nr 1 jest 12
Argumentem nr 2 jest a
Argumentem nr 3 jest b c
Argumentem nr 4 jest d e
```

Jak widzimy, *można* do programu przekazywać jako pojedyncze argumenty napisy zawierające odstępy, jeśli tylko ujmiemy je w cudzysłowy lub apostrofy. W powyższym przykładzie argumentów wywołania jest pięć; pierwszy, o indeksie 0, jest nazwą programu, w tym przypadku ***argumenty***.

— 3 —

Preprocesor

Wspominaliśmy przy okazji omawiania dyrektywy `#include` o preprocesorze. Jest to specjalny program przetwarzający *tekst* programu w C/C++ przed jego właściwą kompilacją. Instrukcje dla preprocesora nie są zatem elementem samego języka, ale są bardzo pomocne w tworzeniu programu. Omówimy tu tylko najważniejsze z tych instrukcji (zwanych dyrektywami) — w rzeczywistości jest ich więcej. W C++ zalecane jest oszczędne stosowanie dyrektyw preprocesora (choć trudno sobie wyobrazić nieużywanie `#include`); są one jednak bardzo intensywnie używane w programach pisanych w czystym C.

Podrozdziały:

3.1 Co to jest preprocesor?

Preprocesor to program przetwarzający tekst programu w C/C++ *przed* jego właściwą kompilacją. W szczególności, preprocesor *nie* sprawdza poprawności składniowej i nie wie nic o języku C++ ani tym bardziej o języku maszynowym. Szuka on w tekście programu dyrektyw dla siebie, wykonuje je i usuwa z tekstu: kompilator nawet ich nie zobaczy.

Dyrektywy preprocesora nie są przeznaczone dla kompilatora, zatem *nie* kończą się znakiem średnika.

Dzięki preprocesorowi możemy natomiast stosunkowo łatwo zmieniać tekst programu „widziany" przez kompilator. Należy przy tym pamiętać, że podczas kompilacji dyrektywy preprocesora nie są już widoczne — widoczny jest tylko tekst przetworzony przez preprocesor, a linie rozpoczynające się od znaku '#' są usunięte (gdyż spowodowałyby błąd kompilacji). Stwarza to niebezpieczeństwo, że komunikaty kompilatora staną się niezrozumiałe dla programisty, gdyż trudno czasem ustalić, jaki właściwie tekst programu został wysłany do kompilacji po jego przetworzeniu przez preprocesor. Generalnie zatem należy stosować tylko najpotrzebniejsze dyrektywy preprocesora i nie wykorzystywać ich wszystkich możliwości. Wydatnie zwiększa to czytelność kodu. Dlatego nie o wszystkich możliwościach preprocesora będziemy mówić; zainteresowani mogą znaleźć pozostałe informacje w literaturze.

Instrukcje przeznaczone dla preprocesora nazywamy **dyrektywami**. Wszystkie rozpoczynają się od znaku '#' jako pierwszego niebiałego znaku linii. Zazwyczaj jedna dyrektywa zajmuje jedną linię; jeśli jednak nie mieści się w jednej linii, to linię, która ma być kontynuowana, kończymy znakiem '\' (odwrócony ukośnik); musi to być *ostatni* znak linii, za nim nie może być nawet (niewidocznej) spacji. Na przykład

```
#define wymiar 256
```

jest równoważne

```
#define wymiar \
        256
```

Wymieńmy zatem najważniejsze dyrektywy preprocesora.

3.2 Dyrektywy preprocesora

```
#include <file>    #include "file"
```

włącza w miejscu wystąpienia zawartość pliku ***file***. A zatem to, co zobaczy kompilator, to nasz program i ten dołączony dyrektywami `#include`.

Pomiędzy nawiasami kątowymi (znakami mniejszości/większości) lub znakami cudzysłowu a nazwą pliku nie są dopuszczalne żadne białe znaki. Natomiast odstęp pomiędzy dyrektywą `#include` a otwierającym nawiasem/cudzysłowem jest opcjonalny.

W pierwszej z tych form, z nawiasami kątowymi, plik ***file*** jest poszukiwany w znanym preprocesorowi katalogu systemowym, określonym zwykle podczas instalacji pakietów kompilatora w systemie (nazwą tego katalogu jest często ***include***). Jeśli użyto drugiej formy, ze znakami cudzysłowu, to plik jest najpierw poszukiwany w katalogu bieżącym, a jeśli tam nie zostanie znaleziony, to jest dalej traktowany tak samo, jakby jego nazwa była zapisana w nawiasach kątowych. Plik włączany dyrektywą `#include` jest zwykle normalnym plikiem tekstowym zawierającym kod napisany w C/C++ i, ewentualnie, inne dyrektywy preprocesora (a zatem takie dołączanie plików *wolno* zagnieżdżać). Czasami jednak forma pliku systemowego dołączanego przez użycie '`#include <file>`' z nawiasami kątowymi może być inna: może to być jakaś forma już prekompilowana. Jeśli tak jest, to fakt ten powinien być dla użytkownika w zasadzie niewidoczny.

```
#define nazwa wart    #define nazwa    #undefine nazwa
```

pierwsza z tych form zastępuje w tekście programu każde wystąpienie leksemu `nazwa` leksemem `wart`. Zauważmy, że leksem `nazwa` w ogóle w wynikowym tekście programu nie pojawi się — wszystkie jego wystąpienia będą bowiem zastąpione leksemem `wart`. Wewnątrz samego napisu `nazwa` nie może być spacji; wszystko co występuje za nim (łącznie z ewentualnymi spacjami i znakami cudzysłowu) jest traktowane jako `wart`. Używamy terminu *leksem*, aby podkreślić, że zastąpione zostaną wystąpienia `nazwa`

tylko takie, w których `nazwa` jest pełnym symbolem (np. identyfikatorem zmiennej, nazwą funkcji itd.). *Nie* będzie zastępowania, jeśli `nazwa` będzie tylko częścią identyfikatora. Tak więc następujący fragment

```
#define dim 256

int k = dim;
int dimen = 2*dim;
```

jest równoważny

```
int k = 256;
int dimen = 2*256;
```

i w drugim wierszu na szczęście nie pojawi się '`int 256en = 2*256`'. Jest tak dlatego, że w identyfikatorze `dimen` podciąg `dim` nie jest osobnym leksemem.

Efektem ubocznym dyrektywy '`#define nazwa wart`' jest wpisanie przez preprocesor nazwy `nazwa` na listę nazw zdefiniowanych. Jeśli użyjemy formy bez podawania żadnej wartości, '`#define nazwa`', to będzie to jedyny efekt takiej dyrektywy. Nazwa `nazwa` nie będzie wtedy niczym zastępowana, ale będzie uznana za zdefiniowaną i fakt ten możemy później sprawdzać. Jak to zrobić i do czego to się może przydać, pokażemy za chwilę. Częstym błędem jest pisanie na przykład '`#define dim=256`' po której to dyrektywie każde wystąpienie leksemu `dim` zostanie zastąpione nie napisem '256', a napisem '`=256`' (a więc razem ze znakiem równości). Tak więc na przykład legalna instrukcja '`k=m=dim`' zostanie zastąpiona przez '`k=m==256`', co formalnie może być prawidłowym wyrażeniem, ale znaczy kompletnie co innego (tego typu błędy, spowodowane błędami w dyrektywach preprocesora, należą do najtrudniejszych do wykrycia).

Nazwę zdefiniowana przez `#define` można „oddefiniować" za pomocą dyrektywy '`#undefine nazwa`'.

W poniższym fragmencie funkcja **function** zostanie skompilowana po zastąpieniu wszystkich deklaracji typu **int** przez deklarację typu **double**, po czym normalne znaczenie **int** zostanie przywrócone:

```
...
#define int double
int function(int k, int m) {
    int x,y,z;
    ...
    ...
}
#undefine int
...
```

Za pomocą dyrektywy `#define` często definiowane są stałe, których potem używa się przy deklarowaniu tablic jako ich wymiar (patrz (sect. 5.1, str 51). Nie jest to polecana praktyka: znacznie lepiej użyć wtedy stałych definiowanych bezpośrednio w programie, co wyjaśnimy w rozdz. 7.4.2, str 90.

`defined` `!defined`

Funkcja, która może się pojawić w treści dyrektywy przed dowolną nazwą, zwraca 1 (**true**), jeśli ta nazwa jest zdefiniowana, a 0 (**false**) w przeciwnym przypadku. Zwracana wartość może być następnie wykorzystana w dyrektywach warunkowych (patrz niżej). W formie z wykrzyknikiem znaczenie jest odwrócone: zwracaną wartością będzie 1 (**true**), jeśli nazwa występująca po `!defined` *nie* jest zdefiniowana. Aby podkreślić, że `defined` jest swego rodzaju funkcją, dopuszczalny jest też zapis `defined(nazwa)` i `!defined(nazwa)` z użyciem nawiasów. Funkcja `defined` może się pojawiać tylko za `#if`, `#elif` lub jako podwyrażenie bardziej złożonych wyrażeń logicznych (patrz niżej).

`#if` `#ifdef` `#ifndef` `#else` `#elif` `#endif`

Za pomocą tych dyrektyw można zawiadywać kompilacją warunkową, to znaczy włączeniem lub wyłączaniem pewnych fragmentów tekstu programu do tekstu wynikowego jaki zostanie przesłany do kompilacji. Znaczenie `#if`, `#else`, `#endif` jest oczywiste i zgodne z intuicją; `#elif` odpowiada **'else if'**. Typowa konstrukcja z ich użyciem ma postać:

```
1   #define dimen
2   ...
3   ...
4   #if defined dimen
5       // fragment do kompilacji jeśli
6       // 'dimen' jest zdefiniowane
7   #else
8       // fragment do kompilacji jeśli
9       // 'dimen' nie jest zdefiniowane
10  #endif
```

Wyrażenie w linii czwartej może być zastąpione przez '`#ifdef dimen`', czyli można traktować `#ifdef` jako skrót od '`#if defined`'. Podobnie `#ifndef` (od *if not defined*) jest skrótem zastępującym '`#if !defined`'.

Rozpatrzmy przykład. Przypuśćmy, że program będzie kompilowany czasem kompilatorem języka C, a czasem C++. Jeśli jest to C++, to chcielibyśmy używać operatora '<<' do wyprowadzania danych. Jeśli jest to C, to operator ten nie zadziała, więc użyjemy funkcji specyficznej dla C, a mianowicie funkcji **printf** (linia ① programu poniżej). Każdy preprocesor związany z kompilatorem C++ definiuje leksem, czyli **makro** `__cplusplus` (dwa podkreślniki na początku). Preprocesor związany z kompilatorem czystego C takiej nazwy nie definiuje. Zatem można postąpić na przykład tak:

P7: ***cvscpp.c*** Kompilacja warunkowa

```
1 #ifdef __cplusplus
2     #include <iostream>
```

```
3     using namespace std;
4 #else
5     #include <stdio.h>
6 #endif
7
8 int main() {
9 #ifdef __cplusplus
10    cout << "Hello, C++" << endl;
11 #else
12    printf("Hello, C\n");          ①
13 #endif
14 }
```

Aby użyć C++, wywołujemy pod Linuksem kompilator ***g++***, natomiast aby użyć C, wywołujemy ***gcc***, a rozszerzeniem w nazwie pliku z programem powinno byc ***.c***. Efekt widać z przebiegu sesji:

```
cpp> g++ -o cvscpp cvscpp.c
cpp> ./cvscpp
Hello, C++
cpp> gcc -o cvscpp cvscpp.c
cpp> ./cvscpp
Hello, C
cpp>
```

Kompilator Visual Studio definiuje zawsze makro `_WIN32` (jeden znak podkreślenia), nawet na maszynach 64-bitowych, a kompilatory ***gcc*** makro `__linux__` (podwójny znak podkreślenia z obu stron). Można to wykorzystać przy kompilacji tego samego pliku źródłowego na tych dwóch platformach.

Konstrukcja oparta na tym samym pomyśle jest stosowana do zabezpieczenia się przed wielokrotną kompilacją tego samego włączanego pliku. Taka możliwość jest bardzo prawdopodobna, jeśli we włączanych plikach zagnieżdżone są kolejne dyrektywy `#include`. Jeśli np. wielokrotnie użyjemy dyrektywy włączającej plik ***plikh.h*** (czyli '`#include "plikh.h"`'), a w pliku tym zastosujemy konstrukcję (zwaną **include guard**):

```
#ifndef PLIKH_H
#define PLIKH_H

   // właściwy kod

#endif
```

to plik ten zostanie tak naprawdę włączony tylko za pierwszym razem: nazwa ***PLIKH_H*** *nie* będzie wtedy zdefiniowana, zatem '`#ifndef PLIKH_H`' będzie warunkiem prawdziwym. Tak więc wszystko pomiędzy '`#ifndef PLIK_H`' i `#endif` zostanie uwzględnione w dalszym przetwarzaniu. Pierwszym wierszem „widzianym”

wewnątrz bloku będzie linia druga definiująca nazwę ***PLIKH_H***. Zatem przy następnej próbie dołączenia pliku ***plikh.h*** warunek '`#ifndef PLIKH_H`' będzie fałszywy i cały kod, aż do linii zawierającej `#endif`, zostanie opuszczony. Oczywiście, musimy bardzo uważać, aby nie użyć tej samej nazwy makra dla różnych plików.

Większość kompilatorów, choć nie jest to wymagane standardem, sama zadba o jednokrotne włączenie pliku jeśli rozpoczyna się on od linii `#pragma once`.

Do budowania wyrażeń logicznych występujących po `#if` i `#elif` można używać operatorów alternatywy logicznej (`||`), koniunkcji (`&&`) i negacji (`!`), tak jak to robimy w C/C++ (patrz rozdz. 9, str. 121).

```
#error komunikat
```

Napotkanie tej dyrektywy powoduje w czasie kompilacji wyświetlenie informacji zawierającej podany komunikat. Niektóre kompilatory zaraz potem w ogóle przerywają dalsze przetwarzanie (choć inne je kontynuują).

Przypuśćmy, że próbujemy skompilować następujący program:

P8: ***preplog.cpp*** Dyrektywa #error

```
#if   defined(POL) && defined(FRA)
   #error Please define only one country
#elif !(defined(POL) || defined(FRA))
   #error Please define a country
#endif

#ifdef POL
   #define country "Poland"
   #define capital "Warsaw"
#elif defined(FRA)
   #define country "France"
   #define capital "Paris"
#endif

#include <iostream>
using namespace std;

int main() {
    cout << capital << " is the capital of "
         << country << "." << endl;
}
```

Powinno to skończyć się błędem w trzeciej linii, gdyż ani nazwa POL, ani FRA nie jest zdefiniowana. Możemy jednak jedną lub obie te nazwy zdefiniować bezpośrednio w komendzie wywołującej kompilację za pomocą opcji `-Dname`, która definiuje nazwę `name` bez przypisywania jej żadnej wartości (moglibyśmy użyć też opcji `-Dname=cokolwiek`, która dodatkowo przypisałaby wartość `cokolwiek` nazwie `name` — zauważmy, że tu *występuje* znak równości). Efekt można prześledzić z zapisu sesji:

```
cpp> g++ -o preplog preplog.cpp
preplog.cpp:4:5: #error Please define a country
cpp> g++ -o preplog -DPOL -DFRA preplog.cpp
preplog.cpp:2:5: #error Please define only one country
cpp> g++ -o preplog -DPOL preplog.cpp
cpp> ./preplog
Warsaw is the capital of Poland.
cpp> g++ -o preplog -DFRA preplog.cpp
cpp> ./preplog
Paris is the capital of France.
cpp>
```

3.3 Predefiniowane nazwy (makra) preprocesora

Jak wspominaliśmy, każdy preprocesor związany z kompilatorem C++ definiuje makro __cplusplus, a związany z kompilatorem czystego, standardowego C nazwę __STDC__. Nie są to jedyne takie predefiniowane makra.

Niezależnie od platformy powinny zawsze być zdefiniowane niektóre nazwy (makra) — na początku i końcu występują zwykle *podwójne* podkreślniki. Nazwy te mają przypisane wartości, którymi zostanie zastąpione każde ich wystąpienie w tekście programu :

__LINE__ : numer aktualnej linii programu w postaci stałej całkowitej;

__FILE__ : napis zawierający nazwę pliku źródłowego;

__DATE__ : napis zawierający datę kompilacji;

__TIME__ : napis zawierający godzinę kompilacji;

__FUNCTION__ : napis zawierający nazwę funkcji w której makro zostało użyte.

Ostanie makro (__FUNCTION__) formalnie nie jest standardowe, ale zwykle jest zaimplementowane. Przykład użycia tych predefiniowanych nazw znajdujemy w poniższym programiku:

P9: ***dataczas.cpp*** Predefiniowane nazwy preprocesora

```
#include <iostream>
using namespace std;

int main() {
    cout << "Plik:     " << __FILE__     << endl
         << "Data:     " << __DATE__     << endl
         << "Linia:    " << __LINE__     << endl
         << "Czas:     " << __TIME__     << endl
         << "Funkcja:  " << __FUNCTION__ << endl;
}
```

którego kompilacja i uruchomienie miało następujący przebieg:

```
cpp> gpp dataczas.cpp
cpp> ./a.out
Plik:    dataczas.cpp
Data:    Jul 10 2017
Linia:   7
Czas:    23:04:34
Funkcja: main
```

— 4 —

Podstawowe typy danych

W rozdziale tym omówimy wbudowane typy danych języka C/C++. Jest to zagadnienie fundamentalne w językach programowania, ponieważ każde dane muszą mieć zrozumiały dla komputera format (liczby, napisy, itd.) w pamięci, aby program mógł nimi jakoś manipulować. Jak się przekonamy, na podstawie typów już zdefiniowanych, możemy definiować nasze własne. Typem występującym w C/C++, a sprawiającym największe kłopoty początkującym, jest typ wskaźnikowy, a także, w C++, referencyjny (odniesieniowy). Ze wskaźnikami w C/C++ wiąże się ściśle typ tablicowy, którego dyskusję odłożymy do następnego rozdziału. Z kolei zastosowania odniesień dokładniej poznamy przy okazji omawiania funkcji i klas.

Podrozdziały:

4.1 Wstęp

Tak jak w Javie czy Pascalu, obowiązuje w C/C++ zasada **ścisłej kontroli typu**. Dla wszystkich występujących w programie zmiennych ich typ musi być określony (zadeklarowany) zanim mogą być użyte. Samą zmienną możemy uważać za nazwane miejsce w pamięci, w którym przechowywana jest wartość o znanym kompilatorowi typie. Kompilator potrzebuje tej informacji, aby zarezerwować na zmienną odpowiednią ilość pamięci i aby wiedzieć, jak interpretować różne operacje na tej zmiennej.

Zasady poprawności identyfikatorów (nazw) zmiennych są w C/C++ podobne jak w innych językach: identyfikatory mogą składać się z liter, cyfr i znaku podkreślenia; nie mogą rozpoczynać się cyfrą (w odróżnieniu od niektórych innych języków, znaki waluty nie są w identyfikatorach dozwolone).

Duże i małe litery *są* rozróżnialne

Tak więc nazwy (identyfikatory) A_book i a_book będą traktowane jako różne.

Wbudowane typy podstawowe są podobne do tych, jakie być może Czytelnik zna z Javy. Występują jednak pewne różnice. W szczególności, *nie* jest gwarantowana stała długość (w bajtach) reprezentacji maszynowej zmiennych poszczególnych typów. Na przykład, zmienne typu **int** mogą mieć rozmiar, w zależności od implementacji, 2, 4 lub 8 bajtów (obecnie jednak prawie zawsze są to cztery bajty). W związku z tym istnieje przydatny, wbudowany operator **sizeof**, który zwraca długość reprezentacji binarnej zmiennej danego typu na danej platformie (w bajtach). Użyć tego operatora możemy tak jak funkcji, której jedynym argumentem jest albo dowolna zmienna typu, dla którego chcemy poznać długość reprezentacji, albo nazwa samego typu (w pierwszym z tych przypadków nawias jest opcjonalny). Podanie nazwy typu wystarczy, gdyż

Wszystkie zmienne tego samego typu mają ten sam rozmiar.

Jak powiedzieliśmy, wszystkie zmienne — nazwane obszary pamięci o określonym adresie i typie — muszą być przed pierwszym użyciem zadeklarowane i zdefiniowane. Jak to zrobić, pokażmy na przykładzie zmiennych typu **int**:

P10: *vardecl.cpp* Definiowanie zmiennych

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int k1;
6     int k2(1);
7     int k3{};
8     int k4{1};
9     int n=1, m = n, i{1}, j{i};
10 }
```

Jak widać, podajemy najpierw nazwę typu, a potem nazwę definiowanej zmiennej. Nie musimy, ale raczej powinniśmy, nadać nowowprowadzonej zmiennej jakąś sensowną wartość. Robimy to poprzez **zainicjowanie** zmiennej od razu w miejscu definicji. Na przykład w powyższym programie:

- k1 jest zdefiniowana, ale nie zainicjowana; jej wartość jest całkowicie nieokreślona;
- k2 jest zdefiniowana i zainicjowana wartością 1;
- k3 i k4 są zdefiniowane i zainicjowane, ale z użyciem nowej składni wprowadzonej w standardzie C++11. Jest to przykład tzw. **jednolitej inicjalizacji** (*uniform initializer*): z nawiasami klamrowymi (zwana w standardzie **brace-init**).

Ostatnia linia pokazuje, że wiele deklaracji/definicji zmiennych tego samego typu można zapisać w jednej instrukcji; taki zapis jest równoważny serii definicji

```
int n = 1;
int m = n;
int i{1};
int j{i};
```

z czego widać, że, na przykład, definiując m możemy traktować n jako już istniejącą zmienną. Pierwsze dwie linijki pokazują tu „klasyczny” sposób inicjowania definiowanych zmiennych, poprzez użycie znaku równości (normalnie znak równości oznacza *przypisanie*, ale jeśli obiekt po lewej stronie nie istniał wcześniej i jest właśnie tworzony, to *nie* jest to przypisanie, ale właśnie inicjalizacja).

Poniższy program ilustruje definiowanie zmiennych oraz operator **sizeof**:

P11: ***lengths.cpp*** Długości danych różnych typów

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    long double ld = 0;
    string      st = "Hermenegilda";
    short       sh = 0;
    long       *lo = nullptr;
    cout << "long double: " << sizeof ld          << endl
         << "double     : " << sizeof(double)     << endl
         << "float      : " << sizeof(float)      << endl
         << "long long  : " << sizeof(long long)  << endl
         << "long       : " << sizeof(long)       << endl
         << "int        : " << sizeof(int)        << endl
         << "short      : " << sizeof sh          << endl
         << "char       : " << sizeof(char)       << endl
         << "wchar_t    : " << sizeof(wchar_t)    << endl
         << "char16_t   : " << sizeof(char16_t)   << endl
         << "char32_t   : " << sizeof(char32_t)   << endl
         << "bool       : " << sizeof(bool)       << endl
         << "string     : " << sizeof st          << endl
         << "long*      : " << sizeof lo          << endl;
}
```

dał wyniki następujące ma linuksowej maszynie 64-bitowej

```
long double: 16
double     : 8
float      : 4
long long  : 8
long       : 8
int        : 4
```

```
short       : 2
char        : 1
wchar_t     : 4
char16_t    : 2
char32_t    : 4
bool        : 1
string      : 32
long*       : 8
```

Jak widać, w tym systemie typ **long** ma tę samą reprezentację co **long long**; na maszynach 32-bitowych **long** ma zwykle ten sam wymiar co **int** (ale jest traktowany jako osobny typ). Zauważmy też, że wymiar obiektów typu **std::string** zależy od implementacji i może być bardzo różny dla różnych kompilatorów.

Jak widać, deklaracja typu ma postać

```
Typ zmienna;
```

choć może to być też

```
auto zmienna = wartość;
```

lub

```
decltype(wyrażenie) zmienna;
```

Słowo kluczowe **auto** znaczy tu, że kompilator sam ma się domyślić, patrząc na wartość inicjującą, jaki ma być typ deklarowanej/definiowanej zmiennej (oczywiście typ ten będzie ściśle ustalony i nie może być potem zmieniony). Z kolei **decltype** znaczy, że zmienna ma być tego typu, jakiego jest typu wyrażenie podane w nawiasie. Wyrażenie to *nie* będzie obliczane, kompilator sprawdzi tylko jaki byłby typ wyniku. Przykład powinien wyjaśnić sposób, w jaki sposób obie konstrukcje mogą być zastosowane:

P12: ***autodecl.cpp*** Konstrukcje **auto** i **decltype**.

```cpp
#include <iostream>
using namespace std;

int main() {
    auto k = 7;              // k jest typu int
    auto x = 1.;             // x jest typu double
    decltype(x) y = 7;       // y jest typu double, choć
                             // '7' jest literałem typu int
    decltype(k*x) z = 7;     // iloczyn k*x jest typu double
    cout << "k/2="   << k/2 << ", y/2=" << y/2
         << ", z/2=" << z/2 << endl;
}
```

Programik ten drukuje

```
k/2=3, y/2=3.5, z/2=3.5
```

co pokazuje, że rzeczywiście y i z są typu **double** — gdyby były typu **int**, to dzielenie przez 2 dałoby wynik dokładnie 3 (tak jak to jest w przypadku k) Inne aspekty użycia **auto** i **decltype** rozpatrzymy później. Przydatność tych konstrukcji może się na tym etapie wydawać wątpliwa, ale przekonamy się, że są one *niezwykle* przydatne!

4.2 Typy całkowite

Wymieńmy standardowe typy całkowite w C/C++. W nawiasach podane są typowe długości danych poszczególnych typów w bajtach. Typowe, bo standard (w odróżnieniu od standardu Javy) określa tylko minimalne długości oraz to, że długości wymienionych poniżej typów muszą tworzyć ciąg niemalejący. Tak więc typy całkowite to: **char** (1), **short** (2), **int** (4) **long** (4 lub 8), i **long long** (8). Typ **short** można też zapisywać jako **short int**, **long** jako **long int**, a **long long** jako **long long int**. Nowy standard definiuje również trzy typy *znakowe*: **wchar_t** (4), **char16_t** (2), i **char32_t** (4); są one traktowane jako oddzielne (bezznakowe) typy przeznaczone do reprezentowania znaków Unicode.

Wszystkie te typy (z wyjątkiem trzech ostatnich typów znakowych) występują w C/C++ w dwu różnych postaciach: mogą być bez znaku (**unsigned**) lub ze znakiem (**signed**). Nazwa typu bez znaku jest taka sama jak nazwa odpowiadającego typu ze znakiem, tylko poprzedzona słowem kluczowym **unsigned** (np. **unsigned int**) — do nazw typów ze znakiem można zresztą również dodać jawnie **signed**, choć nie ma takiej potrzeby. Samo słowo **unsigned** może być użyte zamiast bardziej dosłownego **unsigned int**.

W przypadku typu **char** w różnych implementacjach bywa różnie: **char** może być fizycznie równoważny z **signed char** lub **unsigned char**. Tym niemniej wszystkie trzy typy są dla kompilatora *różne*! Jeśli „znakowość" ma dla nas znaczenie, trzeba jawnie deklarować zmienne jako **signed char** lub **unsigned char**.

Aby ułatwić internacjonalizację programów, istnieją jeszcze trzy dodatkowe typy reprezentujące znaki. Są one zawsze **unsigned** — ich wersje **signed** nie istnieją. Typy te to:
wchar_t (ang. *wide character* – znak szeroki) — wystarczający do reprezentowania znaków najszerszego zestawu znaków na danej platformie
char16_t — dwa bajty interpretowane jako kod Unicode 16;
char32_t — cztery bajty interpretowane jako kod Unicode 32.

Powiedzmy teraz, na czym polega różnica pomiędzy typami ze znakiem i bez znaku.

W zmiennych *bez znaku* poszczególne bity maszynowej reprezentacji liczby interpretowane są jako zera i jedynki w normalnym zapisie dwójkowym tej liczby.

$$1\,0\,0\,1\,0\,0\,1\,1$$

Rozpatrzmy na przykład liczby całkowite typu **char**, które w C/C++ są (z definicji) jednobajtowe. Zapisane one są na ośmiu bitach, np. jak na rysunku. Kolejne cyfry (od prawej do lewej!) interpretowane są jako współczynniki przy kolejnych potęgach dwójki, poczynając od potęgi zerowej. Tak więc, jeśli zinterpretujemy tę liczbę jako **unsigned char**, otrzymamy wartość W_{uns}:

$$W_{\text{uns}} = w_0 \cdot 2^0 + w_1 \cdot 2^1 + w_2 \cdot 2^2 + w_3 \cdot 2^3 + w_4 \cdot 2^4 + w_5 \cdot 2^5 + w_6 \cdot 2^6 + w_7 \cdot 2^7$$

gdzie dla naszej liczby, od prawej do lewej, $w_i = (1, 1, 0, 0, 1, 0, 0, 1)$. Zatem w tym konkretnym przypadku

$$\begin{aligned} W_{\text{uns}} &= 1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4 + 0 \cdot 2^5 + 0 \cdot 2^6 + 1 \cdot 2^7 \\ &= 1 + 2 + 16 + 128 \\ &= 147 \end{aligned}$$

Zauważmy, że nie ma sposobu, aby zapisać wartość ujemną w zmiennej typu **unsigned**.

Dla zmiennych *ze znakiem* wyraz przy najwyższej potędze dwójki brany jest z minusem; jest to tzw. kod uzupełnień do dwóch. W naszym przykładzie jest to bit ósmy, czyli na pozycji nr 7 (zapisywany po lewej stronie; jak zwykle bowiem liczymy od zera: bit pierwszy, czyli na pozycji zerowej, zapisujemy po stronie prawej). Zatem ten sam układ bitów ma teraz interpretację

$$W_{\text{sgn}} = \sum_{i=0}^{6} w_i \cdot 2^i - w_7 \cdot 2^7$$

czyli w naszym przykładzie

$$\begin{aligned} W_{\text{sgn}} &= 1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4 + 0 \cdot 2^5 + 0 \cdot 2^6 - 1 \cdot 2^7 \\ &= 1 + 2 + 16 - 128 \\ &= -109 \end{aligned}$$

i otrzymujemy teraz wartość ujemną.

Widzimy, że dla typu **unsigned char** największą wartość otrzymamy dla $w_i = 1$ dla $i = 0, \ldots, 7$ i będzie ona wynosić 255. Najmniejsza możliwa wartość to oczywiście zero.

Natomiast dla typu **signed char** największa możliwa wartość zmiennej odpowiada $w_i = 1$ dla $i = 0, \ldots, 6$ i $w_7 = 0$, gdyż wtedy część dodatnia jest największa, a składnika ujemnego nie ma. Odpowiada to liczbie 127 o reprezentacji 01111111. Jeśli jest składnik ujemny, a nie ma części dodatniej, tzn. gdy $w_i = 0$ dla $i = 0, \ldots, 6$ i $w_7 = 1$, to otrzymamy najmniejszą możliwą liczbę w tej reprezentacji. Będzie to w naszym przypadku liczba -128 o reprezentacji 10000000. Liczba -1, jak łatwo się przekonać, ma reprezentację 11111111.

Ogólnie, jeśli liczba zapisana jest na n bitach, to dla odpowiedniego typu **unsigned** zakresem wartości będzie $[0, 2^n - 1]$, natomiast dla typu **signed** będzie to $[-2^{n-1}, 2^{n-1} - 1]$. Zatem dla różnych długości danych całkowitych otrzymamy zakresy zamieszczone w tabeli.

Tablica 4.1: Zakres wartości dla typów całkowitych

Bajtów	Znak	Min	Max
1	signed	-128	127
1	unsigned	0	255
2	signed	-32 768	32 767
2	unsigned	0	65 535
4	signed	-2 147 483 648	2 147 483 647
4	unsigned	0	4 294 967 295
8	signed	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
8	unsigned	0	18 446 744 073 709 551 615

Typ **char** (tak jak pozostałe typy znakowe) jest, co prawda, typem całkowitym, ale jest traktowany inaczej niż typy liczbowe, gdyż w zasadzie służy do reprezentowania znaków. Z tego powodu nie powinniśmy, choć jest to formalnie możliwe, używać zmiennych znakowych w operacjach arytmetycznych. Wartości liczbowe zmiennych tych typów odpowiadają zwykle (choć nie musi tak być) kodom ASCII znaków. Tylko pierwsze 128 znaków ASCII (o kodach od 0 do 127) ma określone znaczenie – interpretacja pozostałych może zależeć od platformy. Co gorsza, standard w zasadzie nie wymaga, aby implementacja w ogóle korzystała z kodowania ASCII. W szczególności oznacza to, że teoretycznie nie ma pewności, iż kolejne litery alfabetu mają kolejne kody. Zwykle tak jednak jest, gdyż większość istniejących programów opiera się na założeniu, że znaki kodowane są według standardu ASCII.

Przykłady deklaracji zmiennych różnych typów:

P13: ***dekl1.cpp*** Deklaracje

```
#include <iostream>
using namespace std;

int main() {
    unsigned long int ul1 = 13UL;
    unsigned long     ul2 = 0xD;  // 13 szesnastkowo
    signed   short    ss1 = 015;  // 13 ósemkowo
    short             ss2 = 13;   // 13 dziesiętnie
    unsigned char     aa1 = 65;
    signed   char     aa2 = 'A';  // ASCII('A') = 65
    int               aa3 = 65;
    int               aa4 = 'A';
    char              aa5 = '\101';   // 65 ósemkowo
    char              aa6 = '\x41';   // 65 szesnastkowo
    cout << aa1 << " " << aa2 << endl
         << aa3 << " " << aa4 << endl
         << aa5 << " " << aa6 << endl;
}
```

Program ten drukuje

```
A A
65 65
A A
```

Wszystkie zmienne aa1...aa4 mają taką samą wartość liczbową: 65 (jest to kod ASCII dużej litery 'A'). Podczas drukowania jednak zmienne typu **char** są traktowane jako znaki, a więc drukowany jest znak odpowiadający kodowi ASCII 65, czyli litera 'A'; zmienne typu **int** natomiast są drukowane jako liczby, nawet jeśli były zainicjowane za pomocą literału znakowego (jak aa4).

Czasem zachodzi potrzeba wyspecyfikowania precyzyjnie typu literału liczbowego. Literał całkowity (np. 12345) będzie zinterpretowany jako literał typu **int** (czyli **signed int**). Jeśli chcemy wymusić zinterpretowanie go jako literału typu bezznakowego, dodajemy na końcu literę 'U' (dużą lub małą), a jeśli ma to być literał typu **long**, dodajemy literę 'L' lub 'LL' dla typu **long long**. Modyfikatory te można łączyć w dowolnej kolejności. Tak więc np. '13UL' jest literałem liczby 13 typu **unsigned long** (zob. zmienna ul1), a '1LL' literałem liczby 1 typu **long long**.

Literały liczbowe (całkowite) można też zapisywać w systemie ósemkowym (oktalnym) i szesnastkowym (heksadecymalnym). Aby literał był potraktowany jako zapis liczby w systemie ósemkowym, poprzedzamy go cyfrą 0 (zero). Tak np. 037 zostanie zinterpretowane jako literał typu **int** liczby o wartości dziesiętnej $3 \cdot 8 + 7 = 31$, a 015 jako $8 + 5 = 13$ (zmienna ss1). Oczywiście, w literałach ósemkowych nie mogą występować cyfry większe od siódemki. W szczególności, symbol '\0' oznacza znak o kodzie ASCII zero (który nazywany jest znakiem NUL i odgrywa ważną rolę w tak zwanych C-napisach).

Literały w układzie szesnastkowym poprzedzamy zerem i literą 'X' (dużą lub małą). A zatem 0x2D to dziesiętnie $2 \cdot 16 + 13 = 45$, a 0xD to dziesiętnie 13 (zmienna ul2). W literałach szesnastkowych jako cyfr od 10 do 15 używamy liter od A do F (dużych lub małych).

Literały znakowe można zapisywać jako pojedynczy znak ujęty w apostrofy (*nie* cudzysłowy!) — jak w linii definiującej aa2. Kłopot pojawia się, jeśli danego znaku nie da się wpisać z klawiatury. Można wtedy zapisać taki literał w postaci '\ooo', gdzie ooo to co najwyżej trzycyfrowa liczba w układzie ósemkowym (w tym przypadku nie musimy pisać wiodącego zera; liczba i tak będzie zinterpretowana w układzie ósemkowym). Przykładem jest definicja aa5. W linii następnej skorzystaliśmy z jeszcze innej formy zapisu znaku: '\Xhh'. Po literze 'X' (lub 'x') mogą wystąpić dwie cyfry, oznaczone tu przez 'hh', które teraz będą zinterpretowane w układzie szesnastkowym. Nie można zapisywać literałów znakowych w tej formie używając zapisu dziesiętnego: trzeba zastosować układ ósemkowy lub szesnastkowy.

Niektóre znaki mają specjalne oznaczenia:

Tablica 4.2: Znaki specjalne w C++

\n	nowa linia (LF)	\t	tabulator poz. (HT)
\v	tabulator pion. (VT)	\b	cofnięcie (BS)
\r	powrót karetki (CR)	\f	nowa strona (FF)
\a	sygnał dźwiękowy (BEL)	\\	ukośnik

Tablica 4.2: Znaki specjalne w C++

\?	pytajnik	\'	apostrof
\"	cudzysłów		

W świetle tego, co powiedzieliśmy zrozumiałe jest, że na przykład poniższa instrukcja

```
cout << "\101ni\x61 Ania\n\x4Fl\141 Ola" << endl;
```

wypisze na ekranie

```
Ania Ania
Ola Ola
```

gdyż np. kod ASCII dużej litery A to 65_{10}=101_8=41_{16}, a pojawienie się '\n' w wyświetlanym tekście powoduje przejście do nowej linii.

4.2.1 Użyteczne aliasy typów całkowitych

Jak mówiliśmy, rozmiary danych poszczególnych typów nie są przez standard ustalone (tak jak są w Javie). Czasem jednak chcielibyśmy być pewni, że deklarowana zmienna ma odpowiedni rozmiar. Na przykład, jeśli nasza zmienna ma być ze znakiem i móc przyjmować wartości większe niż 33 tysiące, to jej wymiar musi być nie mniejszy niż 4 bajty. Na platformie, na której **int** ma 2 bajty (co jest rzadkie, ale możliwe), nie możemy zatem użyć **int**, ale raczej **long** (który musi mieć przynajmniej 4 bajty). W takich sytuacjach można uniknąć deklarowania typu jawnie; możemy posłużyć się aliasami (żargonowo: typedef'ami), zdefiniowanymi w nagłówku ***cstdint***, nazw istniejących typów, które mają żądany wymiar (**typedef** omówimy szerzej w rozdz. 6.2, str. 76). Na przykład, **int32_t** jest aliasem (inną nazwą) całkowitego typu ze znakiem, który, na danej platformie ma rozmiar dokładnie 32 bity (4 bajty) — zazwyczaj będzie to **int**. Liczba w nazwie aliasu wskazuje liczbę bitów, nie bajtów. Tak więc, możemy użyć **int8_t**, **int16_t**, **int32_t**, **int64_t** jako nazwy typu *ze znakiem* o podanym rozmiarze. Podobnie, dodając 'u' na początku nazwy, otrzymujemy aliasy nazw typów *bez znaku*: **uint8_t**, **uint16_t**, **uint32_t**, **uint64_t**. Wszystkie te nazwy są określone poprzez **typedef** w przestrzeni nazw ***std***, więc ich pełne nazwy to **std::int32_t** itd.

Nagłówek ***cstdint*** definiuje również *makra preprocesora*, które rozwijają się do minimalnych i maksymalnych *wartości* danych dla różnych typów: **INT32_MIN**, **INT32_MAX**, **UINT32_MAX**, ect. Nie ma makr dla minimalnych wartości typów bez znaku, bo są one zawsze 0. Ponieważ są to makra, ich nazwy nie mogą być poprzedzane przedrostkiem ***std::***.

4.3 Typy zmiennopozycyjne

Standardowe typy zmiennopozycyjne (zmiennoprzecinkowe) to **float**, **double** i **long double**. Typy zmiennopozycyjne odpowiadają w przybliżeniu matematycznemu po-

jęciu liczb rzeczywistych (z częścią całkowitą i ułamkową).

Dane typu **float** zajmują zwykle 4 bajty, dane typu **double** 8 bajtów, jak to widzieliśmy w programie ***lengths.cpp*** (str. 29). Reprezentacja nie jest przez standard języka określona (tak jak *jest* określona w specyfikacji języka Java), ale we wszystkich współczesnych implementacjach C/C++ jest ona, jak w Javie, zgodna z powszechnie akceptowanym standardem IEEE 754 (IEEE 854 dla typu **long double**). Do wymienionych dwóch typów dochodzi w C/C++ nieznany w Javie typ **long double**: dane tego typu zajmują zwykle 12 lub 16 bajtów, ale w praktyce są używane bardzo rzadko. Rzadko zresztą są również używane zmienne typu **float**. Operacje arytmetyczne na zmiennych tego typu są i tak wykonywane po ich konwersji do typu **double**, więc ich stosowanie w zasadzie nie daje żadnych korzyści, a może wydłużyć czas wykonania i pogorszyć dokładność obliczeń.

Literały wartości typów zmiennopozycyjnych mają standardową, wspólną dla większości języków programowania postać: można używać zapisu z kropką dziesiętną lub notacji naukowej z literą e lub, równoważnie, E, pomiędzy tzw. mantysą a wykładnikiem potęgi dziesięciu, przez którą mantysa ma być pomnożona (kropka w mantysie nie jest konieczna). Wykładnik musi być całkowity; może być poprzedzony znakiem plus lub minus. Domyślnie literał liczbowy zmiennopozycyjny (a więc z kropką lub z częścią potęgową po literze e) jest traktowany jako literał wartości typu **double**. Jeśli, co zdarza się rzadko, chcemy wymusić traktowanie literału jako litarału wartości typu **float** lub **long double**, opatrujemy go odpowiednio literą F lub L:

```
float k = 1.23F, m = .1F, n = 3.F;
double x = 1.2, y = 50., z = 5e-3, v = 0.1e2;
long double u = 1.23L, v = 30.4e-20L;
```

W powyższym przykładzie z wynosi $5 \cdot 10^{-3} = 0.005$ a v ma wartość $0.1 \cdot 10^2 = 10$

4.4 Typ logiczny

Typ logiczny nazywa się **bool** (a nie, jak w Javie, **boolean**). Pamiątką z języka C (albo, sięgając jeszcze głębiej w historię, PL/I) jest, że rolę zmiennej typu **bool** może pełnić dowolna zmienna typu całkowitego lub wskaźnikowego, przy czym wartość 0 (dla wskaźników **nullptr**) jest równoważna **false**, a *dowolna* (niekoniecznie 1) wartość niezerowa — **true**. Tym niemniej, zalecane jest deklarowanie zmiennych logicznych jawnie jako typu **bool**.

Na przykład wykonanie następującego fragmentu spowoduje, że zmienna k będzie miała wartość równą liczbie niezerowych elementów tablicy tab przed pierwszym elementem zerowym (jeśli wiadomo, że element zerowy tam w ogóle jest; w przeciwnym przypadku wynik byłby nieprzewidywalny, nie ma bowiem w C/C++ automatycznego sprawdzania przekroczenia zakresu indeksów tablicy!):

```
int tab[] = { 2, -4, -3, 0, 3 }, k = -1;
while ( tab[++k] );     // teraz k = 3
```

Dzieje się tak dlatego, że gdy wartość k osiągnie 3, odpowiedni element tablicy,

tab[3], będzie miał wartość zerową, co zostanie zinterpretowane jako **false** i zakończy wykonywanie pętli **while**. We wcześniejszych obrotach pętli tab[i] przyjmuje różne wartości, ale wszystkie są niezerowe, więc zostaną zinterpretowane jako **true**. W szczególności należy pamiętać, że prawdzie (**true**) odpowiada dowolna wartość niezerowa, a nie tylko wartość 1, jak to czasem jest błędnie sugerowane.

Do typu **bool** są też konwertowane wskaźniki (o których za chwilę) — wartość pusta (**nullptr** jest interpretowana jako **false**, a każda inna jako **true**.

Literałami wartości typu **bool** są oczywiście **false** i **true** — są to słowa kluczowe języka i nie mogą być przedefiniowane.

Dodajmy, że w klasycznym C typ **bool** w ogóle nie istnieje — jego rolę pełnią wyłącznie typy całkowite lub wskaźnikowe.

4.5 Typy wyliczeniowe

Charakterystycznym dla C/C++ typem (a właściwie typami) danych są **wyliczenia** (zwane też *enumeracjami*, z angielskiego *enumeration*).

Formalnie

> wyliczenie jest typem zdefiniowanym przez zbiór (zwykle niewielki) nazwanych stałych całkowitych.

Na przykład

```
enum dni {pon, wto, sro, czw, pia, sob, nie};
```

definiuje typ wyliczeniowy o nazwie **dni**. Zmienne tego typu będą mogły przybierać dokładnie siedem wartości, wyliczonych w nawiasach klamrowych w definicji tego typu (stąd nazwa *wyliczenia*, typ *wyliczeniowy*). Pewną analogią jest typ **bool**, który ma dwie nazwane wartości – **true** i **false**.

Definicja typu wyliczeniowego składa się ze słowa kluczowego **enum**, nazwy wprowadzanego typu oraz ujętej w nawiasy klamrowe listy nazw symbolicznych wartości elementów wyliczenia. Symbolom można przypisywać wartości liczbowe (patrz niżej), a nazwę typu można opuścić. Jeśli opuszczamy nazwę typu, to zwykle tworzymy od razu zmienne tego anonimowego typu, bo inaczej byłby on bezużyteczny — utworzenie później takich zmiennych byłoby już niemożliwe, bo nie byłoby jak określić ich typu w ich deklaracji/definicji. Na przykład po

```
enum {pik, kier, karo, trefl} karta1, karta2;
...
karta1 = kier;
karta2 = karta1;
...
if (karta2 == trefl) { ... }
```

mamy dwie zmienne (karta1,karta2) anonimowego typu wyliczeniowego opisującego kolory kart; żadnej innej zmiennej tego typu nie da się już utworzyć, bo typ ten nie ma nazwy. Innym zastosowaniem wyliczeń anonimowych jest wprowadzenie do programu nazwanych stałych, które mogą na przykład służyć do wymiarowania tablic wtedy, gdy definiowanie stałych za pomocą słowa kluczowego **const** byłoby niewygodne (np. wewnątrz definicji klas; patrz rozdz. 15.3.2, str 304).

Wewnętrznie zmienne typu **dni** będą reprezentowane za pomocą kolejnych liczb całkowitych poczynając od zera: tak więc symbol pon odpowiadać będzie liczbie 0, a np. symbol sob liczbie 5.

Elementom wyliczenia można nadać odpowiadające im wartości liczbowe „ręcznie”,

```
enum dni {pon, wto=0, sro=0, czw=0, pia=0, sob, nie};
```

przy czym obowiązują następujące zasady:

- Pierwszy element będzie odpowiadał wartości 0 jeśli nie nadano mu innej wartości poprzez jawne przypisanie w definicji wyliczenia. W powyższym przykładzie symbol pon odpowiada wartości liczbowej 0.
- Każdemu innemu elementowi zostanie przypisana wartość o jeden większa od wartości elementu poprzedniego, chyba że poprzez jawne przypisanie w definicji określono inną wartość. Tak więc w naszym przykładzie symbolom od wto do pia zostanie również przypisana wartość 0; symbolowi sob nie została jawnie przypisana żadna wartość, a zatem niejawnie otrzyma on wartość 1 (jako następujący po pia, którego wartość jest 0). Podobnie, nie odpowiadać będzie liczbie 2.

Przypisane elementom wyliczenia wartości nie tylko *nie muszą* być różne dla różnych elementów wyliczenia, ale nie muszą też być wartościami kolejnymi; mogą występować „dziury”. Na przykład definicja **dni** mogłaby wyglądać tak

```
enum dni {pon, wto=0, sro=0, czw=0, pia=0, sob, nie=3};
```

Teraz elementowi sob odpowiada w dalszym ciągu wartość 1, ale niedzieli (nie) odpowiada teraz 3; wartości 2 nie odpowiada zaś teraz żaden element wyliczenia.

W nowym standardzie (C++11), możemy jawnie określić typ stałych wyliczeniowych (musi to być typ całkowitoliczbowy). Robimy to poprzez podanie nazwy typu po nazwie wyliczenia i dwukropku:

```
enum kolor : unsigned {pik, kier, karo, trefl};
```

Według nowego standardu, jeśli nie określimy typu stałych wyliczeniowych, to będzie nim **int** (dotychczas nie był on określony).

Przyjrzyjmy się następnemu przykładowi. Definiujemy w nim (①) wyliczenie **dni**. Definicja ta jest globalna, bo jest umieszczona poza wszystkimi funkcjami i klasami (klas tu zresztą w ogóle nie ma). Jest to konieczne, by definicja typu **dni** była widoczna zarówno w funkcji **main**, jak i w funkcji **info**. Następnie definiujemy funkcję

info, której parametr jest właśnie typu **dni**. W programie głównym **main** wywołujemy tę funkcję. Jako argument podajemy zmienną o wartości typu **dni**. W linii ⑤ jest to pon — literał jednego z elementów wyliczenia **dni**, a w liniach ⑦ i ⑨ jest to zmienna dzien zadeklarowana jako zmienna typu **dni**. Jak widzimy, nazwę **dni** traktujemy tak jak nazwę innych typów (np. **int** czy **double**): deklarując zmienną typu **dni** (⑥), podajemy najpierw nazwę typu, potem nazwę zmiennej i inicjujemy ją jedną z dozwolonych wartości (w tym przypadku wartością sob). W linii ⑧ do tej zmiennej przypisujemy inną wartość typu **dni**, a mianowicie nie.

P14: ***enums.cpp*** Wyliczenia.

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  enum dni {pon, wto=0, sro=0, czw=0, pia=0, sob, nie};    ①
6
7  void info(dni day) {
8      static string  typDnia[]={" powszedni",              ②
9                                "   sobotni", "swiateczny"};
10     int stawka = 200*(1 + day);                          ③
11     cout << "Dzien "  << typDnia[day]   << ". "          ④
12          << "Stawka wynosi: " << stawka << " PLN\n";
13 }
14
15 int main() {
16     info(pon);            ⑤
17
18     dni dzien = sob;      ⑥
19     info(dzien);          ⑦
20
21     dzien = nie;          ⑧
22     info(dzien);          ⑨
23 }
```

Wewnątrz funkcji **info** definiujemy trzyelementową tablicę obiektów typu **string** (słowem **static** na razie się nie przejmujmy). Do elementów tej tablicy odwołujemy się w linii ④. Zauważmy, że w linii tej używamy wartości zmiennej typu **dni** jako indeksów tablicy: jest to dozwolone, gdyż nastąpi automatyczna konwersja (promocja) tych wartości do typu **int** zgodnie z wartościami całkowitymi przypisanymi poszczególnym elementom wyliczenia. W naszym przypadku mogą to być wyłącznie wartości 0, 1 i 2, co akurat odpowiada dozwolonym wartościom indeksu tablicy typDnia. Podobnie, w wierszu ③, widzimy dodawanie '`1 + day`'. Ponieważ jeden argument jest typu **int**, a drugi typu **dni**, nastąpi również promocja wartości zmiennej day do wartości typu **int**. Co jest jednak ważne, to fakt, że *nie* będzie konwersji w przeciwną stronę: od **int** do **dni**. Tak więc niemożliwe byłoby wywołanie funkcji **info** z argumentem innego typu niż typ **dni**; na przykład, wywołanie '`int k = 1; info(k);`' spowodowałoby błąd

kompilacji, mimo że jedynka odpowiada jednej z dozwolonych wartości dla zmiennej typu **dni**. Ta cecha jest bardzo pożyteczną cechą wyliczeń: użycie typu wyliczeniowego zapewnia, że funkcja **info** zostanie zawsze wywołana z legalnym argumentem; musi on bowiem być typu **dni** a więc na pewno odpowiadać którejś z jedynych dopuszczalnych wartości całkowitych: w naszym przypadku 0, 1 lub 2. Dzięki temu nie musimy już sprawdzać, czy indeks w linii ④ nie wykracza poza zakres — użytkownik funkcji **info** zostanie zmuszony do świadomego wyboru argumentu, który na pewno nie spowoduje błędu.

Ponieważ nie ma konwersji od typu **int** do typu **dni**, zmiennym zadeklarowanym jako zmienne typu **dni** można przypisywać wartości wyłącznie tego typu. Choć symbolowi sob odpowiada wartość 1, to

```
dni day = 1; // ZLE
```

byłoby nielegalne; należy użyć

```
dni day = sob;
```

Czytelnik może sprawdzić, że wydruk z powyższego programu wygląda następująco:

```
Dzien  powszedni. Stawka wynosi: 200 PLN
Dzien    sobotni. Stawka wynosi: 400 PLN
Dzien swiateczny. Stawka wynosi: 600 PLN
```

Kontrola legalności argumentów wysyłanych do funkcji to bardzo częste zastosowanie typów wyliczeniowych.

Nowy standard C++11 wprowadza dodatkowo inny sposób na definiowanie typów wyliczeniowych — powiemy o tym po wprowadzeniu pojęcia klas.

4.6 Wskaźniki

Zmienne wskaźnikowe (ang. *pointer*) są fundamentalnym pojęciem C/C++. W programach napisanych w tych językach są one wszechobecne i bez ich zrozumienia nie da się zrozumieć nawet najprostszego kodu napisanego w C/C++.

Wskaźniki umożliwiają bowiem, między innymi:

- przetwarzanie dynamicznych struktur danych,
- zarządzanie blokami pamięci, napisami, tablicami,
- przekazywanie argumentów funkcji, w tym parametrów funkcyjnych.

Szczególnym rodzajem wskaźników są wskaźniki funkcyjne, którymi zajmiemy się w rozdz. 11.12 na stronie 184.

4.6.1 Wskaźniki do zmiennych

Jak pamiętamy, typ zmiennej określa rodzaj informacji zapisanej w tej zmiennej. Dla wskaźników informacją tą jest *adres* w pamięci komputera, pod którym zapisana jest *inna* zmienna. Ta inna zmienna może przy tym być dowolnego typu — również wskaźnikowego. Zazwyczaj, choć, jak się przekonamy, są od tego wyjątki, określając typ zmiennej wskaźnikowej musimy określić też typ tych zmiennych, których adresy dana zmienna wskaźnikowa może przechowywać. To, że wskaźnik musi „wiedzieć", na zmienne jakiego typu może wskazywać, związane jest między innymi z tym, że musi być znana długość (w bajtach) tego typu zmiennych: dzięki temu możliwe są pewne operacje na wskaźnikach, które wkrótce poznamy.

Wartością zmiennej wskaźnikowej jest adres innej zmiennej.

Załóżmy, że chcemy wprowadzić zmienną wskaźnikową przystosowaną do przechowywania adresów zmiennych typu **double**. Po utworzeniu chcemy też przypisać tym zmiennym wartości, a więc wpisać do nich adresy *istniejących* zmiennych typu **double**.

Zobaczmy, jak to wszystko można zrobić na przykładzie poniższego programu:

P15: ***pointers.cpp*** Wskaźniki

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      double x, y = 1.5, u;                    ①
6
7      double *px;                              ②
8      px = &x;
9
10     double* py = &y;                         ③
11
12     double *pz, *pu = &u, v;                 ④
13
14     cout << "1. *py = "  << *py
15          << " y = " << y << endl;            ⑤
16
17     x = 0.5;                                 ⑥
18     cout << "2. *px = "  << *px
19          << " x = " << x << endl;
20
21     *px = 5*x;                               ⑦
22     cout << "3. *px = "  << *px
23          << " x = " << x << endl;
24
25     pz = px;                                 ⑧
```

```
26      cout << "4. *pz = "  << *pz << endl;
27
28      *pu = v = *pz = 10;                    ⑨
29      cout << "5. *pu = "  << *pu
30           << " u = " << u << " v = "
31           << v << " x = " << x << endl;
32
33      cout << "6.  py = " <<  py << endl; ⑩
34      cout << "7. &py = " << &py << endl;
35  }
```

którego uruchomienie daje następujący rezultat:

```
1. *py = 1.5 y = 1.5
2. *px = 0.5 x = 0.5
3. *px = 2.5 x = 2.5
4. *pz = 2.5
5. *pu = 10 u = 10 v = 10 x = 10
6.  py = 0xbffffa30
7. &py = 0xbffffa18
```

W wierszu ① programu deklarujemy i definiujemy (przydzielamy pamięć na) trzy zmienne typu **double**. Jedna z nich, y, jest zainicjowana wartością 1.5; pozostałe nie mają określonej wartości, ale, co ważne, *istnieją*, a zatem mają konkretny i ustalony adres w pamięci komputera.

Linia ② to deklaracja/definicja **zmiennej wskaźnikowej** (krócej: **wskaźnika**) do zmiennej typu **double**. Po wykonaniu tej instrukcji zmienna px istnieje i jest przystosowana do przechowywania *adresów* zmiennych typu **double**. Mówimy, że px jest wskaźnikiem do zmiennej typu **double**. Natomiast zmienną, której adres jest wartością zmiennej wskaźnikowej, nazywamy **zmienną wskazywaną** przez ten wskaźnik.

W naszym przykładzie, na razie, wartość utworzonego wskaźnika px jest nieokreślona, to znaczy nie został tam wpisany adres żadnej istniejącej zmiennej typu **double**. A zatem wskaźnik już istnieje, ale na razie nie wskazuje na żadną zmienną.

Jak widzimy, deklaracja wskaźnika ma postać

```
Typ *nazwa;
```

lub

```
Typ* nazwa;
```

lub nawet

```
Typ * nazwa;
```

czyli nie ma znaczenia, czy „przylepimy” gwiazdkę do nazwy typu czy do nazwy zmiennej — możemy również tę gwiazdkę otoczyć białymi znakami, na przykład znakami odstępu.

Nazwą typu wskaźnika px zadeklarowanego jako '`Typ* px;`' jest **Typ***.

Zapis taki oznacza właśnie, że zmienna o nazwie px będzie zmienną wskaźnikową przystosowaną do przechowywania adresów innych zmiennych, ale koniecznie typu **Typ**. Jak teraz przypisać tej zmiennej jakąś wartość? Widzimy to w następnej linii. Do px wpisujemy tu *adres* zmiennej x, która jest typu **double**, gdyż w naszym przypadku zmienna px była zadeklarowana jako zmienna typu **double***. Zmienna x już istnieje, bo została utworzona w linii ①; niewątpliwie więc ma adres, mimo że jej wartość jest wciąż nieokreślona.

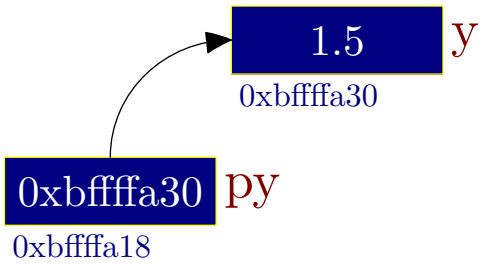
A zatem, jak widzimy, znak '`&`' pełni rolę **operatora wyłuskania adresu** i działa na wyrażenie po swojej prawej stronie. Aby takie wyłuskanie było możliwe, wyrażenie po prawej stronie operatora musi być l-wartością (rozdz. 7.5, str. 96).

Jeśli var jest identyfikatorem zmiennej, to wartością wyrażenia &var jest *adres* tej zmiennej.

Możemy też użyć konstrukcji występującej w linii ③: za jednym zamachem wskaźnik py deklarujemy i inicjujemy adresem istniejącej y, a więc *istniejącej* zmiennej typu **double**. Natomiast wartością zmiennej wskazywanej przez py, czyli zmiennej y, jest 1.5, wpisane do niej w linii ①.

Zmienną, której adres jest wartością zmiennej wskaźnikowej, nazywamy zmienną wskazywaną przez ten wskaźnik.

Sytuację można więc zilustrować jak na rysunku. Zmienna py istnieje w pamięci pod pewnym adresem — w przykładzie na rysunku adres ten to `0xbffffa18` (adresy zapisuje się zwyczajowo w układzie szesnastkowym). Zmienna ta zajmuje 8 bajtów (choć może to zależeć od platformy).



Wartością zapisaną pod tym adresem, a więc wartością zmiennej py jest adres `0xbffffa30` zmiennej y typu **double**. Pod tym z kolei adresem zapisana jest liczba typu **double** (zwykle więc w ośmiu bajtach): jest to wartość zmiennej y wskazywanej przez wskaźnik py. W linii ④ programu ***pointers.cpp*** (str. 41),

```
double *pz, *pu = &u, v;
```

widzimy następną deklarację/definicję trzech zmiennych: pz, pu, i v. Zilustrowany jest tu ważny fakt:

> Deklarując w jednej instrukcji dwa lub więcej wskaźników, należy identyfikator *każdego* z tych wskaźników poprzedzić gwiazdką.

Na przykład, zadeklarowana w linii ④ zmienna v jest typu **double**, a *nie* wskaźnikowego.

Jak poprzez wskaźnik do zmiennej można „dostać" się do samej zmiennej? Robi się to poprzez **operator dereferencji**, zwany też **operatorem wyłuskania wartości**, oznaczany przez gwiazdkę.

> Jeśli pvar jest identyfikatorem wskaźnika, to wyrażenie *pvar jest nazwą (aliasem nazwy) zmiennej wskazywanej przez pvar.

A zatem, py wskazuje na y (patrz linia ③), a wartością zmiennej y jest 1.5 (linia ①). Zatem, ponieważ *py jest nazwą zmiennej wskazywanej przez py, więc jest w tej chwili inną nazwą zmiennej y. Drukując wartość *py (⑤) drukujemy zatem 1.5: aktualną wartość y (patrz linia 1 wydruku z programu).

Podobnie, w linii ⑥ nadajemy wartość 0.5 zmiennej x. Drukując teraz wartość *px drukujemy wartość zmiennej wskazywanej przez px, czyli w tej chwili zmiennej x, czyli 0.5 (linia 2 wydruku).

W linii ⑦ widzimy, że *px, będąc nazwą zmiennej typu **double** wskazywanej przez px, może być użyte po *lewej* stronie przypisania. Ponieważ w tej chwili px wskazuje na x, jest to równoważne przypisaniu 'x = 5*x'. Drukując zatem wartości x i *px otrzymujemy to samo, a mianowicie, zgodnie z przewidywaniem, 2.5.

W linii ④ utworzyliśmy wskaźnik pz. W linii ⑧ wpisujemy do tej zmiennej wartość px; jest nią ten sam adres który jest zapisany w px. Jest to adres zmiennej x. Zatem w tej chwili x, *px i *pz oznaczają to samo: drukując wartość *pz otrzymujemy również 2.5.

W linii ⑨ przypisujemy wartość 10 do *pz. Ponieważ jednak pz wskazuje (po przypisaniu z linii ⑧) na zmienną x, więc przypisując do *pz przypisujemy tak naprawdę do x. Drukując teraz wartość x otrzymujemy, zgodnie z oczekiwaniem, 10, jak widać to w linii 5 wydruku.

Co będzie, jeśli będziemy chcieli wydrukować py? Zmienna ta jest zmienną wskaźnikową, więc jej wartością jest adres, w naszym przykładzie jest to adres zmiennej y. Zatem wstawienie py do strumienia wyjściowego (linia ⑩) spowoduje wypisanie adresu zmiennej y; adres ten to, jak widzimy na wydruku, `0xbffffa30`.

A jaki adres ma sama zmienna py? Żeby to sprawdzić, możemy wydrukować (patrz ostatnia linia) wartość wyrażenia &py: wartością tą jest właśnie adres zmiennej py (a *nie* adres będący *wartością* py). Jak widzimy z wydruku, adresem zmiennej py było `0xbffffa18`. Wyjaśnia to do końca dane z rysunku.

Zauważmy, że operator '`*`' występuje w podwójnej roli (nie licząc jego znaczenia jako operatora mnożenia). Nie powinno to jednak stanowić problemu: jeśli gwiazdka występuje w instrukcji deklaracyjnej i na lewo od niej jest nazwa typu, to znaczy, że chodzi o deklarację/definicję zmiennej wskaźnikowej. W innych przypadkach, jeśli gwiazdka występuje jako operator jednoargumentowy, chodzi o dereferencję. Na przykład w linii

```
int k = 7, *pk = &k, m = *pk;
```

gwiazdka przy pierwszym wystąpieniu pk oznacza deklarację. Co prawda na lewo od niej jest przecinek, a nie nazwa typu, ale pamiętamy, że zapis powyższy jest skrótowym zapisem ciągu deklaracji/definicji

```
int k  = 7;
int *pk = &k;
int m  = *pk;
```

więc wsystko jest tak jak trzeba. Przy drugim wystąpieniu pk, na lewo od gwiazdki jest znak '=', a zatem tutaj chodzi o jednoargumentowy operator — musi to zatem oznaczać dereferencję.

Tak jak w Javie, istnieje wyróżnione odniesienie puste **null** odpowiadające wartości odnośnika nie zawierającego odniesienia do żadnego istniejącego obiektu, tak w C/C++ istnieje wskaźnik „pusty".

W C/C++ rolę wskaźnika pustego, a więc nie zawierającego adresu żadnej zmiennej, może spełniać wskaźnik o wartości 0 (zero).

Wartość 0 można przypisać każdemu wskaźnikowi, niezależnie od jego typu (czyli od typu zmiennej, jaką wskazuje). Zwróćmy tu uwagę na ważny fakt: przypisanie '`pk = 0;`', gdzie pk jest wskaźnikiem, *nie* oznacza wcale, że istnieje jakaś ogólna konwersja od typu całkowitego do typu wskaźnikowego (wartość literału 0 jest typu **int**). Choć adresy, które są wartościami zmiennych wskaźnikowych, są oczywiście całkowite, to typ wskaźnikowy *nie* jest typem całościowym. Wartość zerowa jest wyjątkowa i jest *jedyną* wartością całkowitą, którą można nadać wskaźnikowi poprzez przypisanie: kompilator skonwertuje ją do odpowiedniego typu i otrzymana wartość wskaźnika odpowiadać będzie wskazaniu pustemu. Przypisanie na przykład '`pk = 7;`' spowodowałoby błąd kompilacji.

Każdy wskaźnik przed użyciem należy na coś sensownego „nakierować". Częsty błąd to konstrukcja

```
int *q;
...
...
*q = 7;
```

Zauważmy, że wskaźnik q istnieje, ale nie zawiera na razie adresu żadnej istniejącej zmiennej typu **int**. Zawartość zmiennej q jest w tej chwili całkowicie przypadkowa i nieprzewidywalna. W ostatniej linii tego fragmentu usiłujemy wpisać wartość 7 do zmiennej typu **int** wskazywanej przez q. Ale q niczego nie wskazuje! Wpiszemy więc 7 w całkowicie przypadkowym miejscu pamięci zamazując jego dotychczasową zawartość (jeśli nam system na to pozwoli — jeśli akurat będzie to adres poza obszarem pamięci przydzielonym przez system operacyjny naszemu procesowi, to program załamie się podczas wykonania i na przykład dostaniemy komunikat '`segmentation fault, core dumped`').

Spójrzmy jeszcze na poniższy prosty program

P16: ***pminmax.cpp*** Wskaźniki

```
#include <iostream>
using namespace std;

int main() {
    double x, y, *pmin, *pmax;        ①
    cout << "Podaj dwie liczby: ";
    cin  >> x >> y;
    if (x < y) {
        pmin = &x; pmax = &y;
    } else {
        pmin = &y; pmax = &x;
    }
    cout << "Min = " << *pmin << endl;
    cout << "Max = " << *pmax << endl;
}
```

którego uruchomienie daje

```
Podaj dwie liczby: 9.76 7.34
Min = 7.34
Max = 9.76
```

W linii ① deklarujemy dwie zmienne typu **double** i dwa wskaźniki typu **double***. Następnie wczytujemy dwie liczby do zmiennych x i y, po czym ustawiamy wskaźniki pmin i pmax w ten sposób, aby pmin wskazywało ma mniejszą z liczb x i y, a pmax na większą z nich. Używając następnie operatora dereferencji, w ostatnich dwóch liniach drukujemy najpierw mniejszą, potem większą z liczb.

4.6.2 Wskaźniki generyczne

Czasem potrzebne są wskaźniki wskazujące po prostu pewien obszar pamięci, bez specyfikowania typu zmiennej tam się znajdującej. Służą do tego **wskaźniki generyczne**, które deklarujemy jako typu **void***; w poniższym fragmencie pk, p i q są wskaźnikami typu **int***, natomiast pv jest wskaźnikiem generycznym.

```
1     int k = 8, *pk = &k, *p, *q;
2     void *pv = pk;
3     p = static_cast<int*>(pv);
4     q = (int*)pv;
```

Widzimy, w linii drugiej, że wartość typu **int*** można przypisać do zmiennej typu **void***. Po takim przypisaniu pv zawiera ten sam adres co pk, czyli adres zmiennej k. Jest jednak między nimi ważna różnica: zmienna pv nie zawiera żadnej informacji o typie — jest to „surowy" adres w pamięci. Zatem nie miałoby sensu wyłuskanie wartości (dereferencja) *pv — adres jest znany, ale nie wiadomo byłoby ile bajtów należy uwzględnić i jak je zinterpretować.

Do wskaźników generycznych nie można stosować operatora wyłuskania wartości (dereferencji).

Mając dany w zmiennej typu **void*** surowy adres można, jeśli się wie, co się robi, wymusić zinterpretowanie go jako adresu zmiennej określonego typu. Widzimy to w liniach 3 i 4. Dokonujemy tu jawnej konwersji od typu **void*** do typu **int***. Takie rzutowanie typu oznacza się (podobnie jak w Javie) nazwą typu w nawiasach (patrz linia 4). Jest to jedyny sposób dozwolony w czystym C. W języku C++ istnieje też inna forma, przedstawiona w linii trzeciej (patrz rozdz. 20.2, str. 446). W naszym przykładzie kopiujemy adres zawarty we wskaźniku generycznym pv do p i q — są one typu **int***, więc można dokonać ich dereferencji.

Z tego, co powiedzieliśmy, nie wynika jeszcze jakaś szczególnie fundamentalna rola wskaźników w C/C++. W dalszych rozdziałach jednak, przy okazji omawiania tablic, funkcji, dynamicznego zarządzania pamięcią, polimorfizmu itd., zobaczymy, że właśnie wskaźniki są w tym języku pojęciem centralnym.

4.7 Referencje

Referencje (odnośniki) są często nazywane „przezwiskami" (aliasami) zmiennych — zarówno zmiennych typów prostych, jak i obiektowych. Zostały one wprowadzone w C++; w czystym C takie pojęcie *nie* występuje. Trzeba pamiętać, że w zasadzie nie ma czegoś takiego jak zmienna tego typu. Odnośnik (referencja), jak nazwa sugeruje, zawsze odnosi się do niezależnie istniejącej zmiennej o dobrze określonym typie (**int**, **double**, **string**,...). Tym niemniej referencje nazywa się często zmiennymi odnośnikowymi (referencyjnymi). Jest to dopuszczalne, pod warunkiem, że pamięta się prawdziwe znaczenie referencji. Zatem pamiętajmy, że

referencja jest inną nazwą istniejącej zmiennej.

Zanim wyjaśnimy, co to właściwie znaczy i jaki sens ma nadawanie innych nazw zmiennym, powiemy, jak takie referencje definiować i używać.

Przede wszystkim, skoro referencja to inna nazwa istniejącej zmiennej, to *nie da się* utworzyć referencji bez związania jej ze zmienną, której ma być inną nazwą. Takie związanie referencji ze zmienną jest trwałe: po utworzeniu nie można referencji związać z jakąś inną zmienną.

Tak więc referencja zadeklarowana musi zostać od razu zainicjowana odniesieniem do istniejącej zmiennej: od tej pory nazwa referencji i nazwa tej zmiennej są traktowane równoważnie jako dwie nazwy tej samej zmiennej, a wszelkie operacje na referencjach są równoważne operacjom na pierwotnej zmiennej (ponieważ *są* operacjami na tej zmiennej).

Deklaracja i inicjacja referencji (tu o nazwie refk) może na przykład wyglądać tak:

P17: ***refer.cpp*** Referencje

```
#include <iostream>
using namespace std;

int main() {
    int k = 5;
    int &refk = k;

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

    k = 7;

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

    refk = 9;

    cout << "refk = " << refk << endl;
    cout << "   k = " << k    << endl;
}
```

Na początku deklarujemy/definiujemy tu k, zwykłą zmienną typu **int**. W następnej linii deklarujemy referencję refk do zmiennej typu **int** i związujemy ją z konkretną, istniejącą zmienną tego właśnie typu, zmienną k. Ogólnie zatem

Deklaracja referencji ma postać: 'Typ &ref = zmienna'

gdzie **Typ** jest nazwą pewnego typu, ref dowolną nazwą, jaką nadajemy odnośnikowi, a zmienna nazwą pewnej *już istniejącej* zmiennej typu **Typ**. Po takiej deklaracji

nazwą typu referencji ref jest **Typ&**.

W naszym przykładzie refk odnosi się do k i jak widzimy po wynikach

```
refk = 5
   k = 5
refk = 7
   k = 7
refk = 9
   k = 9
```

nazwy k i refk odnoszą się do tej samej zmiennej: możemy modyfikować tę zmienną poprzez którąkolwiek z tych nazw z tym samym efektem. Również drukowanie wartości zmiennej nie zależy od tego, której nazwy użyliśmy.

Pamiętamy (rozdz. 4.6), że symbol '&' oznacza operator wyłuskania adresu. Tu poznaliśmy jego drugie znaczenie. Podobnie jak to ma miejsce w wypadku symbolu '*' (używanego przy deklarowaniu wskaźników i jako operatora wyłuskania wartości — dereferencji), właściwe znaczenie wynika zawsze z kontekstu: jeżeli chodzi nam o deklarację referencji, to znak '&' jest zawsze poprzedzony nazwą typu, a to z kolei nie jest składniowo możliwe, jeśli '&' ma pełnić rolę jednoargumentowego operatora wyłuskania adresu.

Spójrzmy na prosty, ale pouczający przykład:

P18: ***wskref.cpp*** Wskaźniki i referencje

```
#include <iostream>
using namespace std;

int main () {
    int k = 7, *p = &k, &refk = *p, m = 9;   ①

    p = &m;                                   ②
    k = 11;

    cout << "  *p = " <<   *p << endl;        ③
    cout << "refk = " << refk << endl;        ④
}
```

którego uruchomienie daje:

```
  *p = 9
refk = 11
```

Przyjrzyjmy się deklaracjom/definicjom w linii ①:

- 'k = 7' to definicja zwykłej zmiennej typu **int**, wraz z zainicjowaniem tej zmiennej wartością 7.
- '*p = &k' to deklaracja zmiennej wskaźnikowej p do zmiennej typu **int** (przed gwiazdką jest określenie typu, w takim sensie jak o tym już mówiliśmy w rozdz. 4.6)

i zainicjowaniu jej wartości adresem właśnie przed chwilą utworzonej zmiennej k ('&' za znakiem równości pełni tym razem rolę operatora wyłuskania adresu).

- '&refk = *p' to definicja referencji refk do zmiennej typu **int** (na lewo od '&' stoi określenie typu) i związanie tej referencji ze zmienną wskazywaną przez p. Tym razem gwiazdka występuje za znakiem równości, więc pełni rolę operatora dereferencji — '*p' jest nazwą zmiennej aktualnie wskazywanej przez p. Zmienną aktualnie wskazywaną przez p jest zmienna k — zatem refk zostanie związana *na trwałe* ze zmienną k.
- 'm = 9' to definicja zwykłej zmiennej typu **int**, wraz z zainicjowaniem tej zmiennej wartością 9.

W linii ② zmieniamy wartość p — teraz wskazuje na zmienną m, która ma wartość 9 — wydrukowanie *p (linia ③) daje więc 9. Z kolei wartość k zmieniamy na 11. Zmienna refk została związana na trwałe z k, choć zrobiliśmy to pośrednio, poprzez wskaźnik p. Tego zmienić już się nie da; drukowanie wartości refk zawsze spowoduje pojawienie się na ekranie aktualnej wartości k — drukując w linii ④ wartość refk otrzymujemy 11.

Podobnie jak gwiazdki dla wskaźników, znaki '&' trzeba stawiać przed nazwą każdej referencji na liście deklaracyjnej. Poniższa deklaracja na przykład

```
double x = 3, &y = x, *z = &x, &u = *z;
```

deklaruje dokładnie dwie zmienne: zmienną x typu **double** zainicjowaną wartością 3 oraz zmienną wskaźnikową z typu **double*** zainicjowaną adresem zmiennej x. Zauważmy, że y nie jest zmienną, tylko referencją do zmiennej — mianowicie do zmiennej x. Podobnie u jest referencją do zmiennej aktualnie (w momencie gdy strumień sterowania przechodzi przez tę deklarację) wskazywanej przez wskaźnik z: jest to więc również zmienna x. Po takiej deklaracji referencje y i u będą się odnosić do zmiennej x i tego się już zmienić nie da — zmienna x będzie miała w tym fragmencie programu trzy nazwy!

Wskaźnik z natomiast aktualnie wskazuje na x, ale w dalszej części programu to wskazanie można będzie zmienić.

Wszystko to wygląda nawet zabawnie, ale nie odpowiada na pytanie, po co nam zmienne o kilku różnych nazwach. Rzeczywiście, zwykle referencje nie służą do tego, aby mnożyć nazwy dla tych samych zmiennych. Ich prawdziwa siła tkwi w roli, jaką pełnią przy przesyłaniu danych do funkcji i, odwrotnie, przesyłaniu wyników działania z wywoływanej funkcji do funkcji wywołującej. Dyskusję tego aspektu referencji odłożymy do rozdz. 11.6 (str. 170) i 11.7 (str. 174).

Zauważmy jeszcze, że nie ma czegoś takiego jak tablica referencji. Natomiast mogą istnieć, i są czasem przydatne, referencje do tablic.

— 5 —

Tablice statyczne i wskaźniki

W poprzednim rozdziale omówiliśmy wstępnie typy wskaźnikowe. W tym natomiast omówimy tzw. arytmetykę wskaźników i pokażemy jej nierozerwalny związek z typem tablicowym. Na razie zajmiemy się tablicami statycznymi; omówienie tablic dynamicznych odłożymy do rozdz. 12.2 (str. 212).

Podrozdziały:

5.1 Definiowanie tablic

Tablice są podstawową złożoną strukturą danych. Obecne są we wszystkich niemal językach programowania. Generalnie

> tablice są uporządkowanymi agregatami danych tego samego typu o wspólnym identyfikatorze (nazwie); do poszczególnych elementów mamy dostęp poprzez ich numer kolejny w tablicy.

Numerowanie elementów zawsze rozpoczyna się od zera, tzn. element pierwszy ma numer 0, a element ostatni numer (indeks) o jeden mniejszy od rozmiaru tablicy (czyli liczby elementów tablicy). Jeśli tab jest tablicą, to jej element o indeksie k jest oznaczany `tab[k]`, a więc używamy tu nawiasów kwadratowych.

Jak zadeklarować/zdefiniować tablicę? Można to zrobić na kilka sposobów. Zajmujemy się na razie tablicami statycznymi, to znaczy, że ich rozmiar musi być znany już na etapie kompilacji programu. Tak więc deklarując tablicę musimy określić jej rozmiar. Tablice statyczne nie będące składowymi obiektów klas będą w czasie wykonania tworzone na stosie (a nie na stercie, czyli w tzw. pamięci wolnej, gdzie tworzone są obiekty). Jeśli tablica statyczna jest zadeklarowana w ciele funkcji (ogólniej: w bloku ograniczonym nawiasami klamrowymi), to jest zmienną lokalną i zostanie usunięta po wyjściu przepływu sterowania z tego bloku.

Rozpatrzmy tablice zdefiniowane w poniższym przykładzie:

```
1     const int N = 20; // lub constexpr int N = 20;
2     int tab1[100],
3         tab2[N],
4         tab3[] = {1,2,3,4,5},
5         tab4[5] = {1,2};
6
7     int tab5[]{1,2,3,4,5}, // C++11
8         tab6[5]{1,2},      // C++11
9         tab7[5]{};         // C++11
```

W linii drugiej powyższego przykładu zdefiniowaliśmy tablicę tab1 o 100 elementach typu **int**.

W linii trzeciej zdefiniowaliśmy podobną tablicę, ale o 20 elementach. Do zwymiarowania tablicy tab2 użyliśmy wartości zmiennej N i jest to prawidłowe jeśli, jak w tym przykładzie, zmienna ta zadeklarowana została z modyfikatorem **const** lub, jeszcze lepiej, **constexpr**, i przypisana jej została wartość stała (w naszym przykładzie 20), której nie da się już zmienić. Takie stałe *mogą* być użyte do wymiarowania tablic, bo ich wartość jest już znana w czasie kompilacji. Natomiast *nie* mogą być do tego celu użyte zwykłe zmienne całkowite, a więc zadeklarowane *bez* modyfikatora **const** lub **constexpr** (wiele kompilatorów dopuszcza taką konstrukcję, ale nie jest to zgodne ze standardem; program staje się zatem nieprzenośny). Takie tablice są zatem **statyczne** — ustalenia ich rozmiaru nie można odłożyć do etapu wykonywania programu, a więc, na przykład, uzależnić od wartości przez program wczytywanych.

Tablice tworzone na stosie jako zmienne lokalne muszą być deklarowane z rozmiarem, który jest znany w czasie kompilacji; rozmiar ten musi więc być podany jako literał liczbowy, nazwa zmiennej ustalonej (**const** lub **constexpr**) lub jako wyrażenie zbudowane z literałów i zmiennych ustalonych.

Po takich deklaracjach tablice tab1 i tab2 zostały utworzone, tzn. został im przydzielony odpowiedni obszar w pamięci, ale wartości poszczególnych elementów nie zostały określone i pozostają niezainicjowane: ich wartości są z punktu widzenia programisty całkowicie przypadkowe.

Inaczej jest w przypadku tab3 i tab4. Elementy tablicy tab3 będą zainicjowana podanymi wartościami; tablica będzie miała taki rozmiar, jaki wynika z liczby elementów podanych w nawiasach klamrowych. Zauważmy, że w ogóle nie podaliśmy tu jawnie wymiaru: kompilator policzy zainicjowane elementy i dobierze rozmiar sam.

A jak będzie z tab4? Tu liczbę elementów podaliśmy: ma ich być pięć. Z drugiej strony, zainicjowaliśmy tylko dwa elementy. Otóż kompilator utworzy teraz tablicę pięcioelementową, zainicjuje dwa pierwsze elementy podanymi liczbami, natomiast pozostałe *zainicjuje* zerami. Wynika z tego, że jeśli chcemy utworzyć tablicę o określonym wymiarze i od razu wypełnić ją zerami, to możemy użyć składni

```
int tab[100000] = {};
```

bo podanie inicjatora klamrowego, nawet pustego, wymusza zainicjowanie wszystkich elementów.

Zauważmy też, że w nowym standardzie (C++11) znaki równości przed inicjatorem klamrowym mogą być opuszczone, jak to widzimy w trzech ostatnich liniach przykładu.

5.2 Typ tablicowy

Tam gdzie widoczna jest deklaracja tablicy tab, nazwa tab oznacza zmienną typu tablicowego o określonym rozmiarze. Na przykład po

```
int tab[100];
```

typem tab jest **int[100]**, a więc *stuelementowa tablica liczb całkowitych typu* **int**. Zauważmy, że wymiar jest elementem specyfikacji typu: typy **int[6]** i **int[5]** są *różnymi typami*.

Wartością wyrażenia `sizeof(tab)` będzie zatem rozmiar *w bajtach* całej tablicy. Znając tę wartość można łatwo obliczyć ilość elementów; dla np. tablicy liczb całkowitych, będzie to

```
sizeof(tab) / sizeof(int)
```

lub w formie niezależnej od typu tablicy

```
sizeof(tab) / sizeof(tab[0])
```

Niestety, *nie* oznacza to, że tablice mają własność tablic z Javy polegającą na tym, że „tablica zna swój wymiar". Wymiar jest znany i może być obliczony w sposób powyżej pokazany tylko wewnątrz bloku, gdzie tablica została zdefiniowana. Nie jest to specjalnie użyteczne, bo definiując ją i tak musieliśmy znać jej wymiar. W prawie każdej operacji wykonywanej na zmiennej, a w szczególności przy przesyłaniu tablicy do funkcji, zmienna tab jest niejawnie konwertowana do typu wskaźnikowego. Wartością tab jest wtedy *adres* pierwszego elementu tablicy, a więc elementu o indeksie zero. A zatem po deklaracji

```
int tab[20];
```

zmienna tab może być traktowana jako wskaźnik typu **int* const** wskazujący na pierwszy element tablicy. Modyfikator **const** znaczy tutaj, że zawartość tablicy może być zmieniana, ale nie można do tab wpisać adresu innej tablicy. Ponieważ tab może być przekonwertowane do wskaźnika wskazującego na pierwszy element, więc wyrażenie *tab jest niczym innym jak nazwą pierwszego elementu tablicy.

Konwersja, o której wspomnieliśmy, zachodzi w szczególności, gdy tablicę „wysyłamy" do funkcji. Wysyłamy tak naprawdę (przez wartość) adres początku tablicy i *nic więcej*. W szczególności *nie* ma tam żadnej informacji o wymiarze tablicy. Więcej: nie ma nawet informacji, że to w ogóle jest adres tablicy, a nie adres „normalnej"

pojedynczej zmiennej odpowiedniego typu. Zatem jeśli argumentem funkcji jest tablica, to typem odpowiedniego parametru funkcji (w jej deklaracji/definicji) jest typ wskaźnikowy, a nie tablicowy! Przyjrzyjmy się następującemu programowi:

P19: *tablice.cpp* Tablice jako argumenty

```
#include <iostream>
using namespace std;

void fun1(double tab[]) {
   cout << "Wymiar \'tab\' w fun1: " << sizeof(tab) << endl;
   cout << "Wartosc *tab w fun1: "   << tab[0]      << endl;
}

void fun2(double* tab) {
   cout << "Wymiar \'tab\' w fun2: " << sizeof(tab) << endl;
   cout << "Wartosc *tab w fun2: "   << tab[0]      << endl;
}

int main() {
   double tab[] = {6,2,3,2,1};
   cout << "Wymiar \'tab\' w main: " << sizeof(tab) << endl;
   cout << "Wartosc *tab w main: "   << *tab        << endl;
   fun1(tab);
   fun2(tab);
}
```

którego uruchomienie daje (na platformie, na której wskaźniki są ośmiobajtowe)

```
Wymiar 'tab' w main: 40
Wartosc *tab w main: 6
Wymiar 'tab' w fun1: 8
Wartosc *tab w fun1: 6
Wymiar 'tab' w fun2: 8
Wartosc *tab w fun2: 6
```

W programie głównym (**main**) tworzymy tu tablicę tab. Wypisując teraz wartość `sizeof(tab)` dostajemy 40, co jest prawidłowym wymiarem w bajtach tej tablicy (5 elementów po 8 bajtów). Wartość *tab wynosi 6, bo jest to wartość pierwszego elementu tablicy. Następnie tablicę tab wysyłamy do dwóch funkcji. Funkcje te są prawie identyczne, różnią się tylko deklaracją typu parametru: w funkcji **fun1** mamy **double tab[]**, a w funkcji **fun2 double* tab**. Jak widzimy z wydruku, obie funkcje są równoważne, mimo, że pierwsza forma sugeruje typ tablicowy. W obu, również w **fun1**, gdzie typem zadeklarowanym był **double tab[]**, zmienna tab wewnątrz funkcji jest dokładnie typu **double***. W związku z tym jej wymiar (wynik `sizeof(tab)`) jest teraz osiem (lub cztery na maszynie 32-bitowej)! Jest to bowiem wymiar wskaźnika, a nie tablicy.

Zwróćmy uwagę na funkcję **fun2**. Parametr jest typu **double*** i nigdzie nie ma tu mowy o żadnych tablicach. Mimo to użyliśmy wyrażenia z indeksem, `tab[0]`, tak jak dla tablic! Jest to przykład tzw. arytmetyki wskaźników, którą omówimy w następnym podrozdziale. Jeszcze raz uwypukla to związek typu wskaźnikowego z tablicami.

Tak więc

> przekazując tablicę do funkcji przekazujemy tak naprawdę tylko adres jej początku. We wnętrzu tej funkcji wymiar tablicy *nie jest już znany*.

Wniosek z tego jest taki, że niemal zawsze, kiedy posyłamy do funkcji tablicę, musimy w osobnym argumencie przesłać informację o jej rozmiarze (liczbie elementów).

Jeszcze ważniejszym wnioskiem jest fakt, że choć przekazanie argumentu zachodzi jak zwykle przez wartość, to tą przekazywaną do funkcji wartością jest adres początku tablicy, a nie sama tablica. Dysponując adresem początku, wewnątrz funkcji mamy dostęp do *oryginalnych* elementów tablicy, a nie do ich kopii. Samego wskaźnika do tablicy nie możemy zmienić, bo jej adres funkcja otrzymuje w postaci kopii poprzez stos. Modyfikacje *elementów* tablicy będą jednak widoczne w funkcji wywołującej, bo adres przekazany w kopii wskaźnika był „prawdziwy”. Rozpatrzmy na przykład:

P20: ***tabfunk.cpp*** Przekazywanie tablic do funkcji

```
#include <iostream>
using namespace std;

int* fun(int *tab1, int *tab2, int size) {
    int i, x, y, s1{}, s2{};
    for (i = 0; i < size; ++i) {
        x = tab1[i];
        y = tab2[i];
        tab1[i] = y;
        tab2[i] = x;
        s1 += y;
        s2 += x;
    }
    return s1 > s2 ? tab1 : tab2;
}

void printTable(int *tab, int size) {
    int i;
    for (i = 0; i < size; ++i) cout << tab[i] << " ";
    cout << endl;
}

int main() {
```

```
24      int tab1[]{1,2,3}, tab2[]{4,5,6}, *tab3;
25
26      cout << "tab1 przed: "; printTable(tab1,3);
27      cout << "tab2 przed: "; printTable(tab2,3);
28      tab3 = fun(tab1,tab2,3);
29      cout << "tab1    po: "; printTable(tab1,3);
30      cout << "tab2    po: "; printTable(tab2,3);
31      cout << "tab3      : "; printTable(tab3,3);
32  }
```

Zauważmy, że w definicjach funkcji **fun** i **printTab** parametrami są wskaźniki do zmiennych całkowitych (tab1, tab2, tab) i, osobno, parametr całkowity (size), poprzez który przekazujemy rozmiar — liczbę elementów — tablicy.

Przy wywoływaniu tych funkcji przekazujemy, przez wartość, adresy pierwszych elementów tablic. Funkcja **fun** zamienia elementy dwóch tablic: elementy z tab1 są kopiowane do odpowiednich elementów tab2 i odwrotnie. Przy okazji obliczana jest suma elementów w obu tablicach. Następnie funkcja zwraca wartość albo tab1, albo tab2 w zależności od tego, w której z tablic suma elementów po zamianie była większa (użyta tu konstrukcja oznacza „jeśli `s1>s2` to zwróć tab1, a jeśli nie, to zwróć tab2” — więcej w rozdz. 9.2.10, str. 143). Zauważmy, że w związku z tym zadeklarowanym typem zwracanym funkcji **fun** był typ **int*** — a więc typ wskaźnikowy: zwrócony zostanie adres „większej” tablicy.

Rezultat tego programu to:

```
tab1 przed: 1 2 3
tab2 przed: 4 5 6
tab1    po: 4 5 6
tab2    po: 1 2 3
tab3      : 4 5 6
```

W programie głównym, po powrocie z funkcji **fun** wypisujemy — za pomocą funkcji **printTab** — zawartość tablic: widzimy, że wartości elementów w tablicach zamieniły się. Natomiast rezultat zwracany przez funkcję zapamiętujemy w zmiennej tab3, typu **int***, a nie typu tablicowego. Jej wartość będzie zatem identyczna z wartością jednej ze zmiennych tab1 lub tab2 — po wywołaniu funkcji **printTab** z argumentem tab3 przekonujemy się (ostatnia linia wydruku), że musiał to być adres tablicy tab1. Zauważmy, że wysyłamy tab3 do funkcji **printTab** chociaż tab3 nie jest zadeklarowane jako tablica. To jest legalne, bo funkcja spodziewa się adresu zmiennej typu **int**, a wartość tab3 właśnie jest tego typu.

5.3 Arytmetyka wskaźników

Do wskaźników można dodawać (i odejmować) liczby całkowite. Typ wskaźnikowy *nie jest* jednak typem całkowitym, takie dodawanie jest zdefiniowane w specjalny sposób.

Załóżmy, że p jest wskaźnikiem typu **Typ*** wskazującym na element tablicy, a zmienna shift jest zmienną typu całkowitego.

Wartością wyrażenia `p+shift` jest wtedy adres zawarty w zmiennej p powiększony o shift wielokrotności wymiaru zmiennej typu **Typ** (czyli **sizeof**`(Typ)`).

Jeśli zatem shift wynosi 2, a **sizeof**`(Typ)` wynosi 4 (jak dla **int**), wartością `p+shift` jest adres zawarty w p zwiększony o 8 ($= 2 \cdot 4$). Jeśli (przy tej samej wartości zmiennej shift) **sizeof**`(Typ)` wynosiłby 8, jak dla typu **double**, to wartością `p+shift` byłby adres zawarty w p powiększony o 16 ($= 2 \cdot 8$). Właśnie, między innymi, ze względu na arytmetykę wskaźników deklarując zmienną wskaźnikową trzeba określić, na jakiego typu zmienne będzie ona wskazywać. Z tego też powodu nie można stosować arytmetyki wskaźników do wskaźników generycznych (typu **void***) — brak określonego typu powoduje, że nie wiadomo by było, o ile należy zwiększyć taki adres.

Spójrzmy na następujący przykład:

P21: ***arytmwsk.cpp*** Arytmetyka wskaźników

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int tab[] = {11,22,33,44,55}, i = 3, *p, *q;
6
7      p = &tab[0] + 3;                              ①
8      cout << "*p     = " << *p << endl;
9
10     p = p - 2;                                    ②
11     cout << "*p     = " << *p << endl;
12
13     q = tab;                                      ③
14     cout << "*(q+2) = " << *(q+2) << endl;
15     cout << "q[2]   = " << q[2]   << endl;
16
17     cout << "q[i]   = " << q[i] << endl;           ④
18     cout << "i[q]   = " << i[q] << endl;           ⑤
19 }
```

którego rezultatem jest

```
*p     = 44
*p     = 22
*(q+2) = 33
q[2]   = 33
q[i]   = 44
i[q]   = 44
```

W linii ① wartością wyrażenia `&tab[0]` jest *adres* pierwszego elementu tablicy (czyli elementu tab[0] o wartości liczbowej 11). Zauważmy tu, że

wartość wyrażenia `&tab[0]` jest dokładnie tym samym, co wartość wyrażenia `tab`.

Po dodaniu do tej wartości liczby całkowitej 3 otrzymujemy ten sam adres, ale przesunięty o trzy wielokrotności długości jednej zmiennej typu **int**, czyli adres elementu czwartego (o indeksie 3 i wartości 44). Dlatego wypisując na ekranie wartość *p otrzymamy 44.

W linii ② wartość p pomniejszamy o 2; adres zawarty w p zostaje zatem pomniejszony o dwie wielokrotności długości jednej zmiennej typu **int**, a zatem jest to teraz adres elementu drugiego (o indeksie 1) i wartości liczbowej 22.

W linii ③ wartość tab, czyli adres elementu `tab[0]`, wpisujemy do zmiennej q typu **int***. Zastanówmy się, czym jest `*(q+2)` z następnej linii. Ponieważ q jest wskaźnikiem wskazującym na zmienną tab[0], więc dodanie 2 odpowiada adresowi powiększonemu o dwie długości zmiennej typu **int**. Adres ten zatem to adres elementu tablicy o indeksie 2. Operator dereferencji (gwiazdka) wyłuska wartość zmiennej wskazywanej, czyli 33. Zauważmy, że w takim razie dostaniemy dokładnie to samo, co daje wyrażenie `tab[2]` — wartość elementu tablicy o indeksie 2.

Ogólnie, jeśli p jest wskaźnikiem, a i ma wartość całkowitą, to

wyrażenie `p[i]` jest dokładnie równoważne `*(p+i)`.

W rzeczywistości forma `*(p+i)` jest formą podstawową. Zapis `p[i]` jest tylko ułatwieniem dla programisty: takie wyrażenie zostanie przez kompilator zamienione na formę '`*(p + i)`' jeszcze przed dalszą analizą. Spójrzmy na przykład na dwie ostatnie linie. W linii ④ wyrażenie `q[i]` ma sens `*(q+i)`. Ale w linii ⑤ mamy wyrażenie `i[q]`. Zmienna i nie jest tu wskaźnikiem, tylko zmienną całkowitą; z kolei rolę indeksu pełni tu wskaźnik. A mimo to jest to wyrażenie dziwne, ale jak najbardziej legalne: zostanie przekształcone do formy `*(i+q)`, a to jest prawidłowe: z punktu widzenia arytmetyki wskaźników jest ono równoważne wyrażeniu `*(q+i)`, a zatem również wyrażeniu `q[i]`.

Rozpatrzmy jeszcze następujący przykład, gdzie zastosowano konwersje typów wskaźnikowych:

P22: ***littlebig.cpp*** Konwersje wskaźników

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     // starszy bajt: 'a'; młodszy bajt: 'b'
6     short sh = 'b'+256*'a';
7
8     void *v = static_cast<void*>(&sh);
9     char *c = static_cast<char*>(v);
```

```
10      cout << "Kolejność w pamięci: najpierw "
11           << c[0] << " potem " << c[1] << endl;
12  }
```

Tworzymy tu dwubajtową zmienną typu **short** i wpisujemy tam `'b'+256*'a'`, a zatem liczbę, w której starszym bajtem jest kod ASCII litery 'a' (czyli 97), a młodszym bajtem jest kod ASCII litery 'b' (czyli 98). Adres tej zmiennej konwertujemy najpierw do typu **void***, a potem do typu **char***. Użyty tu operator **static_cast** zostanie wyjaśniony w rozdz. 20.2.1 (str. 446). Po tych operacjach w zmiennej c zawarty jest adres początkowego bajtu z pary bajtów tworzących sh, przy czym typem c jest **char*** (przypomnijmy, że zmienne typu **char** są jednobajtowe). Możemy zatem „zajrzeć do wnętrza" zmiennej sh traktując jej kolejne bajty jako kolejne elementy tablicy c. Pytanie jest, czy bajtem wskazywanym przez c jest starszy czy młodszy bajt liczby sh. Program drukuje

```
Kolejność w pamięci: najpierw b potem a
```

co świadczy o tym, że liczby zapisywane są w kolejnych komórkach pamięci od bajtu najmłodszego do najstarszego. Jest to tzw. kolejność **little endian**. Mogliśmy też dostać rezultat

```
Kolejność w pamięci: najpierw a potem b
```

co świadczyłoby o tym, że pracujemy w architekturze **big endian** (terminy pochodzą z *Podróży Guliwera* Jonathana Swifta). Warto wiedzieć, że standardem przy przesyłaniu danych przez sieć jest właśnie kolejność *big endian*.

Wracając do dyskusji wskaźników zauważmy, że nie ma sensu przesuwanie, czyli dodawanie wartości całkowitych, dla wskaźników generycznych (**void***). Nie wiadomo by bowiem było, o ile bajtów ma nastąpić przesunięcie; wskaźniki takie przechowują surowy adres i nie są związane z żadnym typem danych, który określiłby wielkość pojedynczej „porcji" danych.

Podobnie nie wolno przesuwać wskaźników funkcyjnych, o których powiemy w rozdz. 11.12 na stronie 184.

Ma natomiast sens odejmowanie wskaźników: jeśli p1 i p2 wskazują na dwa elementy tej samej tablicy tab o indeksach odpowiednio i1 i i2, to wyrażenie 'p2-p1' jest równe „odległości" między adresami zmiennych wskazywanych przez p1 i p2 mierzonej w jednostkach równych długości pojedynczej zmiennej w tablicy. A zatem 'p2-p1' ma tę samą wartość co 'i2-i1'.

Dodawanie wskaźników żadnego sensu nie ma.

5.4 Tablice znaków (C-napisy)

Nieco specjalne są tablice i wskaźniki znakowe. Związane jest to z faktem, że w klasycznym C nie ma typu napisowego, a rolę zmiennych napisowych pełnią tablice znaków, przy czym koniec napisu oznaczany jest znakiem '\0'. Znak ten nazywany jest

NUL; nie należy go mylić ze *wskaźnikiem* pustym NULL (podwójne L) używanym w C. Znak NUL to znak o kodzie ASCII równym zeru, który nie odpowiada żadnemu znakowi graficznemu. Znak ten możemy wprowadzić do kodu programu używając literału '\0' (z apostrofami).

Różne możliwości definiowania tablic znakowych (C-napisów) znajdujemy w następującym programie. W linii 9 '`char tab1[] = "Kasia";`' tworzy tablicę sześciu (*sic!*) znaków na podstawie literału napisowego: pięć znaków imienia i jako szósty znak '\0', który musi tam być, aby oznaczyć koniec napisu — kompilator doda ten znak automatycznie.

P23: ***tabchar.cpp*** Napisy: tablice znaków

```
1  #include <iostream>
2  using namespace std;
3
4  void napisz (const char* tab) {
5      cout << "Napis: " << tab << endl;
6  }
7
8  int main() {
9      char  tab1[] = "Kasia";
10     char  tab2[] = {'B', 'a', 's', 'i', 'a', '\0'};
11     const char *tab3 = "Wisia";
12     cout << "Wymiar    tab1: "  << sizeof(tab1)    << endl;
13     cout << "Wymiar    tab2: "  << sizeof(tab2)    << endl;
14     cout << "Wymiar    tab3: "  << sizeof(tab3)    << endl;
15     cout << "Wymiar \'Wisia\': "<< sizeof("Wisia")<< endl;
16
17     tab1[0] = 'B';
18     tab2[0] = 'K';
19     //tab3[0] = 'C'; // ŹLE
20
21     napisz(tab1);
22     napisz(tab2);
23     napisz(tab3);
24 }
```

Dlatego, jak widać z poniższego wydruku, wymiar tablicy tab1 jest 6:

```
Wymiar    tab1: 6
Wymiar    tab2: 6
Wymiar    tab3: 8
Wymiar 'Wisia': 6
Napis: Basia
Napis: Kasia
Napis: Wisia
```

Taki sam jest wymiar tablicy tab2, gdzie inicjalizacji dokonaliśmy sposobem analogicznym do tego, jaki znamy dla tablic innych typów: poprzez podanie wartości kolejnych elementów tablicy w nawiasie klamrowym. Tu znak '\0' musieliśmy dodać „ręcznie". W obu przypadkach powstały sześcioelementowe statyczne tablice znakowe, które, jak widać dalej w programie, można modyfikować.

Szczególna jest definicja zmiennej tab3. Inicjujemy tu zmienną typu **const char*** adresem literału napisowego. Jest to zmienna wskaźnikowa, więc jej wymiar wynosi 8 (lub 4 na maszynie 32-bitowej). Wydawałoby się, że odpowiada to przypadkowi pierwszemu (linia 9). Jest to jednak co innego. W linii 9 utworzyliśmy literał napisowy, następnie na stosie utworzona została tablica sześcioelementowa, do której przekopiowany został, znak po znaku, ten literał (wraz z kończącym znakiem '\0'). Po przekopiowaniu tab1 jest normalną, modyfikowalną tablicą znaków o wymiarze 6.

Natomiast definiując tab3 utworzyliśmy trwały literał napisowy, *nie* na stosie, i przypisaliśmy adres początku tego napisu do zmiennej tab3. Sama tablica będzie utworzona w niemodyfikowalnym obszarze pamięci. Dlatego typ wskaźnika powinien być **const char*** a nie **char*** — o modyfikatorze **const** powiemy w następnym rozdziale. Zauważmy, że kompilator pozwoliłby nam zadeklarować typ tab3 jako **char*** (bez **const**), ale program i tak załamałby się przy próbie modyfikacji elementu wskazywanej tablicy (ta niekonsekwencja jest zaszłością historyczną).

Zauważmy też, że przekazując do funkcji napis w postaci tablicy znakowej *nie* musimy przekazywać jej wymiaru: obecność znaku '\0' pozwala bowiem określić koniec napisu, a więc i jego długość.

Zwróćmy też uwagę na fakt, że wyjątkowe jest traktowanie zmiennych typu **char*** (lub **const char***) przez operator wstawiania do strumienia. W zasadzie po '`cout « nap`', gdzie nap jest typu **char***, powinna zostać wypisana wartość zmiennej nap, czyli pewien adres (tak by było, gdyby zmienna nap była np. typu **int***). Wskaźniki do zmiennych typu **char** są jednak traktowane odmiennie: zakłada się, że wskaźnik wskazuje na pierwszy znak napisu i wypisywane są wszystkie znaki od tego pierwszego poczynając aż do napotkania znaku '\0'. Lepiej więc, żeby ten znak '\0' tam rzeczywiście był!

Więcej na temat napisów w rozdz. 17 na stronie 355.

5.5 Tablice wielowymiarowe

Tablice w C/C++ mogą być wielowymiarowe, choć ich implementacja nie jest tak efektywna jak w innych językach programowania (w szczególności Fortranie). W zasadzie tablica n-wymiarowa jest jednowymiarową tablicą wskaźników do tablic $(n-1)$-wymiarowych,..., i tak dalej, rekursywnie. Implementacja takich tablic jest nieco inna dla napisów i dla danych innych typów. Jako przykład tablic tego drugiego rodzaju rozpatrzymy macierze liczbowe; tablice napisów przedstawimy w podrozdziale następnym. Pamiętajmy, że w tym rozdziale mówimy tylko o tablicach statycznych, a więc takich które tworzone są jako zmienne lokalne (na stosie) i ich wymiar jest znany na etapie kompilacji.

5.5.1 Macierze

Rozpatrzmy następującą deklarację/definicję dwuwymiarowej tablicy:

```
int tab[2][4] = { {1,2,3}, {5,6,7,8} };
```

Zadeklarowaliśmy tu tablicę dwuwymiarową (macierz) o dwóch „wierszach" i czterech „kolumnach". Umawiamy się bowiem, że pierwszy indeks numeruje wiersze, a drugi kolumny (jest to oczywiście konwencja, w pamięci komputera struktura tych danych i tak będzie liniowa).

Liczba elementów naszej tablicy będzie 8. Znaną nam techniką od razu inicjujemy utworzoną tablicę (dwoma tablicami opowiadającymi wierszom tworzonej macierzy). Aby to prawidłowo zrobić, musimy pamiętać o kolejności, w jakiej ułożone są elementy tablicy w pamięci komputera. W C/C++ zapis danych macierzy następuje *wierszami* (a nie kolumnami, jak na przykład w Fortranie) a więc najpierw kolejne elementy wiersza o indeksie 0:

```
tab[0][0]  tab[0][1]  tab[0][2]  tab[0][3]
```

potem elementy wiersza o indeksie 1

```
tab[1][0]  tab[1][1]  tab[1][2]  tab[1][3]
```

Inicjując tablicę ujmujemy poszczególne wiersze w nawiasy klamrowe: tu było to konieczne, gdyż w pierwszym z nich nie podaliśmy wszystkich czterech wartości, a tylko trzy. Ostatni element pierwszego wiersza, czyli element tab[0][3], zostanie zainicjowany zerem. Gdybyśmy nie zostawili żadnych „dziur" i podali wszystkie wartości, to nawiasów klamrowych do zaznaczenia poszczególnych wierszy zwykle nie musimy podawać

```
int tab[2][4] = { 1,2,3,4,5,6,7,8 };
```

choć nie jest to zbyt czytelne. Lepiej więc zawsze ujmować nawet „pełne" wiersze w nawiasy klamrowe; ma to tę dodatkową zaletę, że oddaje prawdziwą naturę takiej tablicy jako tablicy tablic.

Do elementów tablicy dwuwymiarowej odwołujemy się poprzez nazwę tablicy z dwoma indeksami, każdy w osobnej parze nawiasów kwadratowych — np. tab[1][2] ma w naszym przykładzie wartość 7; indeksowanie jest od zera.

Załóżmy, że zadeklarowaliśmy tablicę dwuwymiarową o dim1 wierszach i dim2 kolumnach

```
constexpr int dim1 = ...;
constexpr int dim2 = ...;

int tab[dim1][dim2];
```

Wiemy, że elementy rozmieszczone są w pamięci wierszami. Jak, mając adres początku tablicy i dwa indeksy m i n, znaleźć odpowiadający tym indeksom element

tab[m][n]? Szukany element jest w wierszu $(m+1)$-szym (o indeksie m), a więc, licząc od początku tablicy, musimy najpierw „przeskoczyć" m wierszy, każdy o długości dim2, czyli $m \cdot \text{dim2}$ elementów. Następnie musimy „przeskoczyć" n elementów tego wiersza, w którym znajduje się szukany element (który ma indeks kolumnowy n, czyli jest w tym wierszu $(n+1)$-szym elementem). Zatem przesunięcie elementu tab[m][n] względem elementu tab[0][0] wynosi

$$\text{shift} = n + m \cdot \text{dim2}$$

Co jest tu ważne, to fakt, że do obliczenia tego przesunięcia *nie* jest potrzebna znajomość pierwszego wymiaru tablicy, czyli liczby wierszy dim1. Ogólnie, dla tablic wielowymiarowych, do prawidłowego obliczenia przesunięcia elementu względem początku tablicy potrzebna jest znajomość wszystkich wymiarów *prócz* pierwszego. Rozpatrzmy przykład

P24: ***tab2dim.cpp*** Tablice wielowymiarowe

```
#include <iostream>
using namespace std;

void zamien(int tab[][4], int w1, int w2) {
    int t,k;
    for (k = 0; k < 4; k++) {
        t           = tab[w1][k];
        tab[w1][k] = tab[w2][k];
        tab[w2][k] = t;
    }
}

void printTable(int tab[][4], int dim1) {     ①
    int w,k;
    for (w = 0; w < dim1; w++) {
        for (k = 0; k < 4; k++)
            cout << tab[w][k] << " ";
        cout << endl;
    }
}

int main() {
    int tab[3][4] = { {1,2,3,4}, {5,6}, {1} };

    cout << "Tablica przed:\n"; printTable(tab,3);
    zamien(tab,0,1);
    cout << "Tablica    po:\n"; printTable(tab,3);
}
```

Wynik tego programu to

```
Tablica przed:
1 2 3 4
5 6 0 0
1 0 0 0
Tablica    po:
5 6 0 0
1 2 3 4
1 0 0 0
```

W programie głównym deklarujemy/definiujemy tablicę dwuwymiarową o wymiarach 3×4. W części inicjującej używamy nawiasów dla każdego wiersza, gdyż inicjujemy tylko niektóre elementy poszczególnych wierszy — pozostałe, jak widać z wydruku, są automatycznie inicjowane zerami.

Funkcja **zamien** zamienia dwa wiersze o podanych indeksach. Indeksy tych wierszy są podane jako dwa ostatnie parametry funkcji. Pierwszym parametrem funkcji jest natomiast tablica dwuwymiarowa: zauważmy, że typ parametru został określony jako `int[][4]` — informacja o drugim wymiarze *musi* być podana. Informacja o wymiarze pierwszym jest niepotrzebna do prawidłowego określenia typu i, nawet jeśli ją podamy, zostanie zignorowana. Zauważmy, że informacji o pierwszym wymiarze w ogóle nie przesyłamy do funkcji **zamien**. Przesyłamy ją natomiast do funkcji **printTable**, gdyż tam będzie potrzebna do wydrukowania całej tablicy: pierwszy wymiar przesyłamy jednak nie jako część typu tablicy, ale jako dodatkowy argument funkcji (parametr dim1 — linia ①).

Zastanówmy się, czym, z punktu widzenia związku między tablicami a wskaźnikami, jest zmienna tab po następującej deklaracji/definicji:

```
int tab[dim1][dim2];
```

Mówiliśmy już w rozdz. 5.3, że tab[i] jest równoważne `*(tab+i)`. Na tej samej zasadzie tab[i][j], które ma wartość typu **int**, to to samo co `*(tab[i]+j)`. Zatem tab[i] musi być typu **int***. To jest logiczne: tab[i] *wskazuje* na początek wiersza o indeksie i. Ale tab[i] to `*(tab+i)` i wyłuskana wartość ma być typu **int***, a więc tab musi być typu „wskaźnik do wskaźnika do **int**”. Zatem taki typ tablicowy odpowiada typowi **int****. Odpowiada, ale nie jest z nim tożsamy!

W funkcjach **printTable** i **zamien** musieliśmy jawnie specyfikować drugi wymiar tablicy. A więc funkcje te mogą być zastosowane tylko do takich tablic. Gdybyśmy chcieli zamieniać wiersze w macierzach o innej liczbie kolumn, musielibyśmy napisać do tego osobną funkcję. Jak zatem napisać funkcję bardziej ogólną?

Rozwiązaniem może być zastąpienie macierzy jednowymiarową tablicą wskaźników do poszczególnych wierszy, które są jednowymiarowymi tablicami liczb. Dla tablic jednowymiarowych mamy standardową konwersję do typu wskaźnikowego.

Powyższy program można więc zastąpić takim:

P25: *tab2dim2.cpp* Macierz jako tablica wskaźników

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

```
3
4  void zamien(int* tab[],int w1,int w2,int dim1,int dim2) {
5      for (int k = 0; k < dim2; k++) {
6          int t           = tab[w1][k];
7          tab[w1][k] = tab[w2][k];
8          tab[w2][k] = t;
9      }
10 }
11
12 void printTable(int* tab[], int dim1, int dim2) {
13     for (int w = 0; w < dim1; w++) {
14         for (int k = 0; k < dim2; k++)
15             cout << tab[w][k] << " ";
16         cout << endl;
17     }
18 }
19
20 int main() {
21     int tt[3][4] = { {1,2,3,4}, {7,6}, {1} };       ①
22
23     const int dim1 = 3;
24     const int dim2 = 4;
25
26     int* tab[dim1];                                  ②
27     for (int i = 0; i < dim1; i++)
28         tab[i] = tt[i];
29
30     cout << "Przed:\n"; printTable(tab,dim1,dim2);
31     zamien(tab,0,1,dim1,dim2);
32     cout << "Po:\n";    printTable(tab,dim1,dim2);
33 }
```

Zauważmy, że teraz typem pierwszego parametru funkcji **zamien** jest tablica wskaźników typu **int***. Nie ma tu wyspecyfikowanego raz na zawsze żadnego wymiaru. Aktualne wymiary tablicy przesyłamy osobno poprzez argumenty dim1 i dim2. Zatem funkcję tę można stosować do wszelkich tablic dwuwymiarowych, o dowolnym pierwszym i drugim wymiarze.

Tyle tylko, że co innego trzeba do funkcji „wysłać”. W linii ① tworzymy tablicę dwuwymiarową — tym razem o nazwie tt. Ale funkcja potrzebuje tablicy jednowymiarowej wskaźników wskazujących na początki wierszy. Zatem tworzymy taką tablicę, o nazwie tab, w linii ②. Jej wymiar to liczba wierszy tablicy tt. W następnych dwóch liniach do kolejnych jej elementów wpisujemy wskaźniki do początków wierszy tablicy tt, a więc wartości elementów tt[0], tt[1] i tt[2]. Do funkcji posyłamy teraz nie tablicę tt, tylko tak przygotowaną tablicę tab. Zauważmy, że dzięki arytmetyce wskaźników wewnątrz funkcji możemy odnosić się do elementów naszej tablicy używając notacji z dwoma indeksami, choć posłana do funkcji tablica tab była jednowymiarowa; była

za to tablicą nie wartości typu **int**, ale wskaźników typu **int***.

Rezultat

```
Przed:
1 2 3 4
7 6 0 0
1 0 0 0
Po:
7 6 0 0
1 2 3 4
1 0 0 0
```

przekonuje nas, że obie funkcje działają prawidłowo, bez konieczności wpisywania do ich kodu „na sztywno" żadnych wymiarów.

5.5.2 Tablice napisów

Powyższe stwierdzenia mogą dla początkujących wydawać się nieco zagmatwane. Aby oswoić się z zapisem wskaźnikowym i tablicami, rozpatrzmy teraz jednowymiarowe tablice napisów, które będziemy traktować jak dwuwymiarowe tablice znaków.

P26: ***tabnap.cpp*** Tablice napisów

```
#include <iostream>
using namespace std;

int main() {
    const char **v;
    const char *t[] = {"abcd", "efghi", "jklmno" };   ①
    v = t;                                            ②
    cout << "v+2          = " << v+2          << endl;
    cout << "v[2]         = " << v[2]         << endl;
    cout << "*(v+2)       = " << *(v+2)       << endl;

    cout << "*(*(t+1)+2)  = " << *(*(t+1)+2)  << endl; ③
    cout << "t[1][2]      = " << t[1][2]      << endl;

    cout << "*(*(v+1)+2)  = " << *(*(v+1)+2)  << endl;
    cout << "v[1][2]      = " << v[1][2]      << endl;
}
```

Czym jest `t` zadeklarowane w wierszu ①? Jest tablicą (bo na prawo są nawiasy kwadratowe) wskaźników do **const char** (bo na lewo mamy **const char***). Wskaźniki do **char** odpowiadają, jak wiemy, jednowymiarowym tablicom znaków, czyli napisom: są one zainicjowane napisami podanymi w formie literałów (i dlatego zastosowaliśmy typ ustalony znaków). Ponieważ *'tablica elementów typu* **Typ***'* odpowiada typowi **Typ***,

zatem w naszym przykładzie t, jako tablica elementów typu **const char***, odpowiada typowi **const char****. Dlatego właśnie przypisanie w linii ② jest prawidłowe.

Następnie drukujemy wartość `v+2`. Jest to przesunięty adres zawarty w zmiennej v. Gdyby v było typu **char***, to wydrukowany zostałby napis, ale v jest typu **char****, więc wydrukowany zostanie po prostu adres.

Natomiast w następnych liniach drukujemy wartość zmiennej wskazywanej przez adres `v+2`, czyli wartość v[2] lub, równoważnie, `*(v+2)`. Ta wartość *jest* typu **char***, czyli zawiera adres znaku, więc wydrukowany zostanie nie ten adres, ale napis rozpoczynający się od znaku pod tym adresem. A zatem obie linie spowodują wydrukowanie napisu o indeksie 2, czyli trzeciego: '`jklmno`'.

Dość okropnie wyglądające linie poczynając od ③ ilustrują podobne rozumowanie na poziomie pojedynczych znaków.

Spójrzmy na dwie pierwsze z nich. Zauważmy, że `*(t+1)` to to samo, co t[1], a zatem *wskaźnik* do znaku (konkretnie: adres pierwszego znaku drugiego napisu). Zatem, jeśli ten adres chcemy przesunąć o dwie długości odpowiadające rozmiarowi zmiennej typu **char**, to napiszemy `*(t+1)+2`; jeśli teraz spod tego adresu chcemy wyłuskać wartość (typu **char**), to napiszemy `*(*(t+1)+2)`, co jest równoważne t[1][2]. W sumie oboma sposobami wydobyliśmy trzeci znak z drugiego napisu (literę 'g'), o czym przekonuje nas wydruk z programu:

```
v+2          = 0x7fff364d7080
v[2]         = jklmno
*(v+2)       = jklmno
*(*(t+1)+2)  = g
t[1][2]      = g
*(*(v+1)+2)  = g
v[1][2]      = g
```

W ostatnich dwóch liniach programu pokazujemy, że do tych wszystkich operacji możemy też używać zmiennej v, która w ogóle nie została zadeklarowana jako tablicowa, tylko jako typu **const char****.

Opisane tu konstrukcje dotyczą również tablicy napisów, jaką jest tablica argumentów wywołania — omówiliśmy ją w rozdz. 2.4, a przykład zastosowania pokazaliśmy w programie ***argumenty.cpp*** (str. 16).

Chociaż tablica argumentów wywołania jest zadeklarowana jako char* argv[] a nie const char* argv[], nie powinniśmy nigdy próbować modyfikować napisów wskazywanych przez elementy tej tablicy!

Pewną trudność może sprawić wczytywanie napisów. Związane jest to z tym, że wszelkie białe znaki interpretowane są przez proces czytający jako separatory danych. Jeden ze sposobów jest przedstawiony w poniższym programie (ale, ze względów bezpieczeństwa, nie jest to sposób polecany!); polega na wczytywaniu kolejnych fragmentów do osobnych tablic znaków:

P27: ***czytab.cpp*** Czytanie napisów

```
#include <iostream>
using namespace std;

int main() {
   char nap1[100], nap2[100];

   cout << "\nWpisz dwa napisy oddzielone bialymi znakami: ";
   cin  >>  nap1 >> nap2;
   cout << "\nNapis 1: " << nap1 << endl;
   cout <<   "Napis 2: " << nap2 << endl << endl;
}
```

Tworzymy tablice znakowe (w przykładzie nap1 i nap2) o rozmiarze wystarczającym, aby pomieścić spodziewane napisy. Następnie, za pomocą normalnego mechanizmu zapewnianego przez cin, wczytujemy napisy do tych tablic. Tak jak wspominaliśmy w rozdz. 2.2.2 na stronie 13, wczytywanie jednej danej kończy się, gdy napotkany zostanie biały znak, traktowany jako separator między danymi. Możemy zatem wczytać dwa napisy, jak poniżej

```
Wpisz dwa napisy oddzielone bialym znakiem: 'Ala Ola'

Napis 1: 'Ala
Napis 2: Ola'
```

Zauważmy, że ujęcie napisu w apostrofy nic nie pomaga: tak czy owak znak odstępu jest traktowany jako separator. Zauważmy też, że mechanizm dostarczany przez użycie obiektu cin zapewnia przy wczytywaniu napisów do tablicy znaków dodanie na końcu wczytanego ciągu znaku '\0' kończącego napis. Oczywiście, wymiar tablicy musi być taki, aby ten znak jeszcze w niej się zmieścił. Lepsze sposoby na wczytanie napisów do programu poznamy w rozdz. 16 (str. 323).

5.6 Tablice typu *std::array*

Od wersji standardu C++11 istnieje w bibliotece standardowej specjalny typ danych **std::array** (z nagłówka ***array***). Właściwie, jak się przekonamy, jest to tzw. szablon klasy. Na razie wystarczy nam wiedzieć, że zamiast tworzyć „normalne" tablice w stylu C, możemy tworzyć obiekty klasy **array<Typ,size>**, gdzie **Typ** jest typem elementów tablicy, a size jest jej wymiarem. W dalszym ciągu podany wymiar musi być stałą kompilacji. Zmiennej tego typu można używać tak jak tablicy, poprzez indeksowanie. Również inicjowanie wygląda podobnie jak dla zwykłych tablic; na przykład

```
#include <iostream>
#include <array>
int main() {
```

```
4     std::array<double,5> a1;
5     std::array<int,3>    a2{1,2,3};
6     for (int i = 0; i < 5; ++i) a1[i] = i+0.5;
7     for (int i = 0; i < 5; ++i) std::cout << a1[i] << " ";
8     std::cout << std::endl;
9     for (int i = 0; i < 3; ++i) std::cout << a2[i] << " ";
10    std::cout << std::endl;
11 }
```

drukuje

```
0.5 1.5 2.5 3.5 4.5
1 2 3
```

Tablice tego typu mają jednak szereg zalet w stosunku do zwykłych tablic. Przede wszystkim są obiektami i „znają" swój wymiar, również wtedy, kiedy zostaną wysłane do funkcji jako argument. Można ten wymiar uzyskać poprzez wywołanie metody na rzecz takiego obiektu: `arr.size()` (linia ① w przykładzie poniżej), albo poprzez wywołanie bibliotecznej funkcji: `std::size(arr)` (linia ③), gdzie arr jest taką tablicą. Jak widzimy w przykładzie, to będzie działać również dla tablicy przesłanej do funkcji (w tym przypadku funkcji **printArray**). Do poszczególnych elementów można się odwoływać poprzez indeks, jak to już widzieliśmy, albo poprzez metodę: `arr.at()` (linia ④):

P28: ***Arrays.cpp*** Tablice typu *std::array*

```
1  #include <array>
2  #include <cstddef>   // size_t
3  #include <iostream>
4  #include <string>
5  using std::array; using std::cout;
6
7  void printArray(const array<int,8>& a) {
8      cout << "Version <int,EIGHT>: "
9           << "a.size() = " << a.size() << "\n";          ①
10     for (const auto& e : a) cout << e << " ";          ②
11     cout << "\n";
12 }
13
14 template <typename E, std::size_t SIZE>
15 void printArray(const array<E,SIZE>& a) {
16     cout << "Template version: "
17          << "std::size(a) = " << std::size(a) << "\n"; ③
18     for (std::size_t i = 0; i < a.size(); ++i)
19         cout << a.at(i) << " ";                         ④
20     cout << "\n";
21 }
```

```
22
23 int main() {
24     array<int,8> ai{1,2,3,4,5};
25     for (auto& e : ai) ++e;                                    ⑤
26     printArray(ai);
27
28     array<std::string,5> as{"K","L","M","D"};
29     for (auto& e : as) e += "!";
30     printArray(as);
31 }
```

Przykład ilustruje też kilka elementów, których jeszcze nie znamy. Forma pętli (na przykład w liniach ② i ⑤) jest charakterystyczna dla kolekcji z biblioteki standardowej — omówimy ją w rozdz. 8.9.4 (str. 116). Z kolei druga forma funkcji **printArray** jest tak naprawdę szablonem — dzięki temu może działać dla tablic o różnych typach elementów i o różnych rozmiarach. Więcej o tym w rozdz. 11.14 (str. 200).

Program drukuje

```
Version <int,EIGHT>: a.size() = 8
2 3 4 5 6 1 1 1
Template version: std::size(a) = 5
K! L! M! D! !
```

Może się zdarzyć, że tablicę typu **std::array** chcemy wysłać do funkcji, która oczekuje zwykłej tablicy (czyli właściwie wskaźnika). Można to łatwo zrobić poprzez wywołanie `arr.data()`, które zwraca te same dane jako wskaźnik do „zwykłej” C-tablicy.

5.7 Wektory (*std::vector*)

Tablice statyczne, o których mówiliśmy, należy znać (i czasem stosować...), ale nie są one zbyt wygodne. Tworząc je, musimy z góry znać ich wymiar i to już na etapie pisania kodu źródłowego, bo wymiarem może być tylko stała kompilacji. Niewygodne jest też przesyłanie zwykłych C-tablic do funkcji, bo trzeba osobno przesyłać wymiar, jako że do funkcji przesyłany jest tak naprawdę tylko adres pierwszego elementu. Znacznie wygodniejszym typem tablicy jest **wektor** (**vector** z nagłówka ***vector***). Jest to typ kolekcji bardzo podobny, i podobnie implementowany, jak **ArrayList**, znany nam z Javy. Tak jak **array** jest to właściwie szablon, który musimy parametryzować (w nawiasach kątowych) typem elementów. Wektory, w porównaniu z C-tablicami, mają szereg zalet:

- można tworzyć puste wektory, a potem dodawać tyle elementów ile potrzeba;
- można tworzyć wektory o zadanej zawartości, a potem dodawać następne elementy;

- wektory „znają” swój wymiar: w każdej chwili, dla wektora vec możemy za pomocą wywołania `vec.size()` dowiedzieć się, jaki jest jego aktualny wymiar;
- informacja o wymiarze zachowuje się, jeśli przesyłamy wektor do funkcji (raczej przez referencję, choć niekoniecznie);
- można wektor zmniejszyć, jeśli nie potrzebujemy wszystkich jego elementów;
- dostęp do i-tego (licząc, oczywiście, od zera) elementu poprzez `vec.at(i)` (ale nie `vec[i]`) zapewnia kontrolę legalności wartości indeksu (w przypadku C-tablic żadnej kontroli nie ma).

O kolekcjach, w tym wektorach, będziemy mówić później, bo na razie nie wiemy co to są klasy, metody i szablony, ale poniższy przykład (i dokumentacja) pozwolą nam już je stosować.

P29: *vecsimple.cpp* Wektory

```
#include <iostream>
#include <string>
#include <vector>

int main() {
    using std::vector; using std::cout;

    vector<int> v1{2,1};                                    ①
    vector<int> v2(3,1);                                    ②
    v1.push_back(0);
    v2.push_back(1);
    cout << "v1: size = " << v1.size() << " -> ";
    for (const auto& e : v1) cout << e << " ";
    cout << "\n";
    cout << "v2: size = " << v2.size() << " -> ";
    for (const auto& e : v2) cout << e << " ";
    cout << "\n";

    vector<std::string> v3;                                 ③
    // v3[0] = "A";  // WRONG!!
    v3.push_back("A");
    for (int i = 1; i < 5; ++i)
        v3.push_back(v3.at(i-1) + char('A' + i));           ④
    for (const auto& e : v3) cout << e << " ";
    cout << "\n";

    cout << "First: " << v3.front() << ", last : "          ⑤
         << v3.back() << "\n";
    while (v3.size() > 0) {
        cout << "Removing " << v3.back() << "\n";
```

```
31          v3.pop_back();                                    ⑥
32      }
33  }
```

Powyższy program drukuje

```
v1: size = 3 -> 2 1 0
v2: size = 4 -> 1 1 1 1
A AB ABC ABCD ABCDE
First: A, last : ABCDE
Removing ABCDE
Removing ABCD
Removing ABC
Removing AB
Removing A
```

Stosując nawiasy klamrowe, możemy utworzyć wektor o z góry zadanej zawartości. Na przykład wektor `v1`, utworzony w linii ①, zawiera dwie liczby typu **int** o wartościach 2 i 1. Można też do zainicjowania wektora użyć nawiasów okrągłych (jak w linii ②). Taki zapis znaczy jednak co innego: zainicjuj trzy elementy wektora (pierwszy argument 3) wartością 1 (drugi argument 1). Można w końcu utworzyć wektor pusty, jak w linii ③. Zauważmy, że wtedy nie zawiera on żadnego elementu, więc przypisanie `v3[0]="A"` byłoby nielegalne, bo `v3[0]` znaczy *pierwszy* element, ale przecież w pustym wektorze nie ma ani jednego! Nowe elementy można dodawać do wektora za pomocą **push_back**, jak w przykładzie (albo, jeszcze lepiej, **emplace_back**, którą to funkcję poznamy później). Elementy są wtedy dodawane – co, oczywiście, wiąże się ze zmianą wymiaru – na końcu kolekcji. Istnieją co prawda sposoby, aby dodać nowe elementy na dowolnej pozycji, ale należy ich unikać, bo są bardzo nieefektywne. Dostęp do poszczególnych elementów wektora mamy za pomocą składni `vec[i]` (bez kontroli zakresu indeksu) albo `vec.at(i)` (④, z kontrolą indeksu). Do pierwszego i ostatniego elementu można też się odwołać poprzez `vec.front()` i `vec.back()` (⑤). Ostatni element możemy usunąć za pomocą `vec.pop_back()` (⑥).

Trzeba pamiętać, że w przypadku wektorów, tak jak i innych kolekcji z Biblioteki Standardowej, dodawanie elementu do kolekcji oznacza dodawanie jego *kopii*, natomiast funkcje zwracające elementy kolekcji zwracają do nich *referencje*.

— 6 —

Typy „złożone”

Język dostarcza szeregu predefiniowanych typów danych (jak **int**, **double**, etc.). Z tych typów możemy budować bardzie skomplikowane typy na dwa sposoby: definiując klasy albo konstruując typy złożone jako kombinacje tych, które już mamy. W tym rozdziale będziemy sie zajmować drugą z tych możliwości. Omówimy również bardzo pożyteczne słowa kluczowe **typedef** i **using**, ułatwiające takie konstrukcje.

PODROZDZIAŁY:

6.1 Definiowanie złożonych typów danych

Przez typy „złożone” rozumiemy takie, o których można powiedzieć, że są złożeniami typów różnego rodzaju: na przykład *tablica wskaźników*, *referencja do tablicy* albo *wskaźnik do wskaźnika do tablicy wskaźników* itp. Określanie i zrozumienie tego rodzaju typów sprawia często wiele kłopotu nawet osobom dobrze znającym C/C++.

Definiowanie typów pochodnych może czasem prowadzić do skomplikowanych wyrażeń. Czym na przykład są x, y, z, f po następujących deklaracjach/definicjach:

```
int tab[] = {1,2,3};
int (&x)[3] = tab;
int *y[3] = {tab,tab,tab};
int *(&z)[3] = y;
int &(*f)(int*,int&);
```

Linia 1 określa oczywiście, że tab jest typu *’trzyelementowa tablica zmiennych typu* **int***’*, co w wyrażeniach w sposób naturalny konwertowane jest do typu **int***. Pozostałe deklaracje mogą sprawiać kłopoty. Ogólne zasady są następujące:

1. zaczynamy od nazwy deklarowanej zmiennej,
2. patrzymy w prawo: jeśli jest tam nawias otwierający okrągły, to będzie to *funkcja* (odczytujemy liczbę i typ parametrów); jeśli będzie tam nawias otwierający kwadratowy, to będzie to *tablica* (odczytujemy rozmiar),
3. jeśli po prawej stronie nic nie ma lub jest nawias okrągły zamykający, to przechodzimy na lewo i czytamy następne elementy kolejno od prawej do lewej aż do końca lub do napotkania nawiasu otwierającego,

4. jeśli napotkaliśmy nawias okrągły otwierający, to wychodzimy z całego tego nawiasu i kontynuujemy znów od jego prawej strony,

5. gwiazdkę (’*’) czytamy *jest wskaźnikiem do*,

6. ampersand (’&’) czytamy *jest referencją do*,

7. po odczytaniu liczby i typu parametrów funkcji dalszy ciąg procedury określa typ zwracany tej funkcji,

8. po odczytaniu wymiaru tablicy dalszy ciąg procedury określa typ elementów tablicy.

Rozpatrzmy po kolei linijki naszego przykładu:

```
int (&x)[3] = tab;
```

x jest:

- na prawo nawias zamykający, więc patrzymy na lewo (zasada 3): widzimy ampersand, więc (zasada 6) REFERENCJĄ DO,
- patrzymy dalej w lewo, napotykamy nawias otwierający; wychodzimy zatem z nawiasu (zasada 4), patrzymy na prawo: jest nawias otwierający kwadratowy, więc (zasada 2): TABLICY TRZYELEMENTOWEJ,
- przechodzimy na lewo (zasada 8): ZMIENNYCH TYPU **int**.

Ponieważ x jest referencją, musieliśmy od razu dokonać inicjalizacji — widać, że jest ona prawidłowa, bo tab właśnie jest trzyelementową tablicą **int**-ów.

```
int *y[3] = {tab,tab,tab};
```

y jest:

- na prawo nawias kwadratowy otwierający, więc: TRZYELEMENTOWĄ TABLICĄ,
- patrzymy w lewo i czytamy do końca w lewo, bo nie ma już żadnych nawiasów (zasady 5 i 8): WSKAŹNIKÓW DO ZMIENNYCH TYPU **int**.

Tu nie musieliśmy od razu dokonywać inicjalizacji, ale ta której dokonaliśmy jest prawidłowa, bo tab standardowo jest konwertowana do typu **int***. W tym przykładzie wszystkie elementy tablicy y wskazują na tę samą liczbę całkowitą, a mianowicie na pierwszy element tablicy tab.

```
int *(&z)[3] = y;
```

z jest:

- na prawo nawias okrągły zamykający, więc patrzymy na lewo: ODNOŚNIKIEM DO,
- na lewo nawias okrągły otwierający, więc wychodzimy z całego nawiasu i przechodzimy na prawo, napotykamy nawias otwierający kwadratowy: TRZYELEMENTOWEJ TABLICY,
- patrzymy w lewo i czytamy do końca w lewo, bo nie ma już żadnych nawiasów: WSKAŹNIKÓW DO ZMIENNYCH TYPU **int**.

Tu znów musieliśmy od razu dokonać inicjalizacji — do jej wykonania użyliśmy tablicy y z poprzedniego przykładu.

```
int &(*f)(int*,int&);
```

f jest:

- na prawo nawias okrągły zamykający, więc patrzymy na lewo: WSKAŹNIKIEM DO,
- na lewo nawias okrągły otwierający, więc wychodzimy z całego nawiasu i przechodzimy na prawo: FUNKCJI O DWÓCH PARAMETRACH, PIERWSZYM TYPU **int***, DRUGIM REFERENCYJNYM TYPU **int&**,
- patrzymy w lewo i czytamy do końca w lewo, bo nie ma już żadnych nawiasów: ZWRACAJĄCEJ REFERENCJĘ DO **int**.

O wskaźnikach do funkcji będziemy jeszcze mówić szczegółowo w rozdz. 11.12 na stronie 184. Na razie przykład programu z powyższymi deklaracjami:

P30: ***dekl.cpp*** Złożone deklaracje

```
#include <iostream>

int& fun(int *k, int &m) {
    return *k > m ? *k : m;
}

int main() {
    using std::cout; using std::endl;
    int tab[]{1,2,3};

    int (&x)[3] = tab;
    cout << "x[2]    = " << x[2]    << endl;

```

```
14      int *y[3] = {tab,tab,tab};
15      cout << "y[2][0] = " << y[2][0] << endl;
16
17      int *(&z)[3] = y;
18      cout << "z[2][0] = " << z[2][0] << endl;
19
20      int &(*f)(int*,int&);
21      f = fun;
22      int v1 =  f(&tab[1], tab[2]);            ①
23      int v2 = (*f)(&tab[1], tab[2]);          ②
24      cout << "v1      = " << v1      << endl;
25      cout << "v2      = " << v2      << endl;
26  }
```

W świetle powyższych rozważań powinien być zrozumiały wynik

```
x[2]    = 3
y[2][0] = 1
z[2][0] = 1
v1      = 3
v2      = 3
```

Zauważmy, że obie formy wywołania funkcji, z linii ① i ②, są równoważne: f jest wskaźnikiem do funkcji, ale przy wywołaniu można, choć nie trzeba, używać operatora dereferencji (więcej szczegółów w rozdz. 11.12).

6.2 Specyfikatory *typedef* i *using*

Skomplikowanym typom, które omawialiśmy w poprzednim podrozdziale, można nadawać nazwy („aliasy”) za pomocą deklaracji **typedef**. Procedura jest przy tym następująca:

- piszemy deklarację zmiennej tego typu, któremu chcemy nadać nazwę,
- poprzedzamy tę deklarację słowem kluczowym **typedef**: nazwa zmiennej staje się teraz nazwą nie zmiennej, ale typu, jaki ta zmienna miałaby, gdyby słowa **typedef** nie było.

Pamiętajmy, że w ten sposób określamy *nazwę* (alias) istniejącego, być może skomplikowanego, typu, a *nie* tworzymy nowego typu. Do tworzenia nowych typów służą w C++ inne konstrukcje: klasy i struktury.

Na przykład po deklaracji

```
long MY_INT;
```

zmienna MY_INT byłaby typu **long**. Zatem po

```
typedef long MY_INT;
```

MY_INT jest inną nazwą typu **long**. Deklaracja

```
MY_INT k;
```

jest zatem od tej pory (w zakresie, w którym widoczna jest deklaracja **typedef**) równoważna deklaracji

```
long k;
```

Zwykle jednak wprowadzamy nazwy dla bardziej skomplikowanych typów, często zagnieżdżając deklaracje **typedef**, co jest dozwolone. Jako przykład rozpatrzmy program, w którym:

- **CH1** jest nazwą typu *trzyelementowa tablica znaków*;
- **CH2** jest nazwą typu *dwuelementowa tablica trzyelementowych tablic znaków*;
- **CH3** jest nazwą typu *dwuelementowa tablica dwuelementowych tablic trzyelementowych tablic znaków.*

P31: ***typedef.cpp*** Deklaracja *typedef*

```
#include <iostream>
using namespace std;

int main() {
    typedef char CH1[3];
    typedef CH1  CH2[2];
    typedef CH2  CH3[2];

    CH1 ch1 = {'a','b','c'};

    CH2 ch2 = {{'a','b','c'},{'d','e','f'}};

    CH3 ch3 = {
                  {{'a','b','c'},{'d','e','f'}},
                  {{'g','h','i'},{'j','k','l'}},
              };

    cout << "sizeof(CH1)   = " << sizeof(CH1)   << endl
         << "sizeof(CH2)   = " << sizeof(CH2)   << endl
         << "sizeof(CH3)   = " << sizeof(CH3)   << endl
         << "ch1[2]        = " << ch1[2]        << endl
         << "ch2[1][1]     = " << ch2[1][1]     << endl
         << "ch3[1][0][2] = " << ch3[1][0][2] << endl;
}
```

W tym przypadku typ **CH3** jest koncepcyjnie równoważny *trzywymiarowej tablicy znaków o wymiarach* 2×2×3. Wynik tego programu

```
sizeof(CH1)   = 3
sizeof(CH2)   = 6
sizeof(CH3)   = 12
ch1[2]        = c
ch2[1][1]     = e
ch3[1][0][2] = i
```

a w szczególności wypisane dla poszczególnych typów rozmiary, świadczą o poprawnym zinterpretowaniu zadeklarowanych typów.

Często używa się specyfikatora **typedef**, aby wygodniej określać typ parametrów lub typ zwracany funkcji, szczególnie, jeśli pojawia się w wielu miejscach programu. Rozpatrzmy na przykład program:

P32: ***typedef1.cpp*** Deklaracje *typedef* i funkcje

```
#include <iostream>
using namespace std;

typedef int IN3[][2][2];

int fun(IN3 t) {
    int max = t[0][0][0];
    for (int k = 0; k < 2; ++k)
        for (int j = 0; j < 2; ++j)
            for (int i = 0; i < 2; ++i)
                if (t[k][j][i] > max)
                    max = t[k][j][i];
    return max;
}

int main()
{
    IN3 in3 = { {{4,3},{2,1}},
                {{7,8},{5,6}} };

    int max = fun(in3);

    cout << "max = " << max << endl;
}
```

W przykładzie powyżej typ **IN3** jest nazwą typu *trzywymiarowa tablica liczb o drugim i trzecim wymiarze równym 2*. Dzięki zadeklarowaniu nazwy **IN3** definicja funkcji i deklaracja zmiennej in3 w programie są prostsze i czytelniejsze. Funkcja **fun** znajduje

i zwraca wartość największego elementu argumentu, będącego trzywymiarową tablicą liczb — wynikiem jest oczywiście 'max=8'.

Poczynając od standardu C++11, nowa, bardziej czytelna, składnia może być użyta do definiowania aliasów nazw typów. Zaczynamy od słowa kluczowego **using**, po którym piszemy coś w rodzaju przypisania: nazwa aliasu po lewej stronie a specyfikacja typu po prawej. Po prawej stronie nie piszemy wtedy nazwy „zmiennej". Na przykład, we fragmencie poniżej, kolejne pary definicji aliasów są równoważne:

```
typedef int TAB[3];
using TAB = int[3];

typedef int& (*FUN)(int&,double);
using FUN = int& (*)(int&,double);

typedef double* (&T)[3]
using T =  double* (&)[3]
```

Jak później zobaczymy, gdy będziemy mówić o szablonach, taka składnia jest bardziej elastyczna, pozwala bowiem definiować aliasy nazw dla całych rodzin typów (aliasy mogą być parametryzowane typami).

— 7 —

Zmienne

W rozdziale tym omówimy sposoby definiowania i deklarowania zmiennych. Zapoznamy się ze znaczeniem różnych specyfikatorów i modyfikatorów używanych przy deklarowaniu zmiennych, a mających wpływ na przydział tym zmiennym pamięci lub modyfikującymi ich typ; wyjaśnimy też pojęcia widoczności i zasięgu deklaracji oraz pojęcie l-wartości.

PODROZDZIAŁY:

7.1 Słowa kluczowe i nazwy

Nazwy (identyfikatory) są wprowadzane do jednostki kompilacji (pliku wraz z innymi plikami włączonymi za pomocą `#include`) poprzez **deklaracje**. Deklaracja określa własności i sposób interpretacji nazwy (zmiennej, funkcji, itd.). Zazwyczaj (choć, jak zobaczymy, nie zawsze) deklaracja wiąże się z **definicją**, czyli z przydziałem pamięci na nazwany obiekt. Obowiązuje przy tym *„zasada jednej definicji."* (ang. *one definition rule*)

W jednostce kompilacji nie może być więcej niż jednej definicji tej samej zmiennej, funkcji, klasy, wyliczenia lub wzorca.

Zasada ta nie dotyczy jednak *deklaracji*, które *mogą* się powtarzać.

Język C++ definiuje pewne **słowa kluczowe** jako identyfikatory zarezerwowane: nie mogą pojawić się w innym znaczeniu niż nadane im w standardzie języka. W szczególności, nie mogą być użyte jako identyfikatory (nazwy) zmiennych, funkcji, klas czy przestrzeni nazw. Poniższa tabela przedstawia wszystkie 92 słowa kluczowe języka C++.

Tablica 7.1: Słowa kluczowe języka C++

alignas	const_cast	module	static_cast
alignof	continue	mutable	struct
and	decltype	namespace	switch
and_eq	default	new	synchronized
asm	delete	noexcept	template
atomic_cancel	do	not	this
atomic_commit	double	not_eq	thread_local
atomic_noexcept	dynamic_cast	nullptr	trrow
auto	else	operator	true
bitand	enum	or	try
bitor	explicit	or_eq	typedef
bool	export	private	typeid
break	extern	protected	typename
case	false	public	union
catch	float	register	unsigned
char	for	reinterpret_cast	using
char16_t	friend	requires	virtual
char32_t	goto	return	void
class	if	short	volatile
compl	import	signed	wchar_t
concept	inline	sizeof	while
const	int	static	xor
constexpr	long	static_assert	xor_eq

Dodatkowo, cztery identyfikatory *są* słowami kluczowymi, ale tylko, gdy użyte są w specyficznych kontekstach; są to **override**, **final**, **transaction_safe** i **transaction_safe_dynamic**.

Legalne identyfikatory, inne niż podane tu słowa kluczowe, mogą w zasadzie zawierać dowolną liczbę znaków alfanumerycznych (liter i cyfr) oraz znaki podkreślenia. Formalnie niełacińskie znaki Unicode są dozwolone, ale może to powodować kłopoty i lepiej ich unikać. Tak jak w innych językach, pierwszy znak nazwy nie może być cyfrą. Nazwy zawierające gdziekolwiek dwa następujące po sobie znaki podkreślenia, albo rozpoczynające się od znaku podkreślenia po którym następuje duża litera są zastrzeżone i nie powinny być używane. W końcu, nazwy rozpoczynające się od znaku podkreślenia po którym nie następuje duża litera mogą być używane tylko w zakresie globalnym.

Litery duże i małe *są* rozróżnialne; nazwa val_X jest inna niż np. nazwa val_x.

Przyjęło się, że nazwy zmiennych piszemy małymi literami, nazwy typów wielu programistów rozpoczyna od dużej litery, a nazwy stałych zdefiniowanych przez dyrektywę preprocesora `#define` — samymi dużymi literami.

Jeśli nazwa składa się z kilku słów, to oddzielamy je znakami podkreślenia (numer_klienta) lub każde słowo, z wyjątkiem, być może, pierwszego, rozpoczynamy od litery dużej (nazwaPliku) — jest to tzw. notacja „wielbłądzia” (ang. *CamelCase*).

7.2 Zasięg i widoczność zmiennych

Zasięg (ang. *scope*) deklaracji to ta część programu, gdzie deklaracja jest aktywna, czyli nazwa deklarowana może być użyta i odnosi się do tej właśnie deklaracji. Zakres **widoczności** (ang. *visibility*) natomiast to ten fragment (fragmenty), w których nazwa (identyfikator) może być użyta bez żadnej kwalifikacji (nazwą klasy, przestrzeni nazw itd.). Zakres widoczności zadeklarowanej zmiennej może być węższy niż zasięg deklaracji na skutek przesłonięcia (patrz dalej).

Zmienne zadeklarowane poza wszystkimi funkcjami mają zasięg od miejsca deklaracji do końca pliku (a właściwie jednostki translacji), w szczególności obejmuje on wszystkie funkcje zdefiniowane w tym module po wystąpieniu deklaracji. Zasięg może być rozszerzony na inne jednostki translacji jeśli nazwa jest w nich zadeklarowana ze specyfikatorem **extern**. Mówimy wtedy, że takie zmienne są **eksportowane** (patrz rozdz. 7.3.2). Zmienne (ogólnie: nazwy) zadeklarowane poza wszystkimi funkcjami nazywamy **zmiennymi globalnymi**. W odróżnieniu od zmiennych lokalnych, definiowanych wewnątrz funkcji, *są* one automatycznie inicjowane zerami odpowiedniego typu (dotyczy to również wskaźników). Ich czas życia to okres od początku do końca działania programu.

Zmienne lokalne są deklarowane wewnątrz funkcji, a ogólniej bloku ograniczonego nawiasami klamrowymi. Ich zasięg obejmuje część programu od miejsca deklaracji do końca tego bloku. Parametry funkcji można uważać za zmienne lokalne zdefiniowane na samym początku funkcji i inicjowane w trakcie wykonania wartościami argumentów wywołania. Czasem życia niestatycznej zmiennej lokalnej jest czas od napotkania jej definicji przez przepływ sterowania do jego wyjścia z funkcji, a ogólniej z najwęższego bloku, w którym ta definicja wystąpiła: w tym momencie zmienne są z pamięci usuwane. W szczególności oznacza to, że kiedy przepływ sterowania powraca do tej samej funkcji (bloku), zmienne lokalne są tworzone na nowo i nie „pamiętają" swoich wcześniejszych wartości. Takimi zmiennymi są też zmienne deklarowane w części inicjalizacyjnej pętli **for** (część w nawiasie okrągłym przed pierwszym średnikiem) jak i zmienne deklarowane w części warunkowej instrukcji **if**, **while**, **for**, **switch**: ich zasięgiem jest ciało danej instrukcji (więcej na ten temat powiemy później).

Zmienne globalne mogą być przesłonięte, jeśli wewnątrz funkcji (lub bloku) zadeklarujemy inną zmienną o tej samej nazwie (choć niekoniecznie tym samym typie). Wówczas nazwa tej zmiennej w ciele funkcji odnosi się do zmiennej *lokalnej*. Zmienna globalna istnieje, ale jest w zakresie funkcji (bloku) niewidoczna. Nie znaczy to, że nie mamy do niej dostępu. Do przesłoniętej zmiennej globalnej możemy odwołać się poprzez operator zasięgu '`::`' („czterokropek"). Jeśli na przykład przesłonięta została zmienna o nazwie `k`, to do globalnej zmiennej o tej nazwie odwołać się można poprzez nazwę kwalifikowaną `::k`.

Inicjalizację, operator zasięgu i widoczność zmiennych globalnych ilustruje poniższy program; demonstruje też deklarowanie zmiennych wewnątrz bloku: jeśli wewnątrz bloku zasłonimy zmienną *lokalną*, to ta lokalna zmienna staje się w tym bloku całkowicie niewidoczna, podczas gdy zmienna globalna o tej samej nazwie jest widoczna w dalszym ciągu, jeśli użyjemy operatora zasięgu.

P33: ***przesl.cpp*** Widoczność i przesłanianie zmiennych

```
#include <iostream>

int k;                                          ①

int main() {
    using std::cout; using std::endl;
    cout << "  k: " <<   k << endl;             ②
    cout << "::k: " << ::k << endl;

    int k = 10;                                 ③

    cout << "  k: " <<   k << endl;
    cout << "::k: " << ::k << endl;

    ::k = 1;                                    ④

    cout << "  k: " <<   k << endl;
    cout << "::k: " << ::k << endl;

    {                                           ⑤
        int k = 77;
        cout << "W bloku:"     << endl;
        cout << "  k: " <<   k << endl;
        cout << "::k: " << ::k << endl;
    }

    cout << "Po bloku:" << endl;
    cout << "  k: " <<   k << endl;
    cout << "::k: " << ::k << endl;
}
```

Zmienna k zadeklarowana w linii ① jest zmienną globalną (i jako taka jest zainicjowana zerem). Jest zatem widoczna i w funkcji **main**. W linii ② drukujemy wartość k — ponieważ k nie zostało przesłonięte inną zmienną, w tym miejscu programu zarówno k, jak i ::k odnoszą się do tej samej zmiennej — zmiennej globalnej.

W linii ③ wprowadzamy *lokalną* dla funkcji **main** zmienną k. Przesłania ona globalne k; od tej pory użycie w funkcji **main** nazwy k oznacza odniesienie się do zmiennej *lokalnej*. Jednak zmienna globalna jest wciąż dostępna: trzeba tylko odnosić się do niej za pomocą pełnej nazwy kwalifikowanej operatorem zakresu, a więc ::k. W linii ④ zmieniamy wartość zmiennej globalnej — wartość *lokalnej* zmiennej k pozostaje oczywiście niezmieniona.

W linii ⑤ otwieramy blok wewnętrzny zanurzony w bloku, jakim jest ciało funkcji **main**. Wprowadzona w tym bloku zmienna k *całkowicie przesłania* lokalną dla **main** zmienną k. Zauważmy tu na marginesie, że takie przesłonięcie byłoby nielegalne

w Javie. Wewnątrz tego zagnieżdżonego bloku mamy dostęp wyłącznie do k z tego bloku i k globalnego. Natomiast po wyjściu z tego bloku, zdefiniowana w nim zmienna jest usuwana i znów staje się widoczna zmienna k zdefiniowana w linii ③. Możemy się o tym przekonać patrząc na wynik programu

```
  k: 0
::k: 0
  k: 10
::k: 0
  k: 10
::k: 1
W bloku:
  k: 77
::k: 1
Po bloku:
  k: 10
::k: 1
```

7.3 Klasy pamięci

Klasa pamięci zmiennej (ang. *storage class specifier*) określana jest w deklaracji za pomocą specyfikatora, który jest jednym ze słów kluczowych **extern** lub **static** (trzecie, **register**, nie jest już używane, choć pozostaje słowem kluczowym). Określa on sposób, w jaki zmiennej ma być przydzielana pamięć.

7.3.1 Zmienne statyczne

Zmienne statyczne deklarowane są ze specyfikatorem **static**. Mogą to być zarówno zmienne globalne, jak i lokalne.

Zmienna statyczna jest tworzona tylko raz i inicjowana po utworzeniu zerem odpowiedniego typu. Jeśli jest to zmienna lokalna, to tworzona jest, gdy przepływ sterowania przechodzi po raz pierwszy przez jej deklarację. Jej czas życia to okres od utworzenia do końca programu. Podobnie zmienna statyczna globalna tworzona jest raz i trwa do końca programu. Jaka jest zatem różnica między zmienną globalną niestatyczną a globalną statyczną? Przecież zmienne globalne (zdefiniowane bez specyfikatora **static**) też tworzone są raz i trwają do końca programu. Różnica jest taka, że jeśli zmienne globalne zadeklarujemy z atrybutem **static**, to będą widoczne w danym module (jednostce translacji) i *tylko* w nim: nie można się do nich odwołać z innego modułu nawet poprzez użycie słowa kluczowego **extern** (patrz dalej). Mówimy, że takie zmienne globalne *nie* są eksportowane. Obecnie nie zaleca się stosowania statycznych zmiennych globalnych; ten sam efekt można uzyskać poprzez użycie *przestrzeni nazw* — będziemy o tym mówić w rozdz. 23.2 na stronie 510.

Zmienna statyczna lokalna, mimo że widoczna jest tylko wewnątrz funkcji lub bloku, w którym została zadeklarowana, *nie* jest usuwana po wyjściu sterowania z tej funkcji (bloku). Po powrocie sterowania do tej samej funkcji wartość tej zmiennej

zostanie zachowana — jest to bowiem wciąż fizycznie ta sama zmienna, w tym samym wciąż miejscu w pamięci komputera.

Zauważmy, co ilustruje poniższy program, że zmienne statyczne, tak lokalne jak i globalne, też można wewnątrz funkcji przesłaniać. Zarówno przesłaniane, jak i przesłaniające zmienne statyczne zachowują wtedy swą tożsamość pomiędzy wywołaniami funkcji:

P34: ***stat.cpp*** Zmienne statyczne

```
#include <iostream>
using namespace std;

int stat = 10;

void fun() {
    static int stat;
    cout << "local  stat " << stat++ << endl;
    cout << "global stat " << ::stat << endl;
    {                                                    ①
        static int stat;
        cout << "block  stat " << stat--
                                 << "\n\n";
    }
}

int main() {
    fun();
    fun();
    fun();
}
```

Z wyników tego programu

```
stat lokalne  0
stat globalne 10
stat w bloku  0

stat lokalne  1
stat globalne 10
stat w bloku  -1

stat lokalne  2
stat globalne 10
stat w bloku  -2
```

widzimy, że wszystkie trzy zmienne stat są niezależnymi, trwałymi zmiennymi. W szczególności, ostatnia z nich, wewnątrz nieco sztucznego bloku rozpoczynającego sie w li-

nii ①, również zachowuje swoją wartość pomiędzy wywołaniami funkcji, niezależnie od zmiennej stat zadeklarowanej w pierwszej linii funkcji **fun**.

Oczywiście, nic nie stoi na przeszkodzie, aby w różnych funkcjach używać lokalnych zmiennych statycznych o tej samej nazwie — będą one od siebie całkowicie niezależne. W poniższym przykładzie istnieją trzy statyczne zmienne licznik: dwie lokalne w dwóch różnych funkcjach i jedna globalna.

P35: ***static.cpp*** Statyczne liczniki wywołań

```
#include <iostream>
using namespace std;

int licznik;

void fun1() {
    static int licznik;
    licznik++;   // lokalna
    ::licznik++; // globalna
    cout << "Wywolan   fun1: " << licznik << endl;
}

void fun2() {
    static int licznik;
    licznik++;   // lokalna
    ::licznik++; // globalna
    cout << "Wywolan   fun2: " << licznik << endl;
}

int main() {
    fun1(); fun1(); fun2(); fun1(); fun2();
    cout << "Wywolan fun1/2: " << licznik << endl;
}
```

Wynikiem programu jest

```
Wywolan   fun1: 1
Wywolan   fun1: 2
Wywolan   fun2: 1
Wywolan   fun1: 3
Wywolan   fun2: 2
Wywolan fun1/2: 5
```

Lokalne zmienne licznik zliczają liczbę wywołań funkcji, w której są zadeklarowane, a globalna zmienna licznik zlicza liczbę wywołań obu funkcji.

7.3.2 Zmienne zewnętrzne

Zmienne zewnętrzne deklarowane są ze specyfikatorem **extern**, np.:

```
extern double x;
```

Deklaracja musi się znajdować poza funkcjami i klasami, a więc w zakresie globalnym. Stanowi ona informację dla kompilatora, że zmienna ta jest lub będzie *zdefiniowana* w innym pliku/module. Definicja z kolei może pojawić się tylko raz, w jednym z modułów składających się na program, jako definicja zmiennej globalnej *bez* specyfikatora **extern**. Zatem deklaracja ze specyfikatorem **extern** to przypadek, gdy deklaracja zmiennej *nie* jest związana z jej definicją: żadna zmienna nie jest tworzona, tzn. pamięć nie jest przydzielana.

Ta sama zmienna może być zadeklarowana (w identyczny sposób, tzn. z zachowaniem zgodności typu) wiele razy — również wielokrotnie w jednym pliku — ale *zdefiniowana* może być tylko raz. Zmiennej tylko deklarowanej (więc ze specyfikatorem **extern**) nie wolno inicjować: deklaracja nie wiąże się z przydziałem pamięci, więc nie byłoby gdzie wpisać wartości inicjującej. Większość kompilatorów potraktuje

```
extern double x = 1.5;
```

tak jak po prostu

```
double x = 1.5;
```

gdyż deklaracja połączona z inicjalizacją zmusza kompilator do zaalokowania pamięci, a więc do *zdefiniowania* zmiennej. Słowo **extern** zostanie wtedy zignorowane.

To, co powiedzieliśmy dotyczy zmiennych. Jeśli deklarujemy w module funkcję, której definicja podana jest w innym module, to słowa kluczowego **extern** nie trzeba używać (kiedyś było trzeba). Szerzej o funkcjach w rozdz. 11 na stronie 155.

Rozważmy program składający się z dwóch plików: pierwszy z nich zawiera funkcję **main**

P36: *exter1.cpp* Zmienne "extern". Plik 1

```
#include <iostream>
using namespace std;

double x1 = 11;
extern double x2;
void func();

int main()
{
    cout << "main: x1 = " << x1 << endl;
    cout << "main: x2 = " << x2 << endl;
    func();
}
```

a drugi funkcję **func**

P37: *exter2.cpp* Zmienne "extern". Plik 2

```
#include <iostream>
using namespace std;

extern double x1;
double x2 = 22;

void func()
{
    cout << "func: x1 = " << x1 << endl;
    cout << "func: x2 = " << x2 << endl;
}
```

Zmienna x1 jest definiowana jako zmienna globalna w pliku pierwszym, a w drugim tylko deklarowana. Odwrotnie zmienna x2 — ta jest definiowana w pliku drugim, a deklarowana w pliku pierwszym. Jak widzimy z wyniku

```
cpp> g++ -o extern exter1.cpp exter2.cpp
cpp> ./extern
main: x1 = 11
main: x2 = 22
func: x1 = 11
func: x2 = 22
```

po zlinkowaniu programu zarówno w funkcji **main**, jak i funkcji **func** widoczne są te same zmienne x1 i x2. Sama funkcja **func** nie musiała być deklarowana w pierwszym pliku jako **extern** (choć mogła być), gdyż funkcje domyślnie *są* eksportowane.

7.4 Modyfikatory typów

Modyfikatory **volatile** i **const** są tzw. modyfikatorami typu (ang. *type qualifier*): nie wpływają na sposób przydziału pamięci czy linkowania, ale zmieniają typ deklarowanej zmiennej (choć reprezentacja bitowa pozostaje taka sama).

7.4.1 Zmienne ulotne

Zmienne ulotne deklarowane są z modyfikatorem **volatile**, np.:

```
volatile double x;
```

Modyfikator **volatile** oznacza, że zmienna ta może ulec zmianie „bez wiedzy” programu, np. na skutek obsługi przerwań generowanych przez urządzenia zewnętrzne

(czujniki, mierniki itd.) czy obsługi sygnałów. Zadeklarowanie zmiennej jako ulotnej stanowi wskazówkę dla kompilatora, że przed każdym użyciem należy wartość tej zmiennej odczytać na nowo z pamięci, a każde na nią przypisanie (modyfikacja jej wartości) musi być fizycznie wykonane przed wykonaniem kolejnych instrukcji programu. Normalnie bowiem, widząc, że zmienna nie ulega zmianie pomiędzy pewnymi dwoma punktami programu, kompilator może zachować jej wartość w rejestrze i przy drugim użyciu nie odczytywać jej już z pamięci.

Stosowanie **volatile** nigdy nie bywa potrzebne w „normalnych" programach. Pamiętać też warto, że deklarowanie zmiennych jako ulotnych obniża wydajność programu: utrudnia kompilatorowi optymalizację i powoduje narzuty czasowe podczas wykonania.

Nie jest również prawdą, że używanie zmiennych ulotnych przydaje się przy pisaniu programów wielowątkowych, bowiem szczegóły ich użycia są (i mają prawo być) zależne od implementacji i jako takie są nieprzenośne (zamiast zmiennych **volatile** należy wtedy używać zmiennych **atomowych**).

7.4.2 Stałe

Stałe (zmienne ustalone) są używane często i w wielu kontekstach.

Zmienną każdego typu, zarówno „zwykłego", jak i wskaźnikowego, można ustalić deklarując ją z modyfikatorem typu **const** przed lub za nazwą typu.

Wartości stałej (zmiennej ustalonej) nie można zmienić po jej utworzeniu.

Jedynym sposobem na nadanie stałej jej wartości jest inicjalizacja podczas jej definiowania, bezpośrednio w instrukcji deklaracyjnej:

```
const double PI = 3.1415926536;
```

Błędem byłaby natomiast inicjalizacja *po* utworzeniu

```
const double PI;
PI = 3.1415926536; // NIE !!!
```

Po utworzeniu i inicjalizacji wartości stałej nie można zmienić.

Niektóre stałe mogą mieć nadane wartości już na etapie kompilacji. Aby tak było, stała musi być zainicjowana wyrażeniem, którego wartość może być obliczona na etapie kompilacji, a więc składającego się z literałów, innych stałych wyliczalnych na etapie kompilacji i prostych operacji arytmetycznych na liczbach całkowitych. Inne stałe wyliczane są dopiero podczas wykonania.

```
const int a = 2*11;
const int b = 5*a*a;
const int c = fun(a,b);
```

Zmienne a i b są tu stałymi kompilacji (i mogą, na przykład, być użyte do wymiarowania statycznej tablicy), natomiast c nie jest (choć jest typu **const int**), bo jej inicjowanie wiąże się z wywołaniem funkcji i zajdzie dopiero w czasie wykonania.

Zmienna zadeklarowana jako stała ma inny typ niż zmienna nieustalona. Tak więc typem zmiennej a z poprzedniego przykładu jest nie **int**, ale **const int** — są to *różne* typy. Na przykład poniższy fragment jest nielegalny

```
const int i = 25;
int *pi = &i; // NIE !!!
```

gdyż typem '&i' jest *wskaźnik do ustalonej zmiennej całkowitej*, a zadeklarowanym typem zmiennej pi jest *wskaźnik do (zwykłej) zmiennej całkowitej*.

Przypisanie odwrotne jednak *jest* legalne: można do wskaźnika do ustalonej zmiennej przypisać adres zmiennej nieustalonej. Takie zachowanie jest korzystne i często się stosuje przy przekazywaniu argumentów wskaźnikowych do funkcji: wysyłamy do funkcji adres zmiennej nieustalonej, ale odpowiednim parametrem funkcji jest wskaźnik do stałej. Zapobiega to przypadkowym zmianom wartości zmiennej wskazywanej wewnątrz funkcji. W następującym przykładzie

P38: *stale.cpp* Stałe

```
1  #include <iostream>
2  using namespace std;
3
4  int fun(const int * pi) {
5      //*pi = 2 * (*pi); // NIE !!!
6      return *pi;
7  }
8
9  int main() {
10     int i = 2;
11     int res = fun(&i);                    ①
12     cout << "res = " << res << endl;
13 }
```

wykomentowana linia byłaby nielegalna. Do funkcji posłaliśmy co prawda adres zwykłej zmiennej typu **int** (①), ale funkcja o tym nie wie; parametr zadeklarowany jest jako **const int***, więc kompilator nie zgodzi się na zmianę wartości zmiennej wskazywanej przez pi. Tak więc wewnątrz funkcji zmienna skojarzona z parametrem typu ustalonego jest traktowana jako stała niezależnie od tego czy odpowiedni argument wywołania był adresem ustalonej zmiennej czy nie.

Z drugiej strony, takie zachowanie może prowadzić do pewnych niejasności. Na przykład w programie

P39: *stale1.cpp* Jeszcze o stałych

```
#include <iostream>
using namespace std;

int main() {
    int i = 2;
    const int *pi = &i;                    ①

    i = 2*i;                               ②
    cout << "  i = " <<   i << endl;
    cout << "*pi = " << *pi << endl;
    //*pi = -1;
}
```

zmienna i nie jest ustalona. Ale pi (linia ①) jest wskaźnikiem typu **const int***, czyli zmienna wskazywana przez pi jest traktowana jako ustalona. Tą zmienną wskazywaną jest ta sama zmienna i. Czy zatem i jest, czy nie jest ustalona? Otóż

- jest, jeśli odwołujemy się do niej poprzez wskaźnik pi — dlatego wykomentowana ostatnia linia jest nielegalna.
- nie jest, jeśli odwołujemy się do tej samej zmiennej poprzez nazwę i, wtedy wartość tej zmiennej *możemy* zmienić (linia ②), bo ta nazwa wcale nie była zadeklarowana jako nazwa stałej.

Dlatego program jest prawidłowy i drukuje

```
  i = 4
*pi = 4
```

Pewnej uwagi wymaga staranne rozróżnianie wskaźników do stałych od stałych wskaźników. W przykładach, które rozpatrywaliśmy, to wskazywana zmienna była ustalona, a nie wskaźnik. Zatem to wartości wskazywanej zmiennej nie można było zmienić; nic nie stało jednak na przeszkodzie, aby zmienić wartość wskaźnika, czyli zapisany w nim adres. Ilustruje to następujący przykład:

P40: *stale2.cpp* Stałe zmienne wskazywane

```
#include <iostream>
using namespace std;

int main() {
    const int i = 2, j = 3;

    const int *p = &i;                ①
    cout << "*p = " << *p << endl;

```

```
10      p = &j;
11      cout << "*p = " << *p << endl;
12  }
```

Obie zmienne i oraz j są ustalone. Wskaźnik p jest wskaźnikiem do stałej. Natomiast sam wskaźnik nie jest ustalony, a więc może wskazywać zarówno na i jak i na j — jego wartość (przechowywany adres) w trakcie wykonywania programu zmienia się. Zauważmy, że w linii ① nie była wcale potrzebna inicjalizacja: linia ta definiuje wskaźnik, a ten ustalony nie jest.

Można jednak ustalić wskaźnik. Modyfikator **const** stawiamy wówczas *za* znakiem gwiazdki. A zatem

```
const int *pi;
```

lub

```
int const *pi;
```

to deklaracja/definicja zwykłego (nieustalonego) wskaźnika do stałej całkowitej. Dlatego nie wymaga inicjowania (zauważmy, że **const** może być umieszczone przed lub za nazwą typu **int**). Natomiast

```
int i;
int *const pi = &i;
```

deklaruje w drugim wierszu *ustalony wskaźnik*. Zainicjowanie takiego wskaźnika jest więc konieczne. Zainicjowanie polega na wpisaniu do wskaźnika adresu istniejącej zmiennej (w przykładzie jest to zmienna i): wartości wskaźnika pi nie będzie można już zmienić, czyli nie będzie można wpisać do niego adresu innej zmiennej — będzie on zawsze wskazywał na zmienną i. Ale, co ważne, sama zmienna wskazywana ustalona tu nie jest — jej wartość *można* zmienić również poprzez nazwę wskaźnika (np. `*pi=7;`).

Rozpatrzmy następujący program, ilustrujący dotychczasową dyskusję (program ten jest błędny i nie kompiluje się!):

P41: ***stalwsk.cpp*** Stałe wskaźniki i wskaźniki do stałych

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int *p, k = 5, m = 7;
6
7      const int stala = 3;        // stała
8      const int *q = &k;          // q - wskaźnik na stałą
9      int *const r = &k;          // r - stały wskaźnik
10     const int tab[] = {1,2,3}; // tablica stałych
11
```

```
12      p = &stala;      ①
13      stala = 1;       ②
14      *q = m;          ③
15      q = &m;          ④
16      r = &m;          ⑤
17      k = 10;          ⑥
18      tab[1] = 9;      ⑦
19  }
```

Przeanalizujmy poszczególne fragmenty:

- Linia ① jest nielegalna, bo p ma typ **int***, a &stala ma typ wskaźnik do stałej (**const int***).
- Linia ② to próba zmiany wartości zmiennej ustalonej — oczywiście nielegalna.
- Linia ③ to próba zmiany wartości zmiennej, co prawda nieustalonej (q wskazuje na k) ale poprzez odwołanie się do tej zmiennej za pomocą wskaźnika zadeklarowanego jako wskaźnik do stałej.
- Linia ④: choć q jest wskaźnikiem na stałą, to sam wskaźnik nie jest ustalony: zatem jego zawartość możemy zmienić.
- Linia ⑤ jest nielegalna, gdyż r jest ustalonym wskaźnikiem i wskazuje na zmienną k. Tego wskazania zmienić nie można.
- Linia ⑥ jest legalna: r jest ustalone i cały czas wskazuje na k, ale samą zmienną k można zmienić, gdyż nie jest stałą.
- Linia ⑦: tab[1] jest stałą, bo jest elementem tablicy stałych — nie może być zmienione.

Można też posługiwać się ustalonym wskaźnikiem do stałej:

```
int i;
const int *const pi = &i;
```

Oczywiście, wskaźnik musimy wtedy od razu zainicjować, choć, jak mówiliśmy, niekoniecznie adresem stałej. W takim przypadku nie wolno modyfikować ani zmiennej wskazywanej przez ten wskaźnik (w każdym razie nie wolno jej zmienić poprzez ten wskaźnik), ani samego wskaźnika (czyli adresu w nim zapisanego); na przykład w programie

P42: *stalstal.cpp* Stałe wskaźniki do stałych

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
```

```
5     int i = 1, m = 2;
6     const int *const pi = &i;
7
8     cout << "Przed *pi = " << *pi << endl;
9     i = 3;                                          ①
10    cout << "Po    *pi = " << *pi << endl;
11
12    *pi = 4; // NIE: pi jest wskaźnikiem do stałej
13    pi = &m; // NIE: pi jest ustalonym wskaźnikiem
14 }
```

można zmienić wartość zmiennej i odwołując się do niej poprzez nazwę i, gdyż nie jest to stała (linia①). Natomiast nie jest legalna ani próba zmiany tej samej wartości poprzez odwołanie się do niej za pomocą wskaźnika pi, ani próba zmiany wskazania (adresu) w zmiennej pi (dwie ostatnie linie).

W nowym standardzie C++11 istnieje „silniejsza" wersja określenia stałości — zamiast **const** używamy przy definiowaniu zmiennej słowa kluczowego **constexpr**. Znaczy to, że definiujemy coś (wartość całkowitą, zmiennoprzecinkową, obiektową), czego wartość musi dać się obliczyć już na etapie kompilacji. Obliczana wartość może zależeć od wartości literałów, ale też od wartości innych zmiennych, jeśli również są wyrażeniami stałymi (**constexpr**). Kompilator sprawdzi, czy rzeczywiście potrafi taką wartość wyliczyć i zgłosi błąd, jeśli jest to niemożliwe. W poniższym przykładzie

P43: ***constexpr.cpp*** Wartości *constexpr*

```
#include <iostream>
using namespace std;

int main() {
    constexpr int hourfee = 7;
    constexpr int tim = 5;

    int arr[10 + (tim-1)*hourfee];        ①

    cout << "number of elements in arr "
         << sizeof(arr)/sizeof(arr[0]) << endl;
}
```

wyrażenie użyte do wymiarowania definiowanej tablicy jest stałą kompilacji i rzeczywiście jest obliczane na etapie kompilacji — w przeciwnym przypadku program nie skompilowałby się, bo wymiar definiowanej tablicy statycznej musi być znany kompilatorowi (przynajmniej przy kompilacji z opcją '`-pedantic`').

Zobaczymy w następnych rozdziałach, że wartościami **constexpr** mogą być też wywołania funkcji, konstrukcje obiektów czy wywołania metod na rzecz obiektów, co było niemożliwe w starym standardzie.

7.5 L-wartości

Każda dana w programie musi oczywiście być gdzieś w pamięci komputera zapisana. Nie musi to jednak być segment pamięci przeznaczony na dane o dostępnych programowo adresach: mogą to być rejestry procesora lub specjalne obszary pamięci, gdzie tymczasowe zmienne są dynamicznie tworzone i usuwane.

Na przykład w takim fragmencie

```
int x, y = 1;
// ...
x = y + 1;
```

wynik dodawania '`y+1`', o wartości 2, niewątpliwie musi być gdzieś zapisany: pewnie ta wartość zostanie wpisana do zmiennej `x` po obliczeniu bezpośrednio z rejestru procesora. Tak czy owak, nie mamy na to ani szczególnego wpływu, ani dostępu do tej danej — możemy ją tylko natychmiast po obliczeniu gdzieś zapisać lub użyć do innych operacji. Co ważniejsze, po wykonaniu przypisania ta obliczona wartość najprawdopodobniej zostanie za chwilę zamazana, bo była tymczasowa i służyła wyłącznie do tego, aby można było przekopiować ją do zmiennej `x`. Zauważmy, że tego typu wartości mogą się pojawiać tylko jako wartości wyrażenia po *prawej* stronie przypisania.

Pamiętamy, że

> w instrukcji przypisania prawa strona przypisania mówi co policzyć, lewa zaś — gdzie zapisać wynik.

Zatem po lewej stronie przypisania mogą pojawiać się tylko wyrażenia identyfikujące dane o dostępnym adresie: ich dotychczasowa wartość nie ma znaczenia, bo właśnie w wyniku przypisania zostanie za chwilę zmieniona!

Prowadzi nas to do pojęcia **l-wartości** (ang. *lvalue*, od *left*: lewy) i **p-wartości** (ang. *rvalue*, od *right*: prawy).

> L-wartość to wyrażenie identyfikujące daną o określonym typie, która ma określony, dostępny w programie adres w pamięci.

Zatem np. nazwa zmiennej ustalonej (**const**) jest l-wartością, choć niemodyfikowalną. Identyfikator „zwykłej", nieustalonej zmiennej jest l-wartością modyfikowalną. Generalnie zatem, jeśli wyrażenie występuje po lewej stronie przypisania, to musi być l-wartością; stwierdzenie odwrotne nie jest prawdziwe.

> P-wartość to wyrażenie, które ma określoną wartość.

Zauważmy, że l-wartość jest też p-wartością, choć odwrotnie nie musi to być prawda: p-wartości stanowią znacznie szerszą klasę wyrażeń. Na przykład takie wyrażenie jak

'x+2' ma wartość, więc jest p-wartością (jeśli x jest zdefiniowane); wyrażenie to jednak nie jest l-wartością z powodów, o których mówiliśmy.

Wymieńmy zatem wyrażenia, które mogą być l-wartościami (znaczenie niektórych wyjaśnimy w następnych rozdziałach):

- Identyfikator zmiennej, również kwalifikowany operatorem zakresu, np. ::x.
- Rezultat funkcji o referencyjnym typie zwracanym.
- Wyrażenie składające się z przypisania (jako całość); np. a=c+b ma wartość a *po* przypisaniu i adres &a.
- Wyrażenie z operatorem *przedrostkowego* zwiększenia lub zmniejszenia (ale *nie* przyrostkowego); zatem ++k jest, a k++ *nie jest* l-wartością).
- Wyłuskanie elemenu tablicy, np. p[k].
- Wyrażenie z wyłuskaniem wartości (dereferencji) , na przykład *p.
- Wybór składowej niefunkcyjnej klasy lub obiektu, na przykład obiekt.skladowa lub wsk->skladowa).
- Rezultat operatora warunkowego, jeśli drugi i trzeci argument są l-wartościami; na przykład wyrażenie a>0?++i:++j jest, natomiast a>0?i++:j++ nie jest l-wartością.
- Wynik konwersji do typu referencyjnego, np. (int&)k.
- Wynik operatora, jeśli prawy argument tego operatora jest l-wartością.

Na przykład w ostatniej linii

```
int x = 2, y = x + 1;
// ...
int j = x = y;
```

wyrażenie x=y jest l-wartością (i p-wartością) o wartości równej wartości x po przypisaniu. Skoro jest l-wartością, to może też wystąpić po lewej stronie przypisania:

```
(x=y-2) = 5;
```

Wyrażenie (x=y-2) ma wartość x po przypisaniu (czyli 1) i adres tejże zmiennej x — zatem przypisanie wartości 5 nastąpi właśnie do zmiennej x (wartość 1 zostanie zamazana).

Po następującej instrukcji natomiast

```
int *p = &( x>y ? x : y);
```

wskaźnik p będzie zawierał adres większej ze zmiennych x i y. Inne przykłady:

P44: *lval.cpp* L-wartości

```
#include <iostream>
using namespace std;

int& razydwa(int& m) {
    static int licz;
    cout << " W funkcji: licz = " << licz << endl;
    m *= 2;
    return licz;
}

void printTab(const int *tab, int size) {
    cout << "[ ";
    for (int i = 0; i < size; i++)
        cout << tab[i] << " ";
    cout << "]" << endl;
}

int main() {
    int i = 1, j = 2, k = 3;

    // przypisanie jako l-nazwa
    (i=j) = k;
    cout << " A: i = " << i << " j = " << j
         <<    " k = " << k << endl;           // 3,2,3

    int tab[] = {1,2,3,4}, *p = tab;
    cout << " B: przed "; printTab(tab,4);
    *++++++p = 8;
    cout << " B:    po "; printTab(tab,4);

    // teraz p wskazuje na ostatni element!
    cout << " C: ++*----p = " << ++*----p << endl; // 3
    cout << " C:   tab "; printTab(tab,4);

    // konwersja jako l-nazwa
    int m = 7;
    razydwa( (int&)m=8 )++;  // konwersja niepotrzebna!
    cout << "D1: m = " << m << endl;
    razydwa(m)++;
    cout << "D2: m = " << m << endl;
    int n = razydwa(m) = 10;
    cout << "D3: m = " << m << endl;
    cout << "D4: n = " << n << endl;

```

```
45      // operator przecinkowy
46      k = (i=1, j=2) + 1;
47      cout << " E: i = " << i << " j = " << j
48           <<     " k = " << k << endl;          // 1,2,3
49
50      // selektor jako l-nazwa
51      (k > 2 ? i : j) = 5;
52      cout << " F: i = " << i << " j = " << j
53           <<     " k = " << k << endl;          // 5,2,3
54  }
```

Wynik tego programu to:

```
 A: i = 3 j = 2 k = 3
 B: przed [ 1 2 3 4 ]
 B:    po [ 1 2 3 8 ]
 C: ++*----p = 3
 C:   tab [ 1 3 3 8 ]
 W funkcji: licz = 0
D1: m = 16
 W funkcji: licz = 1
D2: m = 32
 W funkcji: licz = 2
D3: m = 64
D4: n = 10
 E: i = 1 j = 2 k = 3
 F: i = 5 j = 2 k = 3
```

W linii 22 przypisanie `(i=j)` pełni rolę l-wartości. Odpowiednim adresem jest adres zmiennej i, zatem to do tej zmiennej przypisana zostanie wartość k. Natomiast wartość zmiennych j i k pozostanie niezmieniona. Zauważmy, że jest to co innego niż '`i=j=k`'. Ta instrukcja byłaby równoważna '`i=(j=k)`' i zmieniłaby wartości zarówno zmiennej i jak i zmiennej j.

W linii 26 definiujemy tablicę czteroelementową i jej adres (wartość zmiennej tab) przypisujemy do wskaźnika p. Spójrzmy na nieco dziwną konstrukcję `*+++++++p` z linii 28. Najpierw trzy razy zwiększamy wartość p. Ponieważ zmienna p jest wskaźnikowa, więc jej zwiększenie oznacza to samo co `((p+1)+1)+1` i sprowadza się do zmiany wartości p — najpierw p wskazywała na tab[0], po tej operacji wskazuje na tab[3], czyli na ostatni element tablicy. Wyłuskanie (dereferencja) tej zmiennej za pomocą operatora wyłuskania wartości (gwiazdka) daje nam l-wartość zmiennej tab[3] i to do tej zmiennej przypisujemy wartość 8, o czym świadczy druga z linii wydruku oznaczonych literą 'B'. Efektem ubocznym jest fakt, że teraz p wskazuje na tab[3].

Podobna konstrukcja użyta została w linii 32. Najpierw dwa razy zmniejszamy wartość zmiennej p — ponieważ wskazywała ona na ostatni, czwarty element tablicy tab, to po tej operacji wskazuje na element drugi, czyli na tab[1]. Teraz dokonujemy dereferencji: wyrażenie `*---p` jest zatem nazwą drugiego elementu tablicy. Wartość

tego elementu (który jest typu **int**, a nie typu wskaźnikowego!) zwiększamy o jeden za pomocą operatora zwiększania. Zatem drugi element tablicy zwiększy się o jeden, co widać z linii wydruku oznaczonych literą 'C'. Efektem ubocznym będzie to, że teraz wskaźnik p wskazuje na ten właśnie, drugi element tablicy.

W linii 37 w argumencie wywołania funkcji **razydwa** dokonujemy jawnej konwersji wartości przypisania `m=8`, a jest nią l-wartość zmiennej m, do typu **int&**, bo taki jest typ argumentu funkcji. Ta konwersja nie jest tu wymagana, gdyż należy do konwersji standardowych wykonywanych w razie potrzeby automatycznie przez kompilator.

Ponieważ do funkcji posłaliśmy referencję do zmiennej m, operacje, jakie ta funkcja wykonuje na odebranej poprzez argument zmiennej, wykonywane są na oryginale: z wydruku widzimy, że w funkcji **main** wartość m zmienia się po każdym wywołaniu funkcji (linie wydruku oznaczone literą 'D').

Jak widać z definicji funkcji **razydwa**, zwraca ona referencję do lokalnej zmiennej statycznej licz. Zmienna licz jest co prawda lokalna, ale *istnieje* po powrocie z funkcji, ponieważ jest statyczna. Zwracana jest referencja do tej właśnie zmiennej, zatem `razydwa(m)` jest inną nazwą zmiennej licz. Dlatego jest l-wartością i w funkcji **main** możemy zmienić wartość licz, np. poprzez zastosowanie operatora zwiększenia (linie 37 i 39). Może też pojawić się po lewej stronie przypisania, jak w linii 41.

W linii 46 wyrażenie '`(i=1, j=2)`' jest l-wartością zmiennej j po przypisaniu `j=2`, zatem do jej wartości (równej 2) dodana zostanie jedynka i wynik przypisany do k.

W linii 51 selektor `(k>2?i:j)` jest l-wartością, bo są l-wartościami dwa ostatnie jego argumenty i i j. W naszym przykładzie, ponieważ warunek `k>2` jest spełniony, całe wyrażenie `(k>2?i:j)` jest l-wartością zmiennej i i to do tej zmiennej nastąpi przypisanie wartości 5.

— 8 —

Instrukcje

W tym rozdziale omówimy instrukcje języka C++.

PODROZDZIAŁY:

8.1 Rodzaje instrukcji

Instrukcje (ang. *statement*) języka C++ możemy generalnie podzielić na instrukcje:

- deklaracyjne
- instrukcję pustą
- grupujące
- wyrażeniowe
- warunkowe (**if**, **if...else**)
- wyboru (**switch**)
- iteracyjne (**for**, **while**, **do**)
- zaniechania (**break**)

- kontynuowania (**continue**)
- skoku (**goto**)
- powrotu (**return**)
- obsługi wyjątków (**try**, **throw**, **catch**)

Instrukcje pojedyncze kończą się średnikiem, grupujące — zamykającym nawiasem klamrowym. Instrukcje mogą być poprzedzone etykietą.

8.2 Etykiety

Każda instrukcja czynna (a więc nie deklaracja) może być poprzedzona **etykietą**. Etykieta to dowolny identyfikator i następujący po nim dwukropek. Instrukcją opatrzoną etykietą może, w szczególności, być instrukcja pusta. Etykieta może być wykorzystana do wykonania skoku za pomocą instrukcji **goto** (patrz dalej), jak w poniższym przykładzie.

P45: *labincpp.cpp* Skok do instrukcji z etykietą

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int tab[2][2][2]{{{1,2},{3,4}},{{5,6},{7,8}}};
6      bool jest = false;
7      for (int i = 0; i < 2; i++)
8          for (int j = 0; j < 2; j++)
9              for (int k = 0; k < 2; k++)
10                 if (tab[i][j][k] == 5) {     ①
11                     jest = true;
12                     goto LAB;
13                 }
14     LAB:if(jest)                             ②
15         cout << "5 jest w tablicy" << endl;
16     else
17         cout << "5 nie wystepuje w w tablicy" << endl;
18 }
```

W programie tym, wewnątrz zagnieżdżonej pętli **for** poszukiwany jest element trzywymiarowej tablicy liczbowej, który jest równy 5. Jeśli zostanie znaleziony (①), to zmienną jest ustawiamy na **true** i, dzięki instrukcji skoku, wychodzimy z zewnętrznej pętli — przepływ sterowania przechodzi do linii ② z etykietą LAB:. Omówiona dalej instrukcja zaniechania **break** spowodowałaby wyjście tylko z pętli wewnętrznej (a instrukcji **break** z etykietą, znanej z Javy, w C/C++ *nie* ma).

Innym rodzajem etykiet są etykiety w instrukcji **switch**. Jest ich tylko dwie: **case** i **default** (rozdz. 8.8, str. 108).

8.3 Deklaracje

W tym podrozdziale mówimy tylko o deklaracjach zmiennych. Deklaracje funkcji omówimy w rozdz. 11.2 na stronie 156.

Celem deklaracji zmiennej jest określenie jej typu i atrybutów. Deklaracja najczęściej jest też połączona z definicją, tzn. przydzieleniem zmiennej miejsca w pamięci operacyjnej. Ogólna postać deklaracji jest następująca:

```
modyfikatory Typ lista_ident;
```

gdzie:

- **modyfikatory** (które są opcjonalne) określają atrybuty deklarowanych zmiennych (np. **const**, **static**). Modyfikatorów może być kilka lub może nie być ich wcale. Jeśli jest ich kilka, to nie oddziela się ich przecinkami. Omówiliśmy je w rozdziale poprzednim.
- **Typ** jest określeniem typu bazowego deklarowanych zmiennych (np. **double**, **string**, **unsigned short** itd. — patrz rozdz. 4).
- **lista_ident** jest listą, oddzielonych przecinkami, identyfikatorów deklarowanych zmiennych. Po każdym identyfikatorze można umieścić inicjator za pomocą znaku równości i następującego po nim wyrażenia o typie wartości przypisywalnym do typu deklarowanej zmiennej lub za pomocą składni *brace-init* (z nawiasami klamrowymi).

Przykłady:

```
double x{17.5}, y, z = 1;
static const double PI{3.14};
extern double PIHALF;
const double * const pPi = &PI;
```

Trzecia linia to przykład deklaracji bez definicji. Oczywiście zmienna PIHALF musi być zdefiniowana w jakimś innym module programu.

8.4 Instrukcja pusta

Instrukcja pusta (ang. *null statement*) składa się z samego znaku średnika. Jej wykonanie nie powoduje żadnych skutków, ale czasem bywa potrzebna ze względów składniowych. Na przykład w procedurze sortowania szybkiego (*quick sort*) poszukuje się indeksu pierwszego elementu tablicy nie mniejszego od pewnej zmiennej v za pomocą pętli

```
while (a[++i] < v);
```

gdzie ciało pętli jest puste. Składnia jednak wymaga instrukcji po warunku logicznym w nawiasie okrągłym; rolę tej instrukcji pełni więc instrukcja pusta, zaznaczona średnikiem. Gdyby jej nie było, instrukcja następująca za pętlą zostałaby błędnie potraktowana jako ciało tej pętli!

8.5 Instrukcja grupująca

Instrukcje grupujące (ang. *compound statement, grouping statement*) stosuje się, gdy składnia języka wymaga wystąpienia w pewnym miejscu programu dokładnie jednej instrukcji, a tymczasem czynności, które mają być wykonane przez program za pomocą jednej instrukcji zapisane być nie mogą (lub prowadziłoby to do nieczytelnych konstrukcji). Istnieje zatem w języku możliwość potraktowania wielu instrukcji jako jednej instrukcji złożonej (grupującej). Ma ona postać ujętej w nawiasy klamrowe sekwencji instrukcji, z których każda może być instrukcją pustą (sam średnik), niepustą pojedynczą (a więc zakończoną średnikiem) lub również instrukcją grupującą (ujętą w nawiasy klamrowe). Po nawiasie klamrowym kończącym instrukcję grupującą średnika nie stawiamy. Przykładem instrukcji grupującej jest definicja funkcji.

Pamiętać trzeba, o czym już mówiliśmy, że fragment programu ujęty w nawiasy klamrowe tworzy *blok*. Zmienne zadeklarowane wewnątrz takiego bloku tworzonego przez instrukcję grupującą są lokalne dla tego bloku: po wyjściu sterowania z takiej instrukcji zmienne zadeklarowane wewnątrz nie są już znane (i w ogóle nie istnieją; są ze stosu usuwane). Tak więc

```
{
    int i = 5;
    {
        int k = fun(i);
        i += k;
    }
    cout << "i=" << i << endl;
}
```

jest jedną instrukcją grupującą, złożoną z trzech instrukcji, z których jedna jest również instrukcją grupującą. Po wykonaniu tej instrukcji zmienne `i` i `k` nie istnieją i, w szczególności, zmienne o tej nazwie mogą być zadeklarowane w dalszej części ciała funkcji, w której instrukcja ta wystąpiła (zmienna `k` jest usuwana już po wyjściu z wewnętrznego bloku, przed ostatnią linią).

8.6 Instrukcja wyrażeniowa

Każde wyrażenie, a więc grupa leksemów, która posiada wartość, może być traktowana jako samodzielna instrukcja, tzw. instrukcja wyrażeniowa (ang. *expression statement*). Instrukcja taka jest wykonywana poprzez opracowanie wyrażenia i obliczenie jego wartości. Ta wartość w pewnych sytuacjach może być zignorowana, wtedy zatem instrukcja taka ma sens, jeśli powoduje jakieś skutki uboczne (jeśli nie powoduje żadnych skutków ubocznych, to może być przez kompilator całkowicie zignorowana). Instrukcjami wyrażeniowymi mogą być przypisania, operacje zwiększenia (inkrementacji) lub zmniejszenia (dekrementacji), wywołania funkcji, itd.

P46: ***wyrins.cpp*** Instrukcje wyrażeniowe

```
1 #include <iostream>
2 #include <cstdio>
3 using namespace std;
4
5 int main() {
6    int k = 7, m = 8;
7    ++k;                    ①
8    k+1;                    ②
9    k = 5;                  ③
10   printf("OK?");          ④
11   k > m ? ++k : --m;      ⑤
12   new double(3.5);        ⑥
13 }
```

W przykładzie powyższym instrukcją wyrażeniową jest instrukcja z linii ① (inkrementacja), ③ (przypisanie), ④ (wywołanie funkcji rezultatowej – **printf** zwraca **int**), ⑤ (selekcja bez przypisania wartości), ⑥ (utworzenia obiektu bez przypisania wynikowej wartości). Instrukcja z linii ② ('`k+1`') jest też instrukcją wyrażeniową, tyle że nie powodującą żadnych skutków — zostanie ona prawdopodobnie przez kompilator zignorowana (tego typu instrukcja byłaby w Javie nielegalna). Instrukcja z linii ⑥ nie ma sensu, bo tworzymy zmienną typu **double**, której adresu nie zapisujemy i nigdy już nie zdołamy jej usunąć, choć będzie całkowicie bezużyteczna. Kompilator nie może jednak takich instrukcji zignorować, bo w zasadzie jest możliwe, że wykonanie konstruktora powoduje jakieś skutki, zajścia których programista ma prawo się spodziewać.

8.7 Instrukcja warunkowa

Instrukcja warunkowa (ang. *conditional statement*) występuje w dwóch postaciach.

Prostsza to

```
if ( b ) instr
```

gdzie b jest wyrażeniem o wartości logicznej, a instr jest instrukcją. Pamiętamy przy tym, o czym już mówiliśmy, że w C/C++ również liczbowe wartości całkowite i wskaźnikowe mogą być traktowane jako wartości logiczne — cokolwiek różnego od zera jest traktowane jako **true**, a wartość zerowa jako **false**. Opracowanie instrukcji warunkowej w tej formie polega na obliczeniu wartości b a następnie:

- zakończeniu wykonywania całej instrukcji, jeśli obliczoną wartością b okazało się **false**,
- wykonaniu instrukcji instr, jeśli obliczoną wartością b okazało się **true**.

Zauważmy, że składnia mówi o *jednej* instrukcji instr. Jeśli zachodzi potrzeba wykonania (lub zaniechania wykonania) większej liczby instrukcji, to należy ująć je w nawiasy klamrowe, tak aby uczynić z nich jedną instrukcję złożoną (grupującą): np. wykonanie następującego fragmentu

```
if ( s != 0 )
    cerr << "Cos nie w porzadku! Konczymy." << endl;
    exit(1);
```

spowoduje *zawsze* zakończenie programu, bo instrukcja `exit` nie jest wcale „pod ifem”! Powinniśmy raczej napisać

```
if ( s != 0 ) {
    cerr << "Cos nie w porzadku! Konczymy." << endl;
    exit(1);
}
```

Innym często popełnianym błędem jest stawianie średnika zaraz za zamknięciem nawiasu okrągłego w instrukcji warunkowej

```
if ( s != 0 );
{
    cerr << "Cos nie w porzadku! Konczymy." << endl;
    exit(1);
}
```

Zauważmy, że kompilacja powyższego fragmentu powiedzie się. Po `if (...)` jest tu instrukcja, zgodnie ze składnią — jest nią mianowicie instrukcja pusta (średnik). Natomiast instrukcja grupująca, która po niej występuje, nie jest już „pod ifem” i wykona się zawsze, niezależnie od prawdziwości warunku ’`s != 0`’.

Inna pułapka związana jest z faktem, że w C/C++, jak mówiliśmy w rozdz. 4.4, wartość całkowita jest traktowana jak wartość logiczna: **false** odpowiada wartości zerowej, **true** każdej innej. Powoduje to niestety częste błędy (które w Javie zostałyby wychwycone przez kompilator). Na przykład, jeśli a i b są zmiennymi całkowitymi, to wartością *przypisania* ’`a=b`’ jest, zgodnie z zasadą obowiązującą zresztą i w Javie, wartość lewej strony *po* przypisaniu, czyli wartość b. I to ona zostanie zinterpretowana jako wartość logiczna wyrażenia `a=b` w instrukcji

```
if ( a = b ) instr          // prawdopodobnie źle !!
```

choć zwykle intencją programisty było raczej *porównanie*: sprawdzenie, czy wartość a jest czy nie jest równa wartości b. Jeśli tak, to powinna tu być użyta forma `a==b`.

W nowym standardzie, w nawiasie okrągłym tuż przed warunkiem, można umieścić *init-statement* (ze średnikiem na końcu). Może to być dowolone wyrażenie (coś, co ma wartość) albo deklaracja jednej lub więcej zmiennych tego samego typu. W tym drugim przypadku, zadeklarowane zmienne istnieją tylko w zakresie instrukcji **if** (również w jej gałęzi **else**). Na przykład, w poniższym fragmencie, zmienna delta może być użyta we wszystkich gałęziach instrukcji **if**, ale nie jest widoczna poza nią:

```
double a, b, c;
cout << "Enter three numbers: ";
std::cin >> a >> b >> c;

if (auto delta = b*b-4*a*c; delta > 0)
    cout << "Two roots, delta = " << delta << endl;
else if (delta < 0)
    cout << "No roots, delta = " << delta << endl;
else
    cout << "Two equal roots, delta = 0" << endl;
```

Najczęściej jest to zaleta, bo nie zaśmiecamy przestrzeni nazw nazwami zmiennych, które potrzebne są tylko lokalnie.

Druga forma instrukcji warunkowej to

```
if ( b ) instr1
else     instr2
```

gdzie, jak poprzednio, b jest wyrażeniem o wartości logicznej a instr1 i instr2 są instrukcjami — prostymi lub złożonymi. Opracowanie instrukcji warunkowej w tej formie polega na obliczeniu wartości b, a następnie:

- wykonaniu instrukcji instr1, jeśli obliczoną wartością b okazało się **true**,
- wykonaniu instrukcji instr2, jeśli obliczoną wartością b okazało się **false**.

I znów składnia mówi o jednej instrukcji instr1 i jednej instrukcji instr2. Jeśli zachodzi potrzeba użycia kilku instrukcji,to należy, jak poprzednio, zastosować instrukcję grupującą.

Każda z instrukcji instr1 i instr2 może z kolei też być instrukcją warunkową. Obowiązuje przy tym zasada, że frazie **else** odpowiada zawsze najbliższa poprzedzająca ją fraza **if**, po której nie było jeszcze frazy **else** i która jest zawarta w tym samym bloku na tym samym poziomie zagnieżdżenia co ta fraza **else**. Na przykład

```
if ( b1 ) instr0 else if ( b2 ) instr1 if ( b3 ) instr2
else instr3 else instr4
```

jest równoważne

```
if ( b1 )           // 1
    instr0
else                // 1
    if ( b2 ) {     // 2
        instr1
        if ( b3 )   // 3
            instr2
        else        // 3
```

```
            instr3
    }
    else            // 2
        instr4
```

gdzie tymi samymi liczbami w komentarzach oznaczone są odpowiadające sobie frazy **if** i **else**.

Inny przykład:

```
1   if ( val >= 0 )
2   {
3       if ( val > 9 ) cout << "Za duzo" << endl;
4   }
5   else
6       cout << "Za malo!" << endl;
```

Zauważmy, że w powyższym przykładzie konieczne było ujęcie instrukcji z linii trzeciej w blok, mimo że jest to jedyna instrukcja „pod ifem" z linii pierwszej. Gdybyśmy tego nie uczynili, fraza **else** z linii 5 zostałaby zinterpretowana jako para do frazy **if** z linii 3, a nie do **if** z linii pierwszej! Aby takich błędów uniknąć, należy zawsze starannie stosować wcięcia w pisanym kodzie.

8.8 Instrukcja wyboru (*switch*)

Instrukcja wyboru (ang. *switch statement*) w zasadzie zawsze może być zastąpiona instrukcjami warunkowymi, ale czasem czytelniej jest użyć właśnie instrukcji wyboru (co może być też efektywniejsze). Jej najbardziej ogólna postać to:

```
switch (wyr_calk) {
    case stala1: lista1
    case stala2: lista2
    // ...
    dafault: lista
}
```

gdzie wyr_calk jest wyrażeniem o wartości całkowitej, stala1, stala2, ..., są wyrażeniami *stałymi* o wartości całkowitej, a lista1, lista2, ..., są listami instrukcji (być może pustymi). Wyrażeniem stałym całkowitym może być tu liczba podana w postaci literału, nazwa całkowitej zmiennej **constexpr** lub wyrażenie całkowite składające się z tego typu podwyrażeń. Liczba fraz **case** może być dowolna. Stałe występujące w każdej z fraz **case** muszą być różne. Listy instrukcji mogą też być puste. Fraza **default** jest opcjonalna: jeśli występuje, to może wystąpić tylko raz, choć niekoniecznie na końcu.

Najpierw obliczane jest wyr_calk. Następnie, jeśli obliczona wartość jest równa wartości którejś ze stałych stala1, stala2, ..., to wykonywane są instrukcje ze *wszystkich* list instrukcji, poczynając od listy we frazie **case** odpowiadającej tej stałej. A więc

wykonywane są *nie tylko* instrukcje z listy w znalezionej frazie **case**, ale również ze wszystkich dalszych list!

Jeśli żadna ze stałych stala1, stala2, ..., nie jest równa wartości wyr_calk, a fraza **default** istnieje, to wykonywane są wszystkie instrukcje poczynając od tych we frazie **default**. Jeśli natomiast żadna ze stałych stala1, stala2, ..., nie jest równa wyr_calk, a fraza **default** nie istnieje, to wykonanie całej instrukcji wyboru uznaje się za zakończone.

Dowolną z instrukcji może być instrukcja zaniechania **break**. Jeśli sterowanie przejdzie przez tę instrukcję, to wykonanie całej instrukcji wyboru kończy się; podobnie będzie po napotkaniu **return** czy **goto**.

Instrukcję wyboru zilustrowano w poniższym programie; funkcja **sw** powoduje wypisanie różnej liczby gwiazdek w zależności od wartości argumentu: cztery gwiazdki dla argumentu 1, dwie dla argumentu 5, zero dla argumentu 2 lub 3 i trzy gwiazdki dla każdej innej wartości argumentu.

P47: ***switch.cpp*** Instrukcja wyboru 1

```
1  #include <iostream>
2  using namespace std;
3
4  void g( ) {
5      cout << '*';
6  }
7
8  void sw(int k) {
9      cout << k << ": ";
10     switch ( k ) {
11         default: g( );                          ①
12         case  5: g( ); g( );                    ②
13         case  3:
14         case  2: break;                         ③
15         case  1: g( ); g( ); g( ); g( );
16     }
17     cout << endl;
18 }
19
20 int main() {
21     sw(9);
22     sw(5);
23     sw(4);
24     sw(3);
25     sw(2);
26     sw(1);
27     sw(0);
28 }
```

Działanie programu widać z wyników

```
9: ***
5: **
4: ***
3:
2:
1: ****
0: ***
```

Zauważmy, że dla argumentów różnych od 1, 2, 3, 5 sterowanie przechodzi do instrukcji występującej we frazie **default** (linia ①) i przechodzi następnie do instrukcji we frazach **case** odpowiadających wartościom 5, 3 i 2. Dopiero w linii ③ napotykana jest instrukcja **break**, która kończy wykonywanie całej instrukcji wyboru. Dlatego drukowane są wtedy trzy gwiazdki: jedna w linii ① i dwie w linii ② programu.

Fraza **default** wcale nie musi, jak widać z programu, występować na końcu.

Jeśli jedną z instrukcji jest instrukcja **return**, to przejście przez nią sterowania spowoduje oczywiście również zakończenie wykonywania instrukcji wyboru i zakończenie wykonywania funkcji w której występuje. Funkcja **hexVal** w poniższym programie wyznacza wartość liczbową odpowiadającą cyfrze szesnastkowej podanej w postaci znaku lub dostarcza -1, jeśli podany znak nie odpowiada żadnej cyfrze szesnastkowej:

P48: *hex.cpp* Instrukcja wyboru (2)

```
#include <iostream>
using namespace std;

int hexVal(char c) {
    switch ( c ) {
        case '0': case '1': case '2':    ①
        case '3': case '4': case '5':
        case '6': case '7': case '8':
        case '9':
            return c - '0';              ②

        case 'a': case 'b': case 'c':
        case 'd': case 'e':
        case 'f':
            return 10 + c - 'a';         ③

        case 'A': case 'B': case 'C':
        case 'D': case 'E':
        case 'F':
            return 10 + c - 'A';         ④

        default: return -1;              ⑤
    }
```

```
24 }
25
26 int main() {
27     cout << "A = " << hexVal('A') << endl
28          << "f = " << hexVal('f') << endl
29          << "9 = " << hexVal('9') << endl
30          << "b = " << hexVal('b') << endl
31          << "Z = " << hexVal('Z') << endl;
32 }
```

W linii ① i trzech następnych zgrupowanych jest wiele fraz **case** pustych, z wyjątkiem ostatniej, która zawiera instrukcję **return**. Dzięki temu zawsze, gdy argumentem funkcji jest znak odpowiadający którejś z cyfr, sterowanie dojdzie do linii ② zwracając właściwą wartość (bo liczbowo zmienna c ma wartość kodu ASCII odpowiedniej cyfry; odejmując kod ASCII znaku '0' otrzymamy wartość liczbową znaku c). Podobnie dla dowolnej małej litery odpowiadającej którejś z cyfr szesnastkowych sterowanie dojdzie do linii ③, a dla dużej litery — do linii ④. Jeśli znak nie odpowiada żadnej cyfrze szesnastkowej, wejdziemy do frazy **default** i zwrócona zostanie w linii ⑤ wartość −1. Wynik tego programu:

```
A = 10
f = 15
9 = 9
b = 11
Z = -1
```

Podobnie jak dla instrukcji **if**, nowy standard dopuszcza użycie *init-statement* w instrukcji **switch**:

```
srand(time(nullptr));
switch (auto r = std::rand()%30; r/10) {
    case 0:  cout << r << " -> first ten\n"; break;
    case 1:  cout << r << " -> second ten\n"; break;
    default: cout << r << " -> third ten\n";
}
```

8.9 Instrukcje iteracyjne (pętle)

Instrukcje iteracyjne, zwane pętlami (ang. *iterative statement, loop*) mają charakter cykliczny: wykonanie instrukcji jest powtarzane dopóki spełniony jest pewien warunek logiczny. Programista powinien zapewnić, że wykonanie pętli skończy się, a więc że warunek logiczny, od którego zależy dalsze powtarzanie instrukcji, będzie kiedyś niespełniony.

Inną możliwością przerwania pętli jest zastosowanie instrukcji powrotu (**return**), zaniechania (**break**) lub skoku (**goto**), co omówimy dalej.

Instrukcje iteracyjne występują w trzech podstawowych odmianach, przy czym każda forma może być przekształcona do każdej z pozostałych; wybór jednej z tych form zależy tylko od wygody i upodobań programisty.

Przebieg każdej pętli może być modyfikowany lub przerwany instrukcjami **break** i **continue**.

8.9.1 Pętla *while*

Pętla **while** ma postać

```
while ( b ) instr
```

gdzie wyrażenie b ma wartość logiczną a instr jest instrukcją. Składnia wymaga jednej instrukcji instr, więc jeśli ma ich być więcej, to należy wprowadzić instrukcję grupującą (rozdz. 8.5, str. 104), tak jak to było w przypadku instrukcji warunkowych. Wykonanie instrukcji polega na cyklicznym powtarzaniu następujących czynności:

- wyznaczenie wartości b,
- jeśli wartość b jest **false**: zakończenie wykonywania instrukcji,
- jeśli wartość b jest **true**: wykonanie instrukcji instr,
- powrót do początku pętli.

Na przykład poniższy fragment programu wyznaczy pierwszą liczbę naturalną postaci 2^n, która jest większa od zadanej liczby lim:

```
int liczba = 1;
while ( liczba <= lim ) liczba *= 2;
```

Natomiast fragment poniższy spowoduje, że program będzie wczytywał dane od użytkownika aż do chwili, gdy poda on liczbę w zakresie $[5, 100]$:

```
int wiek = 0;
while ( wiek < 5 || wiek > 100) cin >> wiek;
```

8.9.2 Pętla *do*

Pętla **do-while** ma postać

```
do instr while ( b )
```

gdzie wyrażenie b ma wartość logiczną, a instr jest pewną instrukcją. Składnia wymaga jednej instrukcji instr, więc jeśli ma ich być więcej, to należy wprowadzić instrukcję grupującą (tak jak to było w przypadku instrukcji warunkowych i pętli **while**). Wykonanie instrukcji polega na cyklicznym powtarzaniu następujących czynności:

- wykonanie instrukcji instr,

- wyznaczenie wartości b,
- jeśli wartość b jest **false**: zakończenie wykonywania instrukcji,
- powrót do początku pętli.

Jak widać, instrukcja **do...while** różni się od instrukcji **while** tym, że warunek kontynuowania pętli sprawdza się *po* wykonaniu instrukcji instr, a nie przed. Wynika z tego w szczególności, że instrukcja instr będzie zawsze wykonana chociaż raz, nawet jeśli wartość b wynosi **false** (w pętli **while**, jeśli b ma wartość **false**, instrukcja instr nie byłaby wykonana ani razu).

Poniższy program symuluje serię rzutów dwiema kostkami do gry i kończy się, gdy wyrzucone zostaną dwie szóstki. Użyto tu standardowego generatora liczb losowych (**rand** inicjowana jednokrotnym wywołaniem funkcji **srand** z nagłówka *<cstdlib>*) do losowania wyrzucanej liczby oczek na każdej z obu kostek. Ponieważ liczba prób (rzutów) nie jest z góry znana, a zawsze trzeba i tak wykonać co najmniej jeden rzut, forma **do...while** stanowi tu najbardziej naturalny wybór dla realizacji pętli:

P49: *kostki.cpp* Pętla *do...while*

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main() {
    int x, y, roll = 0;

    srand(unsigned(time(0)));

    do {
        x =  (int)(rand()/(RAND_MAX + 1.)*6)+1;
        y =  (int)(rand()/(RAND_MAX + 1.)*6)+1;
        cout << "Rzut nr " << ++roll   << ": ("
             <<  x << ", " << y << ")" << endl;
    } while (x + y != 12);
}
```

Przykładowy wynik tego programu:

```
Rzut nr 1: (2, 1)
Rzut nr 2: (5, 4)
Rzut nr 3: (1, 5)
Rzut nr 4: (2, 5)
Rzut nr 5: (4, 5)
Rzut nr 6: (4, 5)
Rzut nr 7: (4, 4)
Rzut nr 8: (5, 6)
```

```
Rzut nr 9: (1, 6)
Rzut nr 10: (4, 3)
Rzut nr 11: (1, 2)
Rzut nr 12: (2, 2)
Rzut nr 13: (5, 2)
Rzut nr 14: (3, 2)
Rzut nr 15: (2, 4)
Rzut nr 16: (1, 3)
Rzut nr 17: (5, 6)
Rzut nr 18: (6, 6)
```

Oczywiście, ponieważ używamy tu generatora liczb losowych, wyniki będą się różnić przy każdym uruchomieniu programu.

8.9.3 Pętla *for*

Pętla **for** ma postać

```
for ( init ; b ; incr ) instr
```

gdzie instr jest instrukcją (być może złożoną, być może pustą). Wyrażenie b ma wartość logiczną; jeśli jest pominięte, to przyjmuje się wartość **true**. Wyrażenie init może być

- *jedną* instrukcją deklaracyjną;
- listą oddzielonych przecinkami instrukcji wyrażeniowych;
- puste.

Część „inkrementująca", oznaczona tu przez incr, jest listą oddzielonych przecinkami instrukcji wyrażeniowych, albo jest pominięta.

Nawet jeśli poszczególne elementy (init, b lub incr) są pominięte, nie wolno pominąć średników ani nawiasów okrągłych.

Przebieg wykonania instrukcji for jest następujący:

1. Jeśli init nie jest pominięte, to wykonywane są instrukcje zawarte w init. Jeśli jest to lista instrukcji wyrażeniowych, to wykonywane są w kolejności od lewej do prawej; wartości tych instrukcji są ignorowane. Jeśli jest to jedna instrukcja deklaracyjna, to zasięgiem zadeklarowanych tam zmiennych (których może być wiele, ale wszystkie tego samego typu) jest cała pętla, czyli dalsza część init, b, incr i instr. Instrukcje zawarte w init, jeśli nie są pominięte, są wykonywane zawsze (nawet gdy instr nie będzie wykonana ani razu) i zawsze tylko raz — podczas „wejścia" do pętli.
2. Obliczana jest wartość wyrażenia b lub przyjmowana jest wartość **true**, jeśli wyrażenie to jest pominięte. Jeśli wartością b jest **false**, wykonanie pętli kończy się. Jeśli tą wartością jest **true** — przechodzimy do kroku 3.

3. Wykonywane jest ciało pętli, czyli instr.
4. Jeśli część incr nie jest pusta, wykonywane są zawarte tam instrukcje, a wartość tego wyrażenia jest ignorowana.
5. Powrót do kroku 2.

Należy pamiętać, że nie jest możliwe zadeklarowanie w części init zmiennych różnych typów, gdyż wymagałoby to więcej niż jednej instrukcji deklaracyjnej:

```
for (double x = 0, int k = size-1; x < k; x++, k--) {
    // ... ZLE !!!
}
```

Można natomiast zadeklarować więcej niż jedną zmienną tego samego typu; na przykład konstrukcja

```
for (int i = 0, k = size-1; i < k; i++, k--) {
    // OK
}
```

jest prawidłowa.

Pętla **for** jest najczęściej stosowaną formą instrukcji iteracyjnej. Jest szczególnie wygodna, jeśli z góry wiadomo, ile będzie powtórzeń instrukcji stanowiącej ciało pętli (instrukcja instr). Zwykle w części init nadajemy odpowiednie wartości zmiennym używanym w ciele pętli, a w części incr dokonujemy ich zmiany po każdym przebiegu pętli.

Jako przykład, z wykorzystaniem instrukcji deklaracyjnej w części inicjującej init i sekwencji instrukcji wyrażeniowych w części inkrementującej incr, rozważmy poniższy program, w którym funkcja **reverse** odwraca kolejność elementów w tablicy liczb całkowitych:

P50: *revers.cpp* Pętla *for*

```
1  #include <iostream>
2  using namespace std;
3
4  void reverse(int *tab, int size) {
5      if ( size < 2 ) return;
6
7      for (int i = 0, k = size-1, aux; i < k; i++, k--) { ①
8          aux    = tab[i];
9          tab[i] = tab[k];
10         tab[k] = aux;
11     }
12 }
13
14 void printTab(int *tab, int size) {
```

```
15      cout << "[ ";
16      for (int i = 0; i < size; i++)
17          cout << tab[i] << " ";
18      cout << "]" << endl;
19  }
20
21  int main() {
22      int tab[] = { 1, 3, 5, 7, 2, 4, -9, 12 };
23      int size = sizeof(tab)/sizeof(tab[0]);
24
25      printTab(tab,size);
26      reverse (tab,size);
27      printTab(tab,size);
28  }
```

Pętla w linii ① przebiega jednocześnie po elementach tablicy od początku (zmienną indeksującą jest tu i) i od końca (elementy indeksowane zmienną k); pętla kończy się, gdy te dwa indeksy spotkają się. W ten sposób zamieniane są miejscami element pierwszy z ostatnim, drugi z przedostatnim itd. W części inicjującej pętli zadeklarowaliśmy, prócz zmiennych indeksujących, również zmienną pomocniczą aux wykorzystywaną przy zamianie wartości każdej pary elementów tablicy. Wynik tego programu:

```
[ 1 3 5 7 2 4 -9 12 ]
[ 12 -9 4 2 7 5 3 1 ]
```

8.9.4 Pętla *foreach*

W standardzie C++11 wprowadzono jeszcze jedną formę pętli, tzw. pętlę „foreach". Może ona być stosowana do przebiegania (iterowania po) kolekcji z biblioteki standardowej (na przykład **std::array**), ale działa również dla tablic statycznych — pod warunkiem, że w danym zakresie tablica jest widoczna jako posiadająca typ tablicowy (a nie wskaźnikowy, jak to ma miejsce po przesłaniu tablicy do funkcji). Składnię przedstawimy na przykładzie:

P51: *foreach.cpp* Pętla „foreach"

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int arr[] = {1,2,3,4,9,8,7,6};
6
7      for (int e  : arr) cout << e << " ";    ①
8      cout << endl;
9      for (int& e : arr) e -= 1;              ②
10     for (auto e : arr) cout << e << " ";    ③
```

```
11      cout << endl;
12  }
```

W linii ① zmienna e będzie w każdym przebiegu pętli zainicjowana niemodyfikowalną *kopią* kolejnych elementów tablicy arr. W linii ② natomiast, w każdym przebiegu pętli e będzie *referencją* do kolejnego elementu tablicy, a zatem elementy te można modyfikować (tu odejmujemy od wszystkich elementów jedynkę). Jak widzimy w linii ③, nie musimy specyfikować jawnie typu elementów — kompilator może ten typ wydedukować z typu tablicy (bardziej ogólnie z typu kolekcji). To samo dotyczy referencji: w linii ② mogliśmy napisać `auto&` zamiast `int&`. Wydruk programu to

```
1 2 3 4 9 8 7 6
0 1 2 3 8 7 6 5
```

Możliwe jest też iterowanie po kolekcji stworzonej *ad hoc* — w postaci ujętej w nawiasy klamrowe sekwencji wartości tego samego typu, jak w programie

P52: ***adhocforeach.cpp*** Pętla po kolekcji *ad hoc*

```
#include <iostream>

long long factorial(int n) {
    return n < 2 ? 1 : n*factorial(n-1);
}

int main() {
    for (auto n : {12, 14, 16, 18, 20}) {
        if (auto f = factorial(n); f > 1e17)
            std::cout << f << " is really big\n";
        else
            std::cout << f << " is not so big\n";
    }
}
```

który drukuje

```
479001600 is not so big
87178291200 is not so big
20922789888000 is not so big
6402373705728000 is not so big
2432902008176640000 is really big
```

8.10 Instrukcje zaniechania i kontunuacji

Wewnątrz pętli (**for**, **do...while** i **while**) można użyć tzw. instrukcji kontynuacji **continue**. Oznacza ona zaniechanie wykonywania bieżącego obrotu najbardziej wewnętrznej pętli obejmującej tę instrukcję i przejście do jej następnego obrotu. W przypadku

pętli **for** następną wykonywaną instrukcją będą zatem instrukcje wymienione w części inkrementującej, jeśli nie były pominięte. Innymi słowy, wszystkie instrukcje występujące w pętli po instrukcji **continue** traktowane są jak jedna instrukcja pusta.

W następującym programie funkcja **sumaDod** przegląda tablicę liczb i oblicza sumę (która wynosi 17) wszystkich elementów dodatnich — elementy ujemne są pomijane dzięki instrukcji **continue** z linii ①:

P53: *sumadod.cpp* Instrukcja kontynuowania

```
#include <iostream>
using namespace std;

int sumaDod(int *tab, int size) {
    int suma = 0;
    for (int i = 0; i < size; i++) {
        if ( tab[i] <= 0 ) continue;    ①
        suma += tab[i];
    }
    return suma;
}

int main() {
    int tab[] = { 1, -3, 5, -7, 2, 0, 9 };
    int suma = sumaDod(tab,sizeof(tab)/sizeof(tab[0]));
    cout << "Suma: " << suma << endl;
}
```

Instrukcja zaniechania **break** przerywa wykonywanie najbardziej wewnętrznej pętli obejmującej tę instrukcję. Modyfikując poprzedni program, można następująco obliczyć sumę wszystkich elementów tablicy aż do napotkania elementu niedodatniego — po napotkaniu takiego elementu pętla jest przerywana i wszystkie następne elementy, i dodatnie i ujemne, są ignorowane (wynikiem jest 16):

P54: *sumaazdo.cpp* Instrukcja zaniechania

```
#include <iostream>
using namespace std;

int sumaazdo(int *tab, int size) {
    int suma = 0;
    for (int i = 0; i < size; i++) {
        if ( tab[i] <= 0 ) break;       ①
        suma += tab[i];
    }
    return suma;
}
```

```
12
13 int main() {
14     int tab[] = { 1, 3, 5, 7, 0, 4, 9 };
15     int suma = sumaazdo(tab, sizeof(tab)/sizeof(tab[0]));
16     cout << "Suma: " << suma << endl;
17 }
```

W linii ① pętla jest przerywana, jeśli napotkany został element zerowy lub ujemny.

Instrukcja **break** powoduje też zakończenie wykonywania instrukcji **switch**, o czym wspomnieliśmy przy okazji omawiania tej instrukcji (rozdz. sect. 8.8, str. 108). Instrukcja **continue** oczywiście nie miałaby dla instrukcji wyboru sensu, gdyż nie jest ona w ogóle instrukcją iteracyjną wykonywaną cyklicznie (pętlą).

Uwaga dla znających Javę: w tym języku obie instrukcje — **break** i **continue** — mają formę „z etykietą". Takiej formy w C/C++ *nie ma*.

8.11 Instrukcje skoku

Wykonanie instrukcji skoku przenosi sterowanie w inne miejsce programu. Ma ona postać

```
goto etykieta;
```

gdzie etykieta jest etykietą pewnej instrukcji wewnątrz tej samej funkcji. Sterowanie zostanie przeniesione właśnie do tej instrukcji.

W zasadzie język dopuszcza najdziwniejsze skoki, ale należy unikać skoków do wnętrza instrukcji złożonych (choć formalnie takie skoki są legalne, jeśli tylko nie omijają deklaracji z inicjatorami). Najczęściej używa się instrukcji **goto** aby wyjść z wnętrza zagnieżdżonych pętli, jak w następującym programie:

P55: *oceny.cpp* Instrukcja skoku

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      const int st_size = 7;
6
7      int oceny[][st_size] = { { 5, 4, 3, 3, 3, 4, 4 },
8                               { 5, 3, 3, 3, 4, 2, 3 },
9                               { 4, 4, 4, 4, 3, 3, 5 } };
10
11     int gr_size = sizeof(oceny)/sizeof(oceny[0]);
12
13     bool jestDwoja = false;
14
```

```
15      for (int grupa = 0; grupa < gr_size; grupa++)   ①
16          for (int student = 0; student < st_size; student++)
17              if ( oceny[grupa][student] == 2 ) {
18                  jestDwoja = true;
19                  goto KONIEC;
20              }
21  KONIEC:                                              ②
22      if (jestDwoja) cout << "Byla dwojka"    << endl;
23      else           cout << "Nie ma dwojki" << endl;
24  }
```

W zagnieżdżonej pętli rozpoczynającej się w linii ① szukamy oceny niedostatecznej w dwuwymiarowej tablicy ocen indeksowanej numerem grupy i numerem studenta. Ponieważ interesuje nas tylko, czy jest choć jedna ocena niedostateczna, w momencie, gdy taką znajdziemy, chcemy wyjść z obu pętli. Zauważmy, że instrukcja **break** przerwałaby tylko pętlę wewnętrzną; pętla po grupach byłaby kontynuowana. Dlatego użyliśmy instrukcji **goto**, która „wyrzuca" nas od razu z obu pętli aż do linii ②.

8.12 Instrukcje powrotu

Instrukcja powrotu **return** powoduje zakończenie wykonywania funkcji i powrót do miejsca programu, skąd dana funkcja została wywołana. Jeśli funkcja jest bezrezultatowa, to instrukcja **return** jest domniemywana jako ostatnia instrukcja przed nawiasem zamykającym definicję ciała funkcji.

Jeśli funkcja jest rezultatowa, to instrukcja **return** musi mieć formę

```
return val;
```

gdzie val jest wyrażeniem o wartości typu przypisywalnego do zadeklarowanego jako typ rezultatu funkcji. Wartością tą jest, po ewentualnej konwersji, inicjowany rezultat funkcji wywoływanej dostępny w funkcji wołającej. Zauważmy, że funkcja **main** jest wyjątkowa: choć jest funkcją rezultatową (zwraca **int**), to można instrukcję **return** opuścić (kompilator dostawi wtedy na końcu '`return 0`').

8.13 Instrukcje obsługi wyjątków

Są to instrukcje związane z definiowaniem bloków kodu, w których deklarujemy możliwość powstania wyjątków (**try**), definiowaniem bloków obsługujących powstałe wyjątki (**catch**) i zgłaszaniem wyjątków (**throw**). Dyskusję tego tematu odłożymy do rozdz. 22, str. 491.

— 9 —

Operatory

W tym rozdziale poznamy operatory występujące w języku C++, choć dokładniejsze omówienie niektórych z nich odłożymy do następnych rozdziałów. **Operatorami** są pojawiające się w tekście programu niealfanumeryczne leksemy (znak dodawania, znak gwiazdki, znak procentu, itd.) które są interpretowane jako żądanie wywołania odpowiednich funkcji operujących na danych określonych przez wyrażenia sąsiadujące z danym operatorem — są to **argumenty** (lub **operandy**) tego operatora.

Operatory generalnie dzielą się na jedno- i dwuargumentowe. Jak w większości (choć nie we wszystkich) języków programowania, w C/C++ zapis operatorów dwuargumentowych jest **infiksowy** (wrostkowy), czyli operator stawiany jest *pomiędzy* swoimi argumentami. Z kolei zapis operatorów jednoargumentowych, z dwoma wyjątkami, jest **prefiksowy** (przedrostkowy), czyli operator stawiamy *przed* argumentem. Istnieje w C/C++ również jeden operator trzyargumentowy: operator selekcji (operator warunkowy).

Podrozdziały:

9.1 Priorytety i wiązanie

Zapis infiksowy, wygodny i naturalny dla ludzi, prowadzi jednak do sytuacji, gdy z samego zapisu nie wynika kolejność wykonania działań w złożonych wyrażeniach. Na przykład w wyrażeniu

```
a + b / c
```

zmienna b może być traktowana jako prawy argument operatora dodawania albo lewy argument operatora dzielenia. W pierwszym przypadku obliczymy $(a+b)/c$, a w drugim $a \cdot (b/c)$ otrzymując różny wynik.

Aby uniknąć tego rodzaju wieloznaczności, wprowadzono pojęcie priorytetu operatorów. Według priorytetu operatory dzielą się na szereg grup, w ramach których priorytety są jednakowe. W sytuacjach, jak opisana powyżej, gdy to samo wyrażenie może być potraktowane jako argument dwóch operatorów, najpierw zostanie wykonana operacja opisywana przez ten z tych dwóch operatorów który ma wyższy priorytet. Jeśli natomiast oba mają ten sam priorytet, kolejność wykonywania operacji będzie określona ich **wiązaniem**: od lewej do prawej dla operatorów o wiązaniu lewym i od prawej do lewej dla operatorów o wiązaniu prawym. Aby ta reguła nie prowadziła do sprzeczności, operatory o takim samym priorytecie muszą mieć taki sam kierunek wiązania (co w istocie zachodzi). A zatem w wyrażeniu

```
a + b / c
```

najpierw zostanie wykonane dzielenie, gdyż ma wyższy priorytet od dodawania. Ale w wyrażeniu

```
a + b - c
```

najpierw zostanie wykonane dodawanie, gdyż ma ten sam priorytet co odejmowanie, a oba operatory mają wiązanie lewe.

Dla operatora przypisania wiązanie jest prawe. Tak więc w wyrażeniu `a=b=c` przypisanie `b=c` wykonane zostanie najpierw, a jego wynik (wartość b po tej operacji) przypisany będzie do zmiennej a. Ale np. operator wyboru składowej („kropka") ma wiązanie lewe, zatem `obiekt.sklad1.sklad2` oznacza: *najpierw* wybierz składową sklad1 obiektu obiekt, potem z wynikowego obiektu składową sklad2.

Wszystkie prefiksowe (przedrostkowe) operatory jednoargumentowe mają wiązanie prawe: najpierw działa operator z prawej strony, czyli bliższy argumentu. Tak więc `*++p` to to samo co `*(++p)`, natomiast `++*p` to `++(*p)`, bo choć przedrostkowy operator inkrementacji '++' i operator wyłuskania wartości (dereferencji) '*' mają ten sam priorytet, oba mają wiązanie prawe.

9.2 Przegląd operatorów

W tabeli poniżej przedstawiono operatory języka C++. W prawej kolumnie użyte są oznaczenia:

klasa: nazwa klasy	*ob*: obiekt klasy
sklad: składowa klasy lub przestrzeni nazw	*wsk*: wskaźnik
wyr: wyrażenie	*lwar*: l-wartość
pnaz: nazwa przestrzeni nazw	*typ*: nazwa typu
naz: nazwa	

Operatory podzielone są na 18 grup — każda grupa odpowiada operatorom o tym samym priorytecie. Grupy wymienione są w kolejności od grupy operatorów o priorytecie najwyższym, w dół według malejącego priorytetu. W pierwszej kolumnie zaznaczona jest kolejność wiązania: 'R' – od prawej do lewej, 'L' – od lewej do prawej.

Tablica 9.1: Operatory w języku C++

L/R	**Funkcja**	**Użycie**
	Priorytet 16	
L	operator zasięgu	klasa::sklad,pnaz::sklad,::naz
	Priorytet 15	
L	dostęp do składowej	ob.sklad,wsk->sklad
L	indeksowanie	wyr[wyr]
L	wywołanie funkcji	wyr(lista_wyr)
L	konstruowanie wartości	typ(lista_wyr),typ{lista_wyr}
L	postdekrementacja,postinkr.	lwar−−,lwar++
	Priorytet 14	
R	rozmiar obiektu	**sizeof** wyr
R	rozmiar typu	**sizeof**(typ)
R	rozmiar pakietu	**sizeof...**(naz)
R	predekrementacja,preinkr.	−−lwart,++lwart
R	negacja bitowa,logiczn	∼wyr,!wyr
R	minus, plus jednoargumentowy	−wyr,+wyr
R	wyłuskanie adresu	&lwart
R	dereferencja	*wsk
R	przydział pamięci na obiekt	**new** typ
R	przydział pamięci na tablicę	**new** typ[wyr]
R	zwolnienie obiektu	**delete** wyr
R	zwolnienie tablicy	**delete** [ ] wyr
R	rzutowanie w stylu C	(typ)wyr
	Priorytet 13	
L	wskaźnik do składowej	wsk->*sklad_wsk,ob.*sklad_wsk
	Priorytet 12	
L	mnożenie,dzielenie	wyr*wyr,wyr/wyr
L	reszta z dzielenia	wyr % wyr
	Priorytet 11	
L	dodawanie,odejmmowanie	wyr+wyr,wyr−wyr
	Priorytet 10	
L	przesunięcie w lewo, prawo	wyr<<wyr,wyr>>wyr
	Priorytet 9	
L	mniejsze od	wyr < wyr
L	mniejsze lub równe od	wyr <= wyr

...⇒

Operatory C++ — *c.d.*

L/R	Funkcja	Użycie
L	większe od	wyr > wyr
L	większe lub równe od	wyr >= wyr
	Priorytet 8	
L	równe	wyr == wyr
L	nierówne	wyr != wyr
	Priorytet 7	
L	koniunkcja bitowa	wyr&wyr
	Priorytet 6	
L	bitowa różnica symetryczna	wyr^wyr
	Priorytet 5	
L	alternatywa bitowa	wyr|wyr
	Priorytet 4	
L	koniunkcja logiczna	wyr && wyr
	Priorytet 3	
L	alternatywa (suma) logiczna	wyr || wyr
	Priorytet 2	
R	operator warunkowy	wyr ? wyr : wyr
R	przypisanie	lwar = wyr
R	dodawanie z przypisaniem	lwar += wyr
R	odejmowanie z przypisaniem	lwar −= wyr
R	mnożenie z przypisaniem	lwar *= wyr
R	dzielenie z przypisaniem	lwar /= wyr
R	reszta z przypisaniem	lwar %= wyr
R	przesunięcie w lewo z przypisaniem	lwar <<= wyr
R	przesunięcie w prawo z przypisaniem	lwar >>= wyr
R	iloczyn bitowy z przypisaniem	lwar&= wyr
R	alternatywa bitowa z przypisaniem	lwar |= wyr
R	różnica bitowa z przypisaniem	lwar ^= wyr
R	zgłoszenie wyjątku	**throw** wyr
	Priorytet 1	
L	operator przecinkowy	wyr,wyr

Operatory **const_cast**, **static_cast**, **dynamic_cast**, **reinterpret_cast**, **typeid**, **noexcept** i **alignof** nie zostały wymienione, ponieważ ich użycie zawsze jest jednoznaczne.

Dyskusję niektórych z tych operatorów, szczególnie tych związanych z klasami, konwersjami, przestrzeniami nazw i obsługą wyjątków, odłożymy do następnych rozdziałów.

9.2.1 Operatory zasięgu

Operatory te (priorytet 16 w tabeli), zapisywane są za pomocą symbolu „czterokropka” (’::’). Operator zasięgu globalnego już znamy (patrz rozdz. 7.2). Przypomnijmy, że ::x jest nazwą globalnej zmiennej x zadeklarowanej poza wszystkimi funkcjami i klasami. Użycie „czterokropka” jest konieczne tylko wtedy, gdy nazwa (w naszym przypadku x) jest w danym bloku (funkcji) nazwą innej zmiennej, lokalnej, która wobec tego przesłoniła zmienną globalną o tej samej nazwie.

Operatory zasięgu klasy (pozycja pierwsza w tabeli) omówimy w rozdz. 14 (str. 263), a przestrzenie nazw w rozdz. 23.2 na stronie 510.

9.2.2 Grupa operatorów o priorytecie 15

Pierwsze dwa (operator „kropka” i „strzałka”, dotyczą struktur i klas, poznamy je w rozdz. 13.1 (str. 235).

Operator ’[]’ (indeksowania) tablicy już znamy z rozdz. 5.3.

Operator wywołania funkcji oznaczamy nawiasami okrągłymi ’()’:

```
func(k, m, 5)
```

Zatem podanie nazwy funkcji z nawiasami okrągłymi powoduje wywołanie funkcji (jeśli pojawia się w instrukcji wykonywalnej, a nie w definicji lub deklaracji). Ale nie zawsze nazwa funkcji występuje z nawiasami. Czasem chcemy odnieść się do funkcji jako takiej, jako obiektu, a nie powodować jej wywołanie. W takich sytuacjach używamy nazwy funkcji bez nawiasów — ma ona wtedy interpretację *wskaźnika do funkcji*. Tego rodzaju wskaźniki omówimy w rozdz. 11.12 na stronie 184.

Operator konstrukcji wartości (`Typ(list_expr)`, `Typ\{list_expr\}`) jest nam nieznany. Ponieważ dotyczy klas, omówimy go w rozdz. 14.8 na stronie 279. Tu tylko wspomnijmy, że w C++ można konstruować obiekty typów prostych (**int**, **double**, ...) tak jakby były one obiektami jakiejś klasy. Zatem **int**`(3)` kreuje zmienną typu **int** i inicjuje ją wartością 3 — składnia jest wobec tego taka, jak gdybyśmy tworzyli obiekt klasy **int** i wysyłali wartość 3 do konstruktora (tak zwanego konstruktora kopiującego).

Przyrostkowe operatory zmniejszenia i zwiększenia (postdekrementacji i postinkrementacji) zmniejszają (zwiększają) wartość swojego argumentu (który wyjątkowo stoi po ich *lewej* stronie) o jeden. Czynią to jednak *po* obliczeniu wartości wyrażenia, w skład którego wchodzą. Tak więc po

```
int a = 1;
int b = a++;
```

wartość a wynosi 2, ale wartość b wynosi 1, bo w trakcie opracowywania drugiej instrukcji a miało wciąż wartość 1; zwiększenie a nastąpi dopiero po zakończeniu wykonywania instrukcji przypisania.

Argumentem postinkrementacji i postdekrementacji musi zawsze być l-wartość; np.

```
(x+y)++
```

nie ma sensu i jest błędne, gdyż wyrażenie `(x+y)` nie jest l-wartością (choć jest p-wartością). Tak więc argument operatorów postdekrementacji i postinkrementacji musi być l-wartością: ale co z rezultatem działania tego operatora na l-wartość? W Javie wartości uzyskane za pomocą tych operatorów nigdy same *nie* są l-wartościami. W C/C++ jest trochę inaczej: dla przyrostkowych operatorów zmniejszenia i zwiększenia wynik *nie* jest l-wartością, ale dla operatorów przedrostkowych wynik *jest* l-wartością. Dlatego

```
int k = 5;
int m = (++k)--;
```

jest prawidłowe. Wyrażenie w nawiasach jest l-wartością, bo użyty został operator preinkrementacji; można było zatem zastosować następnie operator postdekrementacji. Wynikiem działania tego z kolei operatora nie jest już l-wartość, ale ponieważ użyliśmy jej tylko po prawej stronie przypisania, więc wszystko jest w porządku. Wartość `k` będzie oczywiście wynosić po tym przypisaniu 5, a wartością zmiennej `m` będzie 6.

Gdybyśmy nie użyli nawiasów

```
int k = 5;
int m = ++k--; // ZLE !!!
```

kod byłby błędny: ponieważ priorytet postdekrementacji jest wyższy niż preinkrementacji, więc najpierw obliczone byłoby wyrażenie `k--`. Wynik nie byłby l-wartością — patrz rozdz. 7.5 na stronie 96 — więc podziałanie nań operatorem '`++`' spowodowałoby błąd.

Warto pamiętać, że wyrażenie `++w` *nie* jest całkowicie równoważne instrukcji `w=w+1`. Ta druga forma jest normalną instrukcją przypisania, a zatem najpierw zostanie obliczona wartość wyrażenia po prawej stronie, a potem lokalizacja l-wartości po lewej stronie. Zatem jeśli `w` jest wyrażeniem złożonym (np. zawiera wywołania funkcji), to wyrażenie to będzie obliczane dwukrotnie. Natomiast podczas opracowywania wyrażenia `++w`, samo `w` będzie obliczane jednokrotnie. Rzadko ma to jakieś znaczenie, ale czasem zrozumienie tego może nas ustrzec przed trudno wykrywalnymi błędami.

Operator identyfikacji typu **typeid** (niewymieniony w tableki) pozwala na uzyskanie identyfikatora typu podczas kompilacji, a więc statycznie, jak również identyfikatora typu obiektu (ogólnie p-wartości) w czasie wykonania programu, a więc dynamicznie (RTTI; *run-time type identification*). Ten temat omówimy bardziej szczegółowo w rozdz. 25 na str. 553, ale jeden przykład zastosowania tego operatora podany jest poniżej w programie ***sizes.cpp*** (str. 127).

Operatory **konwersji** (**rzutowania**), równiez niewymienione w tabeli: **const_cast**, **static_cast**, **dynamic_cast** i **reinterpret_cast**, pozwalają na konwersję wartości jednego typu na wartość innego typu. Ponieważ stosowanie konwersji często, choć nie zawsze, świadczy o złej konstrukcji programu i stwarza okazję do użycia błędnych lub zależnych od implementacji konstrukcji programistycznych, nadano tym operatorom celowo tak długą i niewygodną do pisania formę. Konwersje rozpatrzymy w rozdz. 20 na stronie 439.

9.2.3 Grupa operatorów o priorytecie 14

Pierwsze trzy z opertatorów tej grupy dotyczą pobierania rozmiaru za pomocą operatora **sizeof**. Reultat jest typu **size_t** (który jest tożsamy z pewnym typem całkowitym bez znaku, np. **unsigned long**). Operator ten jest jednoargumentowy. Argumentem może być nazwa typu (w nawiasie okrągłym) lub wyrażenie (nawias jest wtedy niekonieczny) albo tak zwany pakiet (w ostatnim przypadku operator zapisujemy z trzema kropkami: **sizeof...**). Rozpatrzmy przykład ilustrujący również użycie operatora **typeid**:

P56: *sizes.cpp* Operator *sizeof*

```
#include <iostream>
#include <typeinfo>
using namespace std;

typedef int TABINT15[15];                                          ①

void siz(TABINT15 t1, TABINT15& t2) {                              ②
   cout << "G. t1 w siz: " << sizeof t1 << endl;
   cout << "H. t2 w siz: " << sizeof t2 << endl;
}

int main() {
   TABINT15 tab1;                                                  ③
   int      tab2[15];                                              ④
   int     *tab3 = tab2;                                           ⑤

   if (typeid(tab1) == typeid(tab2))
      cout << "A. Typy tab1 i tab2 takie same"     << endl;
   else
      cout << "A. Typy tab1 i tab2 nie takie same" << endl;

   if (typeid(tab2) == typeid(tab3))
      cout << "B. Typy tab2 i tab3 takie same"     << endl;
   else
      cout << "B. Typy tab2 i tab3 nie takie same" << endl;

   cout << "C. TABINT15: " << sizeof(TABINT15) << endl;             ⑥
   cout << "D. tab1    : " << sizeof tab1      << endl;             ⑦
   cout << "E. tab2    : " << sizeof(tab2)     << endl;             ⑧
   cout << "F. tab3    : " << sizeof tab3      << endl;             ⑨
   siz(tab2, tab2);
}
```

Wynik tego programu

```
A. Typy tab1 i tab2 takie same
```

```
B. Typy tab2 i tab3 nie takie same
C. TABINT15: 60
D. tab1    : 60
E. tab2    : 60
F. tab3    : 8
G. t1 w siz: 8
H. t2 w siz: 60
```

ilustruje warte zrozumienia własności C/C++.

Na początku dołączamy plik nagłówkowy ***typeinfo***, aby mieć dostęp do narzędzi związanych z identyfikacją typów (patrz rozdz. 25 na str. 553).

W linii ① wprowadzamy, za pomocą znanej już nam instrukcji **typedef**, inną nazwę typu „piętnastoelementowa tablica liczb całkowitych" (patrz rozdz. 6.2 na stronie 76).

Jak widać z linii ⑥ i z pierwszej linii wydruku, operator **sizeof** prawidłowo rozpoznał rozmiar typu **TABINT15**. Zauważmy też, że nie można tu pominąć nawiasów, bo **TABINT15** jest nazwą typu.

W liniach ③ i ④ definiujemy tablice tab1 i tab2 na dwa sposoby — za pomocą wprowadzonej nazwy **TABINT15** i bezpośrednio. Porównując typy tych zmiennych widzimy, że rzeczywiście są one takie same (linia 'A' wydruku).

W linii ⑤ definiujemy zmienną tab3 typu **int*** i przypisujemy do niej wartość zmiennej tab2. Przypisanie jest prawidłowe, ale nie zapominajmy, że zachodzi przy tym konwersja standardowa: typy tab2 i tab3 nie są takie same, co widzimy z linii 'B' wydruku.

W liniach ⑥-⑨ drukujemy rozmiary typu **TABINT15** i zmiennych tab1, tab2 i tab3. Wszystkie rozmiary, za wyjątkiem tab3, wynoszą 60, co odpowiada tablicy piętnastu czterobajtowych liczb. Natomiast rozmiar tab3 jest 8, gdyż jest to zmienna typu wskaźnikowego, a nie tablicowego.

W ostatniej linii posyłamy *tę samą* tablicę tab2 poprzez dwa argumenty do funkcji **siz**. Pierwszy parametr funkcji jest typu **TABINT15**, więc wydawałoby się, że funkcja „wie", że argument będzie tablicą. Drukując jednak (linia 'G' wydruku) wewnątrz funkcji rozmiar zmiennej t1, widzimy, że jest to wskaźnik o rozmiarze 8 — jest tak, gdyż przy wysyłaniu tab2 do funkcji przez wartość i tak zmienna została zrzutowana do typu wskaźnikowego (na stosie został położony adres pierwszego elementu tablicy i *nic więcej*).

Drugi parametr funkcji **siz** jest zadeklarowany jako referencja. Teraz żadnej konwersji nie ma, bo nie jest w ogóle tworzona żadna zmienna lokalna, której należałoby przypisać wartość argumentu. Wewnątrz funkcji t2 jest teraz inną nazwą dokładnie tej samej zmiennej, która w funkcji **main** nazywa się tab2. Zatem i informacja o typie jest ta sama i '`sizeof t2`' drukuje 60 (linia 'H' wydruku).

Przedrostkowe operatory zmniejszenia i zwiększenia (predekrementacji i preinkrementacji) działają podobnie do przyrostkowych operatorów zmniejszenia i zwiększenia. Są jednak ważne różnice: operatory te zmniejszają (zwiększają) swój argument *przed* jego użyciem do obliczenia wartości wyrażenia, w skład którego wchodzą. Tak więc po

```
int a = 1;
int b = ++a;
```

wartość a wynosi 2, ale wartość b wynosi również 2, bo w trakcie opracowywania drugiej instrukcji zmienna a została zwiększona jeszcze przed wykonaniem przypisania. Wynikiem działania operatora preinkrementacji lub predekrementacji *jest* l-wartość (pamiętamy, że tak *nie* było dla operatorów postinkrementacji lub postdekrementacji) — tak więc wyrażenie `++++a` byłoby legalne.

Operatory negacji: bitowej i logicznej omówimy poniżej razem z innymi operatorami logicznymi i bitowymi.

Operator jednoargumentowy '+' jest właściwie operatorem identycznościowym, czyli takim, który „nic nie robi" (ang. *no-op*). Istnieje tylko dla wygody, aby wyrażenia typu '`k = +5`' miały sens. Operatory wyłuskania adresu i dereferencji (`&` i `*`) były już omówione w rozdz. 4 (str. 27).

Operatory **new** i **delete** służą do dynamicznego alokowania i zwalniania pamięci: będą omówione w rozdz. 12 na stronie 211.

Ostatni operator z tej grupy, oznaczany parą nawiasów okrągłych, to operator rzutowania. Jest on operatorem jednoargumentowym: wynikiem działania tego operatora na p-wartość pewnego typu jest odpowiadająca tej wartości p-wartość innego typu — tego wymienionego w nawiasie. Z operatora tego należy korzystać oględnie; w większości przypadków jego zastosowanie świadczy raczej o złym stylu programowania. Czasem jest jednak przydatny. Na przykład w drugiej instrukcji fragmentu

```
double x = 7;
int k = (int)x;
```

rzutowanie wartości zmiennej x jest wskazane, gdyż wartość ta, jako wartość szerszego typu, może być wpisana do zmiennej typu węższego, jakim jest typ **int**, tylko ze stratą informacji (precyzji). Choć nie jest to błąd, to kompilator zwykle wypisuje ostrzeżenia; jeśli zastosujemy jawne rzutowanie, ostrzeżeń nie będzie.

Rzutowanie zawsze działa na p-wartość i w wyniku daje inną p-wartość, ale nigdy l-wartość. W szczególności rzutowanie nie zmienia typu żadnej zmiennej — typu istniejącej zmiennej zmienić się nie da!

Zamiast operatora rzutowania w C++ zaleca się stosowanie bezpieczniejszych operatorów konwersji, wymienionych w tabeli w grupie odpowiadjącej priorytetowi 16. Omówimy je bardziej szczegółowo w rozdz. 20 na stronie 439.

9.2.4 Grupa operatorów o priorytecie 13

Należą do tej grupy dwa operatory wyboru składowej, które omówimy w rozdziałach na temat klas w C++ (rozdz. 15.6 na stronie 319).

9.2.5 Operatory arytmetyczne

Wymienione w tabeli operatory arytmetyczne mają oczywiste znaczenie i ich działanie jest niemal takie samo jak w większości innych języków. Występują czasem drobne

różnice: np. w Javie operator reszty '%' może mieć dowolne agumenty numeryczne, również typu **double**. W C/C++ operator ten wymaga argumentów typu całkowitego. Pewien kłopot mogą sprawiać wyrażenia z wartościami ujemnymi jako argumentami. Wiemy, że dzielenie liczb typu całkowitego daje w wyniku liczbę całkowitą, czyli ewentualna część ułamkowa jest obcinana. Jeśli wynik jest dodatni, to dokładna wartość ilorazu jest obcinana w dół, czyli w kierunku zera, natomiast jeśli wynik jest ujemny, to obcięcie jest w górę, a więc też w kierunku zera. Dla operatora reszty spełniona jest zawsze zasada

```
a = (a/b)*b + a%b
```

dla b różnego od zera. Wynika z niej, przy założeniu, że obcinanie w dzieleniu całkowitoliczbowym jest zawsze w kierunku zera, zasada następująca: wartość `a%b` jest równa co do modułu wartości `|a|%|b|` i ma znak taki, jaki znak ma `a` (kreski oznaczają wartość bezwzględną). Ilustruje to poniższy programik:

P57: ***mod.cpp*** Operator reszty

```
#include <iostream>
using namespace std;

int main() {
    int    i, j;

    i = 19; j = 7; cout << " 19 /  7 = " << i/j << endl;
    i =-19; j = 7; cout << "-19 /  7 = " << i/j << endl;
    i = 19; j =-7; cout << " 19 / -7 = " << i/j << endl;
    i =-19; j =-7; cout << "-19 / -7 = " << i/j << endl;

    i = 19; j = 7; cout << " 19 %  7 = " << i%j << endl;
    i =-19; j = 7; cout << "-19 %  7 = " << i%j << endl;
    i = 19; j =-7; cout << " 19 % -7 = " << i%j << endl;
    i =-19; j =-7; cout << "-19 % -7 = " << i%j << endl;
}
```

którego wynikiem jest

```
 19 /  7 = 2
-19 /  7 = -2
 19 / -7 = -2
-19 / -7 = 2
 19 %  7 = 5
-19 %  7 = -5
 19 % -7 = 5
-19 % -7 = -5
```

Ta zasada będzie inna, jeśli obcinanie w dzieleniu całkowitoliczbowym jest zawsze w dół, a nie w kierunku zera (tak może sie zdarzyć dla starych kompilatorów). Dlatego lepiej unikać stosowania operatora reszty dla liczb ujemnych.

9.2.6 Operatory relacyjne i porównania

Operatory relacyjne ('<', '<=', '>', '>=') i porównania ('==', '!=') mają oczywistą interpretację. Wyrażenie 'a == b' ma wartość logiczną odpowiadającą na pytanie *czy wartość* a *jest równa wartości* b. Wyrażenie 'a != b' ma wartość logiczną odpowiadającą na pytanie *czy wartość* a *jest różna od wartości* b. Argumentami mogą być dwa skalary, czyli p-wartości liczbowe, lub (choć nie zawsze) dwa adresy (wartości zmiennych wskaźnikowych lub wynik operacji wyłuskania adresu). Jest to nieco inaczej niż w Javie, gdzie adresy (czy raczej odniesienia) mogły być argumentami wyłącznie operatorów porównania ('==' i '!='), ale nie operatorów relacyjnych. W C++ można porównywać adresy za pomocą operatorów relacyjnych pod warunkiem, że są to adresy elementów tej samej tablicy.

Wynikiem operacji jest wartość logiczna **true** lub **false**. Jak mówiliśmy (patrz rozdz. 4.4, str. 36), wartości logiczne reprezentowane są w zasadzie przez wartości całkowite: wartość 0 jest równoważna **false**, a dowolna wartość niezerowa **true**. Obowiązuje to również dla wartości wskaźnikowych: wartość pusta (**nullptr**) jest interpretowana jako **false**, a każda inna jako **true**.

9.2.7 Operatory bitowe

Operatory bitowe wymienione są w tabeli na pozycjach odpowiadających priorytetom 11 (przesunięcia bitowe), 8, 7 i 6. Argumentami muszą być wartości całkowite, wynik też jest typu całkowitego. Obowiązują przy tym normalne reguły konwersji argumentów do typu wspólnego.

Operatory bitowe nie „interesują" się wartościa liczbową argumentów, ale ich reprezentacją bitową. Przypomnijmy, że zwyczajowo numeruje się bity reprezentujące wartości zmiennych, poczynając od zera, od bitu najmniej znaczącego (odpowiadającego współczynnikowi przy zerowej potędze dwójki) do bitu najbardziej znaczącego. Reprezentując graficznie układ bitów, bit zerowy umieszcza się po prawej stronie, a bit najbardziej znaczący po lewej (patrz rozdz. 4.2 na stronie 31).

Rozpatrzmy zatem bardziej szczegółowo działanie poszczególnych operatorów bitowych na wartości liczbowe. Jeden z nich — negacja bitowa — jest jednoargumentowy, pozostałe są dwuargumentowe.

Operator bitowej negacji ('~'), działając na wartość całkowitą zwraca nową wartość, w której wszystkie bity ulegają odwróceniu: tam, gdzie w argumencie był

$$w\ \boxed{1}\boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{0}$$
$$\tilde{}w\ \boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{0}\boxed{0}\boxed{1}$$

bit ustawiony (czyli miał umowną wartość 1), w wartości wynikowej będzie on nieustawiony (co odpowiada umownej wartości 0) — jak na rysunku, gdzie dla uproszczenia zilustrowane jest działanie opertatora negacji bitowej dla wartości typu **char**, a więc jednobajtowej (ośmiobitowej). Oczywiście negacja jest inwolutywna, czyli dwukrotne jej zastosowanie prowadzi do wartości wyjściowej.

Alternatywa bitowa ('|') jest operatorem dwuargumentowym: dla kolejnych pozycji sprawdzane są pojedyncze bity w obu argumentach i obliczana ich suma logiczna: w wyniku bit na odpowiadającej pozycji jest jedynką (bit ustawiony), jeśli w którymkolwiek argumencie bit na tej pozycji był ustawiony, a zero, gdy w obu argumentach na tej pozycji również występowało zero (jak na rysunku poniżej).

$$\begin{array}{rr} v & 10001110 \\ w & 00010101 \\ \hline v|w & 10011111 \end{array}$$

Alternatywa bitowa (tzw. ORowanie) jest często stosowana do ustawiania najrozmaitszych opcji. Na przykład w C++, otwarte do czytania lub pisania pliki mają szereg trybów, którym odpowiadają pewne stałe całkowite, np. ios::in, ios::out, zdefiniowane w klasie **ios** (dlatego odwołujemy się do nich poprzez operator zakresu klasy — czterokropek). W programie ***bits.cpp*** (str. 134) drukujemy reprezentację bitową kilku tego rodzaju stałych. Widzimy, że są to pełne potęgi dwójki, a więc w ich reprezentacji bitowej występuje tylko jedna jedynka na odpowiedniej pozycji — dla ios::out na pozycji 4, dla ios::app na pozycji 0 itd. Zatem na przykład stałą określającą tryb otwartego pliku jako pliku jednocześnie do pisania i do czytania będzie ios::in | ios::out i będzie zawierać jedynki na pozycjach 3 i 4 (konkretne pozycje mogą zależeć od implementacji — należy zawsze odwoływać się do tych stałych poprzez ich nazwy).

Koniunkcja bitowa ('&') jest też operatorem dwuargumentowym: dla kolejnych pozycji sprawdzane są pojedyncze bity w obu argumentach i obliczany ich iloczyn logiczny: w wyniku bit na odpowiadającej pozycji jest jedynką (bit ustawiony), jeśli w obu argumentach bit na tej pozycji był ustawiony, a zero, gdy w którymkolwiek z argumentów na tej pozycji występowało zero (patrz rysunek).

$$\begin{array}{rr} v & 10001111 \\ w & 00010101 \\ \hline v\&w & 00000101 \end{array}$$

Koniunkcja bitowa (tzw. ANDowanie) jest często stosowana do tzw. maskowania. Wspomnieliśmy, że stała określająca tryb pliku ma na pozycji trzeciej (licząc od zera) jedynkę, jeśli ustawiony plik został otwarty w trybie in, a zero, jeśli nie (co to dokładnie znaczy, dowiemy się w rozdz. 16 na stronie 323).

Jeśli stała określająca tryb nazywa się tryb, to maskowanie jej ze stałą 8 ($= 2^3$) odpowie na pytanie, czy bit in jest czy nie jest ustawiony. Reprezentacja 8 składa się z samych zer, z wyjątkiem pozycji trzeciej (czwarty bit od prawej), na której bit jest ustawiony. Zatem obliczając koniunkcję dostaniemy na wszystkich innych pozycjach na pewno zero, na pozycji czwartej zaś jedynkę, jeśli w tryb ten bit był ustawiony, a zero, jeśli nie był. Zatem wartość wyrażenia tryb & 8 będzie niezerowa wtedy i tylko

wtedy, jeśli bit in był w zmiennej tryb ustawiony, niezależnie od stanu innych bitów w tej zmiennej.

Operator **bitowej różnicy symetrycznej** ('^') jest też operatorem dwuargumentowym: dla kolejnych pozycji sprawdzane są pojedyncze bity w obu argumentach i obliczana jest ich różnica symetryczna: wynikowy bit na odpowiadającej pozycji jest jedynką (bit ustawiony), jeśli w obu argumentach bity na tej pozycji były *różne*, a zerem jeśli w obu argumentach na tej pozycji występowały bity takie same — dwa zera albo dwie jedynki.

$$\begin{array}{rr} v & \boxed{1}\boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{1} \\ w & \boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{1} \\ \hline v \hat{} w & \boxed{1}\boxed{0}\boxed{0}\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{0} \end{array}$$

Różnica symetryczna (obliczanie jej nazywane bywa XORowaniem) ma ciekawą i użyteczną własność, wynikającą z natępującej tabelki logicznej dla tego działania:

b	m	b^m	(b^m)^m
1	1	0	1
1	0	1	1
0	1	1	0
0	0	0	0

z której wynika, że *dwukrotne* XORowanie dowolnego bitu b z dowolną maską m przywraca pierwotną wartość tego bitu — w tabeli kolumna pierwsza i ostatnia są takie same. Ta własność XORowania jest wykorzystywana między innymi w grafice komputerowej.

Przesunięcia bitowe ('<<' i '>>') są operatorami dwuargumentowymi: lewy argument jest tu pewną wartością całkowitą na bitach której operacja jest przeprowadzana, prawy argument, również całkowity, określa wielkość przesunięcia. Wyobraźmy sobie, że lewy argument w ma układ bitów jak w górnej części rysunku. Przesunięcie w tej zmiennej bitów w lewo o dwa, '`w = w << 2`', odpowiada przesunięciu wszystkich bitów o dwie pozycje w lewo (w kierunku pozycji bardziej znaczących). Bity „wychodzące" z lewej strony są tracone bezpowrotnie. Z prawej strony „wchodzą" bity zerowe. Tak więc po wykonaniu tej instrukcji reprezentacja zmiennej w będzie taka jak w środkowej części rysunku. Analogicznie, po przesunięciu teraz bitów w prawo o trzy pozycje ('`w = w >> 3`') otrzymamy reprezentację zmiennej w jak w dolnej części rysunku (pod pewnymi warunkami — patrz niżej).

$$\begin{array}{rr} w & \boxed{1}\boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{0} \\ w = w \ll 2 & \boxed{0}\boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{0}\boxed{0} \\ w = w \gg 3 & \boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{1}\boxed{1} \end{array}$$

O ile przesuwanie w lewo jest zawsze dobrze określone według wspomnianych zasad, rzecz jest bardziej skomplikowana przy przesuwaniu w prawo. Wychodzące z prawej strony bity są tracone, tak jak te z lewej strony przy przesuwaniu w lewo. Ale nie jest jasne, jakie bity „wchodzą" z lewej strony przy przesuwaniu w prawo. W Javie istnieją dwa operatory bitowego przesunięcia w prawo: w przypadku operatora '>>' z lewej strony „wchodzi" taki bit, jaki był przed przesunięciem na najstarszej pozycji (czyli z lewej strony) — jeśli było to zero, wchodzi zero, jeśli jedynka, wchodzi jedynka. Mówimy, że reprodukowany jest bit znaku. Taka konwencja powoduje, że dla liczb ze znakiem (patrz rozdz. 4.2 na stronie 31) przesunięcie w prawo o jedną pozycję odpowiada dla liczb parzystych dzieleniu przez dwa, zarówno dla liczb dodatnich, jak i ujemnych — podobnie jak przesuwanie w lewo odpowiada mnożeniu przez potęgę dwójki (dla liczb nieparzystych i dla przesuwania w prawo jest tu pewna dodatkowa komplikacja, w którą nie będziemy się wgłębiać, jest to dobre ćwiczenie sprawdzające rozumienie bitowych reprezentacji liczb; ma to związek z konwencją obcinania części ułamkowej przy dzieleniu całkowitoliczbowym — w dół czy w kierunku zera?). Inny operator w Javie, '>>>', oznacza przesunięcia w prawo takie, że z lewej strony wchodzą zawsze zera.

W C/C++ sprawa nie jest taka jasna. Istnieją tu specjalne typy bez znaku (**unsigned**), za to nie istnieje „potrójny" operator przesunięcia '>>>' w prawo.

Wobec tego przyjęto następującą konwencję: jeśli typem wartości lewego argumentu jest typ *bez* znaku (**unsigned**), to przy przesuwaniu w prawo wchodzą z lewej strony bity zerowe; jeśli natomiast typem wartości lewego argumentu jest typ *ze* znakiem (**signed**), to przy przesuwaniu w prawo reprodukowany jest z lewej strony bit znaku, czyli skrajny lewy bit — jeśli było to zero, to zero, jeśli była to jedynka, to jedynka.

Prawy argument operatora przesunięcia, wskazujący na wielkość tego przesunięcia, zawsze powinien być nieujemny i mniejszy od rozmiaru w bitach wartości, na której dokonujemy przesunięcia. W Javie jest dobrze określone przez specyfikację języka, co się dzieje, jeśli te warunki nie są spełnione; w C/C++ może to zależeć od implementacji i wobec tego lepiej takich konstrukcji unikać.

Na zakończenie rozpatrzmy przykład ilustrujący to, o czym mówiliśmy na temat operacji bitowych.

P58: ***bits.cpp*** Operacje na bitach

```
#include <iostream>
using namespace std;

void bitsChar(char k) {
    int bits = 8*sizeof(k);                    ①
    unsigned char mask = 1<<(bits-1);
    for (int i = 0; i < bits; i++) {
        cout << (k & mask ? 1 : 0);
        mask >>= 1;                            ②
    }
    cout << endl;
```

```
12 }
13
14 void bitsShort(short k) {
15     int bits = 8*sizeof(k);
16     unsigned short mask = 1<<(bits-1);
17     for (int i = 0; i < bits; i++) {
18         cout << (k & mask ? 1 : 0);
19         mask >>= 1;
20     }
21     cout << endl;
22 }
23
24 void bitsInt(int k) {
25     int bits = 8*sizeof(k);
26     unsigned int mask = 1<<(bits-1);
27     for (int i = 0; i < bits; i++) {
28         cout << (k & mask ? 1 : 0);
29         mask >>= 1;
30     }
31     cout << endl;
32 }
33
34 int main() {
35     short s = -1; int i = 259;
36
37     cout << "char 'a' : "; bitsChar('a');
38     cout << "short -1 : "; bitsShort(s);
39     cout << "int  259 : "; bitsInt(i);
40     cout << endl;
41     cout << "ios::in  : "; bitsInt(ios::in);
42     cout << "ios::out : "; bitsInt(ios::out);
43     cout << "ios::app : "; bitsInt(ios::app);
44     cout << "ios::in | ios::out\n           ";
45     bitsInt(ios::in | ios::out);
46 }
```

Na początku tego programu definiujemy trzy niemal identyczne funkcje, których zadaniem jest wypisanie bitowej reprezentacji argumentu. Funkcje różnią się tylko typem argumentu: może nim być **char**, **short** lub **int**. W rozdz. 11.14 na stronie 200 dowiemy się, jak można było uniknąć pisania trzech wersji tak podobnych funkcji.

Przyjrzyjmy się jednej z tych funkcji, na przykład funkcji **bitsChar**. W linii ① sprawdzamy, jaki jest rozmiar w bitach wartości argumentu k (tu oczywiście wiemy, że będzie to 8, bo `sizeof(k)` dla k typu **char** da jedynkę). Następnie tworzymy maskę mask typu **unsigned char**. Chodzi nam o to, aby długość maski była taka jak długość k, ale by była to zmienna na pewno *bez* znaku — w ten sposób przy przesuwaniu w prawo będą z lewej strony „wchodzić” zera. Maskę inicjujemy wartością `'1 << 7'`

(bo bits wynosi 8). Reprezentacja jedynki to siedem bitów zerowych i jeden, prawy (czyli najmłodszy), bit ustawiony. Przesuwając ten układ bitów w lewo otrzymamy liczbę, której reprezentacją jest jedynka i siedem zer (jedynka tym razem z lewej strony). Robimy to po to, by pętla drukująca, która teraz następuje, przebiegała przez kolejne bity liczby k od lewej do prawej, a nie odwrotnie.

Następnie w pętli obliczamy koniunkcję bitową k z maską mask. Ponieważ maska ma tylko jeden bit ustawiony, w ten sposób sprawdzamy, czy odpowiedni bit jest też ustawiony w k. Jeśli tak, to wynikiem koniunkcji będzie jakaś wartość niezerowa, interpretowana jako **true** w pierwszym argumencie operatora selekcji, a zatem wartością tejże selekcji '`(k&mask ? 1 : 0)`' będzie jedynka, która zostanie wypisana na ekranie. Jeśli w k odpowiedni bit nie jest ustawiony, wydrukowane zostanie zero.

W linii ② przesuwamy bity w masce o jeden w prawo. Ponieważ zadbaliśmy, aby maska była typu *bez* znaku, z lewej strony będą wchodzić same zera. Zatem w kolejnych przebiegach pętli maska cały czas będzie zawierać dokładnie jedną jedynkę, „wędrującą" od lewej do prawej. A więc w kolejnych przebiegach pętli sprawdzone i wydrukowane będą, w kolejności od lewej do prawej, wszystkie bity zmiennej k.

Podobnie działają pozostałe dwie funkcje: **bitShort** i **bitInt** — jedyna różnica to typ argumentu.

W programie głównym drukujemy reprezentację bitową kilku liczb całkowitych. Wyniki wyglądają następująco:

```
char 'a' : 01100001
short -1 : 1111111111111111
int  259 : 00000000000000000000000100000011

ios::in  : 00000000000000000000000000001000
ios::out : 00000000000000000000000000010000
ios::app : 00000000000000000000000000000001
ios::in | ios::out
           00000000000000000000000000011000
```

Widać, że (jak o tym mówiliśmy w rozdz. 4.2), reprezentacją liczby -1 są jedynki na wszystkich bitach. Znak 'a' odpowiada, jak łatwo policzyć, wartości całkowitej $2^6 + 2^5 + 1 = 64 + 32 + 1 = 97$, co rzeczywiście jest kodem ASCII litery 'a'.

Dalej ilustrujemy to, co mówiliśmy o stałych ios::in, ios::out itd. Widzimy, że są to pełne potęgi dwójki, a więc w ich reprezentacji bitowej występuje tylko jedna jedynka na odpowiedniej pozycji: w stałej ios::trunc na pozycji czwartej (licząc od zera), a w stałej ios::out na pozycji pierwszej. Obliczając ich alternatywę (sumę) bitową otrzymujemy liczbę, w której reprezentacji bitowej te i tylko te dwa bity są ustawione (ostatnia linia wydruku).

Inny przykład: tu upakowujemy cztery liczby z zakresu $[0, 255]$, a więc mieszczące się w jednym bajcie (jak składowe red, green, blue i alfa coloru) do jednej zmiennej 32-bitowej. Funkcja **encode** upakowuje te składowe przez ORowanie i przesuwanie w lewo, a funkcja **decode** je „odpakowuje" i wyświetla:

P59: ***bitColors.cpp*** Upakowywanie składowych RGBA koloru w jednej 32-bitowej zmiennej

```cpp
#include <iostream>
#include <cstdint>

std::uint32_t encode(int r, int g, int b, int a) {
    std::uint32_t n = a;
    n = (n << 8) | b;
    n = (n << 8) | g;
    n = (n << 8) | r;
    return n;
}

void decode(std::uint32_t n) {
    std::cout <<   "r = " << (n & 0xFF);
    n >>= 8;
    std::cout << ", g = " << (n & 0xFF);
    n >>= 8;
    std::cout << ", b = " << (n & 0xFF);
    n >>= 8;
    std::cout << ", a = " << (n & 0xFF) << '\n';
}

int main() {
    decode(encode(23, 44, 129, 255));
}
```

Program drukuje

```
r = 23, g = 44, b = 129, a = 255
```

W standardzie C++20 dodano (w nagłówku ***bit***) szereg nowych operacji bitowych, między innymi rotacje (funkcje **rotl** i **rotr**). Mogą one być zastosowane tylko dla typów **unsigned** i polegają na rotacji bitów w lewo (co wychodzi z lewej, wchodzi od prawej) lub w prawo (co wychodzi z prawej, wchodzi od lewej). Na przykład (tu funkcja **showBits** jest szablonem):

P60: ***rotLR.cpp*** Rotacje bitów

```cpp
#include <bit>      // std::rotl, std::rotr
#include <cstdint>
#include <iostream>

template <typename T>
std::string showBits(T t) {
    size_t sz = 8*sizeof(T);
    std::string s(sz, ' ');
```

```
9       for (size_t i = 0, j = sz-1; i < sz; ++i, --j)
10          s[j] = (t & (1 << i)) ? '1' : '0';
11      return s;
12  }
13
14  // C++20 required
15
16  int main() {
17      using std::cout; using std::rotr; using std::rotl;
18
19      std::uint8_t n = 0b01001101;
20      cout << "n          : " << showBits(n) << '\n';
21      cout << "n rotl by 2: " << showBits(rotl(n, 2)) << '\n';
22      cout << "n rotl by 3: " << showBits(rotl(n, 3)) << '\n';
23      cout << "n rotr by 2: " << showBits(rotr(n, 2)) << '\n';
24      cout << "n rotr by 3: " << showBits(rotr(n, 3)) << '\n';
25  }
```

drukuje

```
n          : 01001101
n rotl by 2: 00110101
n rotl by 3: 01101010
n rotr by 2: 01010011
n rotr by 3: 10101001
```

9.2.8 Operatory logiczne

Argumentami operatorów logicznych '&&' (koniunkcja), '||' (alternatywa) i '!' (negacja) mogą być zarówno wartości typu **bool** jak i wartości całkowite; w tym ostatnim przypadku wartości zostaną zinterpretowane według normalnej zasady: 0 → **false**, niezero → **true**. Obliczona wartość jest typu logicznego: alternatywa (suma logiczna) daje wynik **true** wtedy i tylko wtedy gdy choć jeden z argumentów ma wartość **true**, natomiast koniunkcja (iloczyn logiczny) ma wartość **true** tylko jeśli oba argumenty są **true**.

Koniunkcja i alternatywa są skrótowe. Oznacza to, że prawy argument nie jest w ogóle obliczany, jeśli po obliczeniu lewego wynik jest już przesądzony. Tak więc

- dla koniunkcji ('&&') prawy argument nie będzie w ogóle obliczany, jeśli lewy argument okazał się równy**false** — całe wyrażenie musi bowiem wtedy mieć wartość **false** niezależnie od wartości prawego argumentu;
- dla alternatywy prawy argument nie będzie obliczany, jeśli lewy okazał się **true** — całe wyrażenie musi bowiem mieć wtedy wartość **true** niezależnie od wartości prawego argumentu.

Na przykład, jeśli a, b i r są zmiennymi typu całkowitego, to przypisanie

```
r = a && b;
```

jest równoważne

```
if (a == 0)
    r = 0;
else
{
    if (b == 0) r = 0;
    else        r = 1;
}
```

a przypisanie

```
r = a || b;
```

jest równoważne

```
if (a != 0)
    r = 1;
else
{
    if (b != 0) r = 1;
    else        r = 0;
}
```

Rozpatrzmy jeszcze jeden przykład ilustrujący skrótowość dwuargumentowych operatorów logicznych:

P61: ***skrot.cpp*** Skrótowość operatorów koniunkcji i alternatywy

```
1  #include <iostream>
2  using namespace std;
3
4  bool fun(int k) {
5      k = k - 3;
6      cout << "Fun zwraca " << k << endl;
7      return k;
8  }
9
10 int main() {
11     if ( fun(1) && fun(2) && fun(3) && fun(4) )       ①
12         cout << "Koniunkcja true"   << endl;
13     else
14         cout << "Koniunkcja false"  << endl;
15
16     if ( fun(1) || fun(2) || fun(3) || fun(4) )       ②
```

```
17          cout << "Alternatywa true"  << endl;
18      else
19          cout << "Alternatywa false" << endl;
20  }
```

W linii ① sprawdzany jest warunek w postaci koniunkcji wartości logicznych zwracanych przez funkcję **fun** (równie dobrze mogłyby to być wartości całkowite). Ponieważ `fun(3)` zwraca 0 czyli logiczne **false**, funkcja w ogóle nie zostanie już wywołana z argumentem 4, bo wynik już jest znany: wartością całego wyrażenia w nawiasie musi być **false** niezależnie od tego, co zwróciłaby funkcja **fun** dla argumentu 4. Widzimy to z wydruku

```
Fun zwraca -2
Fun zwraca -1
Fun zwraca 0
Koniunkcja false
Fun zwraca -2
Alternatywa true
```

Podobnie dla alternatywy w linii ②. Już pierwsze wywołanie funkcji **fun** dało wynik **true** (odpowiada to niezerowej wartości zwracanej, w tym przypadku -2). Alternatywa jest prawdziwa, gdy choć jeden argument jest **true**, wobec tego po wywołaniu `fun(1)` wynik całego wyrażenia jest znany (**true**) i wywołań `fun(2)`, `fun(3)` i `fun(4)` nie będzie.

9.2.9 Operatory przypisania

W grupie o priorytecie 2 wymienione są operatory zwykłego przypisania ('=') oraz złożone operatory przypisania.

Lewa strona przypisania musi być zawsze l-wartością. Tak więc

```
double x;
x + 2 = 7; // NIE
```

jest niepoprawne, natomiast

```
double x, *y = &x;
*(y + 2) = 7;
```

byłoby legalne (choć prawdopodobnie bez sensu), bo wyłuskanie wartości daje w wyniku l-wartość.

Wykonanie przypisania polega na obliczeniu wartości prawej strony i umieszczeniu wyniku pod adresem l-wartości występującej po stronie lewej. Zauważmy asymetrię: prawa strona mówi *co* policzyć, lewa *gdzie* zapisać wynik.

Wartością i typem całego wyrażenia przypisania jest wartość i typ lewej strony *po* wykonaniu przypisania. Całe przypisanie *jest* l-wartością. Na przykład

```
int m = 1, n = 2;
(m=n) = 6;
cout << m << " " << n << endl;
```

drukuje '6 2'.
W programie poniżej

P62: *przypis.cpp* Wykorzystanie wartości przypisania

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int k;
7      while ( (k = cin.get()) != '\n' )
8          cout << "Wprowadzono znak '" << (char)k
9               << "', o kodzie ASCII " << k << endl;
10 }
```

w linii 7 przypisujemy do k wartośc znaku — czyli jego kod ASCII — odczytaną z klawiatury za pomocą metody **get** wywołanej na rzecz obiektu cin — patrz rozdz. 16.4.1 na stronie 338. Całe przypisanie '(k=cin.get())' ma wartość k po przypisaniu; tę wartość porównujemy z predefiniowaną stałą EOF, oznaczającą koniec strumienia danych. Zauważmy, że nawias wokół tego wyrażenia był konieczny, bowiem priorytet porównania, !=, jest wyższy niż priorytet operatora przypisania, a nam chodzi o to, aby najpierw dokonać przypisania, a dopiero jego wynik porównać z EOF. Przykładowe uruchomienie tego programu daje

```
cpp> g++ -pedantic-errors -Wall -o przypis przypis.cpp
cpp> ./przypis
Ala ma...[ENTER]
Wprowadzono znak 'A', o kodzie ASCII 65
Wprowadzono znak 'l', o kodzie ASCII 108
Wprowadzono znak 'a', o kodzie ASCII 97
Wprowadzono znak ' ', o kodzie ASCII 32
Wprowadzono znak 'm', o kodzie ASCII 109
Wprowadzono znak 'a', o kodzie ASCII 97
Wprowadzono znak '.', o kodzie ASCII 46
Wprowadzono znak '.', o kodzie ASCII 46
Wprowadzono znak '.', o kodzie ASCII 46
cpp>
```

Dzięki temu, że wartością całego wyrażenia z przypisaniem jest wartość lewej strony po jego wykonaniu, przypisania można stosować kaskadowo. Tak więc

```
int k = 7, j, m;
int n = m = j = k;
```

jest prawidłowe: ponieważ wiązanie operatora przypisania jest od prawej, wartością wyrażenia 'm=j=k', równoważnego 'm=(j=k)', jest wartość m po przypisaniu (czyli w naszym przypadku 7). Ta wartość zostanie użyta do zainicjowania definiowanej zmiennej n. Efektem ubocznym będzie nadanie wartości również zmiennym m i j. Zauważmy, że instrukcja byłaby błędna, gdyby któraś ze zmiennych m, j, k nie była utworzona wcześniej.

Złożone operatory przypisania pozwalają na prostszy zapis niektórych przypisań: tych, w których ta sama l-wartość występuje po lewej i prawej stronie przypisania. Zamiast instrukcji

```
a = a @ b;
```

gdzie symbol '@' oznacza któryś z operatorów

$$\begin{array}{ccccc} + & - & * & / & \% \\ \ll & \gg & \& & \hat{} & | \end{array}$$

można użyć instrukcji

```
a @= b;
```

Drobna różnica, najczęściej bez znaczenia, pomiędzy tymi instrukcjami polega na tym, że w drugiej z nich wartość a jest obliczana jednokrotnie, a w pierwszej dwukrotnie. Zwykle druga z tych form, 'a @= b', jest efektywniejsza.

Jako przykład zastosowania rozpatrzmy program:

P63: *zloz.cpp* Przypisania złożone

```
#include <iostream>
using namespace std;

void bitsInt(int k) {
    unsigned int mask = 1<<31;
    for (int i = 0; i < 32; i++, mask >>= 1) {
        cout << (k & mask ? 1 : 0);
        if (i%8 == 7) cout << " ";
    }
    cout << endl;
}

int main() {
    unsigned int k = 255<<24 | 153<<16 | 255<<8 | 255; ①
    cout << "k przed: "; bitsInt(k);
    (k <<= 8) >>= 24;                                  ②
    cout << "k    po: "; bitsInt(k);
}
```

Funkcja **bitsInt** jest tu podobna do tej z programu ***bits.cpp*** (str. 134) — tak samo służy do drukowania bitowej reprezentacji liczby całkowitej. W tej wersji z góry założyliśmy, że typ **int** jest czterobajtowy. Prócz tego przesuwanie maski przenieśliśmy do części inkrementującej nagłówka pętli, umieszczając tam dwie instrukcje wyrażeniowe oddzielone przecinkiem (o operatorze przecinkowym — patrz podrozdział 9.2.12). Dodaliśmy też drukowanie spacji po każdej grupie ośmiu bitów, aby uczynić wydruk bardziej przejrzystym.

W linii ① konstruujemy liczbę o z góry zadanej reprezenatcji bitowej. Wyrażenie '`255 << 24`' daje liczbę z samymi jedynkami w najstarszym bajcie (255 to osiem jedynek, następnie przesunięte o 24 pozycje w lewo). Wyrażenie '`153 << 16`' to układ bajtów `10011001` przesunięty w lewo o 16 pozycji, czyli do bajtu trzeciego od lewej. Z kolei '`255 << 8`' daje osiem jedynek w bajcie drugim, a samo 255 — osiem jedynek w bajcie najmłodszym. Suma (alternatywa) bitowa „składa" wszystkie te bajty: otrzymujemy zatem liczbę o reprezentacji przedstawionej w pierwszej linii wydruku:

```
k przed: 11111111 10011001 11111111 11111111
k    po: 00000000 00000000 00000000 10011001
```

Operator przypisania złożonego zastosowaliśmy w linii ②. Wyrażenie '`(k <<= 8)`' powoduje przesunięcie w zmiennej k wszystkich bitów w lewo o osiem pozycji. Zatem zawartość bajtu najstarszego zostaje „zgubiona", bajt `10011001` przechodzi na jego pozycję, a kolejne dwa, czyli pierwszy i drugi, stają się drugim i trzecim (od lewej). Bajt najmłodszy wypełniany jest zerami. Wynik całego wyrażenia jest l-wartością, a zatem ma sens zastosowanie do niego następnego przypisania złożonego: tym razem przesuwamy zawartość zmiennej k o 24 pozycje w prawo. Ponieważ zmienna k jest bez znaku, z lewej strony „wchodzą" same zera, a 24 najmłodsze bity „wychodzą" z prawej strony. Układ bitów `10011001` po tej operacji znajduje się na pozycji najmłodszego bajtu. W efekcie widzimy, że cała operacja daje w efekcie liczbę równą tej, której reprezentacja binarna zawarta była w trzecim od lewej bajcie wyjściowej liczby. W podobny sposób moglibyśmy „wyciąć" zawartość pozostałych bajtów. Takie wycinanie pojedynczych bajtów stosuje się na przykład przy kodowaniu trzech (albo czterech) składowych koloru w jednej liczbie.

9.2.10 Operator warunkowy

Operator warunkowy (selekcji) jest jedynym operatorem trzyargumentowym. Jego składnia:

```
b ? w1 : w2
```

Najpierw obliczana jest wartość wyrażenia b i ewentualnie konwertowana do typu **bool**. Jeśli jest to **true**, obliczane jest wyrażenie w1, a wyrażenie w2 jest ignorowane. Wartością całego wyrażenia jest wartość w1. Jeśli jest to **false**, obliczane jest wyrażenie w2, a wyrażenie w1 jest ignorowane. Wartością całego wyrażenia jest wtedy wartość w2. Jeśli zarówno w1 i w2 są l-wartościami, to i wartość operatora warunkowego jest l-wartością.

Klasyczny przykład zastosowania operatora selekcji to implementacja funkcji **max** zwracającej większą z wartości swych argumentów:

```
int maxim(int a, int b) {
    return a > b ? a : b;
}
```

Inny przykład zastosowania operatora selekcji podany zostanie w następnym podrozdziale.

9.2.11 Operator zgłoszenia wyjątku

Jako przedostatni w tabeli występuje operator zgłoszenia wyjątku **throw**: omówimy go w rozdz. 22 na stronie 491.

9.2.12 Operator przecinkowy

Operator przecinkowy jest dwuargumentowy: po dwóch stronach przecinka dwa wyrażenia

```
wyr1 , wyr2
```

Działanie jego polega na:

- obliczeniu wyrażenia wyr1 i zignorowaniu resultatu;
- obliczeniu wyrażenia wyr2; jego wartość staje się wartością całego wyrażenia.

Często operator przecinkowy stosuje się w części inicjalizacyjnej lub inkrementującej nagłówka pętli **for**; przykład to linia 7 programu ***zloz.cpp*** (str. 142).

Inny, nieco dziwaczny, przykład ilustruje program:

P64: ***przec.cpp*** Operator przecinkowy

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int r = 0;
6      int k;
7
8      while (cin >> k, k) {                                    ①
9          r += k > 0 ? (cout << "Dodatnia\n" , +1)
10                    : (cout << "Ujemna\n"   , -1);
11     }
12     cout << "Roznica ilosci dodatnich i ujemnych : "
13          << r << endl;
14 }
```

Operator przecinkowy jest tu użyty w linii ① w warunku pętli **while**. Mamy tu pierwsze wyrażenie, 'cin >> k', wczytujące kolejną daną z klawiatury, i drugie, po prostu k, dzięki któremu pętla skończy się, gdy wczytana zostanie liczba 0 (gdyż wartością całego wyrażenia przecinkowego jest wartość prawego argumentu). W pętli do r dodawana jest (operator złożonego przypisania) wartość nieco skomplikowanego wyrażenia. Jest to operator selekcji — patrz podrozdział 9.2.10 — sprawdzający znak k; w każdym przypadku rezultat będzie znów wartością wyrażenia przecinkowego. Dla k dodatnich będzie to wartość

```
(cout << "Dodatnia\n" , +1)
```

czyli +1 z efektem ubocznym polegającym na wypisaniu słowa "Dodatnia". Analogicznie, dla k ujemnych od r odjęte zostanie 1, a jako efekt uboczny wypisane będzie słowo "Ujemna". Po wyjściu z pętli wypisywana jest różnica między ilością wczytanych liczb dodatnich i ujemnych. Na przykład wynik programu może być następujący:

```
cpp> przec
2
Dodatnia
6
Dodatnia
-3
Ujemna
6
Dodatnia
-1
Ujemna
3
Dodatnia
0
Roznica ilosci dodatnich i ujemnych : 2
cpp>
```

9.2.13 Alternatywne nazwy operatorów

Niektóre operatory mają też formę czysto tekstową:

Tablica 9.2: Alternatywne nazwy operatorów

tekstowa	symboliczna	tekstowa	symboliczna
and	&&	and_eq	&=
bitand	&	bitor	|
compl	~	not	!
not_eq	!=	or	||
or_eq	|=	xor	^
xor_eq	^=		

Forma tekstowa może być zamiennie stosowana z formą wyrażoną za pomocą symboli nieliterowych.

— 10 —

Konwersje niejawne, porządek wartościowania

W następnym rozdziale zajmiemy się funkcjami. Aby jednak zrozumieć proces wywołania funkcji i zasady wyboru odpowiedniej funkcji w przypadku przeciążania nazw, musimy najpierw omówić standardowe (niejawne) konwersje, które, niejako za naszymi plecami, dokonywane są podczas kompilacji i wykonania programu. Przy tej okazji powiemy też o tzw. porządku wartościowania, który często nie ma dla nas znaczenia, ale czasem trzeba o nim pamiętać, aby uniknąć niemal niemożliwych do wykrycia błędów. Szerzej o konwersjach powiemy w rozdz. 20 na stronie 439.

PODROZDZIAŁY:

10.1 Konwersje standardowe

Rozważmy niewinnie wyglądający fragment programu:

```
double x = 1.5, y;
int    k =  10;
    // ...
y = x + k;
```

Jaka funkcja wykonująca dodawanie będzie wywołana, aby wykonać polecenie 'x+k' z ostatniej linii? Zmienna k jest typu **int**, zmienna x zaś typu **double**. Mają one zatem zupełnie inną strukturę w pamięci komputera. Aby je prawidłowo dodać, należy oczywiście tę strukturę uwzględnić i zdecydować, jaki z kolei ma być typ wyniku. Czy istnieje zatem jakaś funkcja realizująca takie dodawanie? A gdyby k była typu **unsigned short**? Dodawanie liczby zapisanej w postaci **double** do liczby zapisanej w postaci **unsigned short** jest oczywiście zupełnie czym innym niż dodawanie **double**'a do **int**'a. Musiałaby zatem istnieć ogromna liczba funkcji realizujących to samo zadanie, ale inaczej, w zależności od typów argumentów. To samo dotyczy wywołań funkcji. Dostępna po dołączeniu pliku nagłówkowego ***cmath*** (albo ***math.h***) funkcja **sin** ma jeden parametr typu **double** (lub **long**). Czy oznacza to, że wywołanie `sin(1)` jest błędem, czy może istnieje inna wersja tej funkcji z parametrem **int**? Tymi właśnie problemami teraz się zajmiemy.

Załóżmy, że w programie występuje wyrażenie z operatorem dwuargumentowym

(dodawanie, mnożenie, ...). Przed wykonaniem operacji dokonywane są na wartościach argumentów **konwersje standardowe**, których celem jest:

- doprowadzenie do tego, aby typy argumentów były takie same;
- zapewnienie, aby ten wspólny typ był „obsługiwany" przez istniejącą funkcję realizującą działanie danego operatora;
- zachowanie, *w miarę możności*, precyzji wyniku.

Cel pierwszy wynika z faktu, że funkcje realizujące działanie operatorów są zdefiniowane tylko dla argumentów takiego samego typu, przy czym *nie* są zdefiniowane dla wszystkich możliwych typów. Ważne jest też, jakiego typu jest wtedy wynik.

Typem wyniku arytmetycznej operacji dwuargumentowej jest wspólny typ argumentów *po* dokonaniu konwersji.

Wyjaśnijmy przebieg poszukiwania tego wspólnego dla wartości obu argumentów typu. Zauważmy, że konwersje są, jeśli to tylko możliwe, promocjami, to znaczy typ „węższy" jest awansowany do typu „szerszego" tak, żeby nie utracić dokładności. W tym sensie np. typ **int** jest „węższy" od **double**, bo każda wartość całkowita może być zapisania w formie **double** bez utraty dokładności, ale nie odwrotnie. Z drugiej strony, nie dla wszystkich pokrewnych typów można zdefiniować taką relację zawierania: zbiory wartości typu **int** i typu **unsigned int** są tak samo liczne (2^{32} elementów), ale żaden nie zawiera się w drugim. Wrócimy do tego problemu w dalszym ciągu.

Zatem:

1. Jeśli jeden z argumentów jest **long double**, to drugi jest też przekształcany do typu **long double** (oczywiście jeśli już nie był tego typu).
2. W przeciwnym przypadku: jeśli jeden jest typu **double**, to drugi też jest przekształcany do typu **double**.
3. W przeciwnym przypadku: jeśli jeden jest typu **float**, to drugi też jest przekształcany do typu **float**.
4. W przeciwnym przypadku oba argumenty są typów całościowych i są poddawane promocji całościowej zgodnie z następującą procedurą:
   - Wartości typu **signed char**, **unsigned char**, **signed short** i **unsigned short** są przekształcane do typu **int**, jeśli wartość typu **int** może reprezentować wszystkie wartości tych typów (co *zachodzi* dla „zwykłych", 32-bitowych maszyn). W przeciwnym przypadku są przekształcane do typu **unsigned int**, co oczywiście *może* zmienić ich wartość!
   - Wartości typu wyliczeniowego są przekształcane do najwęższego z typów **int**, **unsigned int**, **long**, **unsigned long**, który jest dostatecznie obszerny, aby reprezentować wszystkie wartości wyliczenia.

- Pola bitowe (rozdz. 14.10, str. 287) są przekształcane do typu **int** lub jeśli ten typ nie może reprezentować wszystkich wartości pola — do typu **unsigned int**.
- Wartości typu **bool** są przekształcane do wartości 0 (**false**) i 1 (**true**) typu **int**.

5. Następnie: Jeśli jeden z argumentów jest typu **unsigned long**, to drugi też jest przekształcany do typu **unsigned long**.

6. W przeciwnym razie: jeśli jeden z argumentów jest typu **long** a drugi typu **unsigned int** i jeśli wartości typu **unsigned int** można w danej implementacji reprezentować w typie **long** (co zwykle *nie* zachodzi na „zwykłych" maszynach 32-bitowych), to wartość typu **unsigned int** jest przkształcana do typu **long**; w przeciwnym razie oba argumenty są przekształcane do typu **unsigned long** (co może spowodować nieszczęście...).

7. W przeciwnym razie, jeśli jeden z argumentów jest typu **long**, to drugi jest też przekształcany do typu **long**.

8. W przeciwnym razie, jeśli jeden z argumentów jest typu **unsigned int**, to drugi jest też przekształcany do typu **unsigned int**.

9. W przeciwnym razie oba argumenty są typu **int** i żadna konwersja nie jest potrzebna.

Na przykład, zgodnie z tymi zasadami, wyrażenie w trzeciej linii fragmentu

```
int i = 1, j = 2, k = 3, m;
// ...
m = (j > i) + (k > j);
```

jest prawidłowe: wartości typu **bool**, jakimi są wartości wyrażeń `(j > i)` i `(k > j)`, zostaną niejawnie przekształcone do wartości całkowitych (jedynek, bo obie relacje są w naszym przykładzie prawdziwe). Zatem operator dodawania „zobaczy" po obu stronach dwie jedynki i zmienna `m` uzyska wartość 2.

Często popełniany jest błąd przy dzieleniu: pamiętajmy, że dzielenie liczb całkowitych zawsze, zgodnie z powyższymi zasadami, daje wynik całkowity, a więc części ułamkowej *nie ma*. Początkującym często wydaje się, że jeśli przypisują wynik do zmiennej typu **double**, to w jakiś tajemniczy sposób część ułamkowa zostanie „odzyskana": tak nie będzie, jej tam po prostu nie ma, nie została policzona! Tak więc, jeśli aktualną wartością zmiennej `k` typu **int** jest 7, to wartością wyrażenia '`k/2`' jest *dokładnie* 3. Jeśli chodziło nam raczej o $3\frac{1}{2}$, to wystarczy dopisać kropkę przy dwójce: wartością '`k/2.`' jest 3.5, bo '2.' (z kropką) jest interpretowane jako literał wartości typu **double**, a zatem i wartość `k` zostanie przekształcona do typu **double**, a co za tym idzie i wynik będzie również tego właśnie typu.

Do standardowych niejawnych konwersji, które mogą być wykonane bez naszej wiedzy (i, niestety, zgody...), należy też:

- Przekształcenie dowolnego wskaźnika typu obiektowego do typu **void***. Nie dotyczy to wskaźników do funkcji czy metod — rozdz. 11.12 na stronie 184.
- Przekształcenie wyrażenia stałego o wartości 0 (zero) do wskaźnika **nullptr** dowolnego typu oznaczającego wskaźnik pusty — nie wskazujący na żaden obiekt. Zero jest *jedyną* wartością całkowitą, którą można przekształcić niejawnie na wartość wskaźnikową. Niejawne przekształcenie w drugą stronę — od wskaźnika pustego do liczby zero — nie zachodzi.
- Przekształcenie wartości o typie **T*** do wartości o typie **const T*** i od **T&** do **const T&**, gdzie **T** jest pewnym typem. Niejawne przekształcenie w drugą stronę nie zachodzi.
- Przekształcenie wartości typu liczbowego (całościowego lub zmiennopozycyjnego) lub wskaźnikowego do wartości typu **bool** (zero → **false**, nie-zero → **true**).
- Przekształcenie wartości typu całościowego do innego typu całościowego: jeśli typ docelowy jest **unsigned**, to wartość docelową tworzy się przez przekopiowanie, bez żadnej interpretacji, tylu bitów, ile ma typ docelowy.
- Przekształcenie wartości typu **float** do **double** *i odwrotnie!*
- Przekształcenie wartości typów całościowych w zmiennoprzecinkowe i odwrotnie (te ostatnie z obcięciem części ułamkowej).
- Przekształcenie wskaźnika lub referencji do obiektu klasy pochodnej do wskaźnika lub referencji do obiektu klasy podstawowej.

Brzmi to skomplikowanie, bo też, niestety, jest to skomplikowane. W dodatku jest też niebezpieczne: w C/C++ dozwolone są niejawne konwersje, na skutek których traci się informację (zwykle kompilatory wyświetlają wtedy jakieś ostrzeżenia).

Pamiętać przy tym należy, że niejawne konwersje wcale *nie* gwarantują, że otrzymana po przekształceniu wartość jest z naszego punktu widzenia sensowna, to znaczy zgodna w oczekiwanym przez nas sensie z wartością przed przekształceniem. Na przykład rozpatrzmy program:

P65: ***surp.cpp*** Niejawne konwersje

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int      k   = -2;
7      unsigned uns =  1;
8
9      int      x = k + uns;
10     unsigned y = k + uns;
11
```

```
12      cout << "x   = " << x       << endl;
13      cout << "y   = " << y       << endl;
14      cout << "y+1 = " << y + 1   << endl;
15
16
17      signed   char c = 255;
18      unsigned char d = 255;
19
20      cout << "c+1 = " << c + 1   << endl;
21      cout << "d+1 = " << d + 1   << endl;
22      d = d + 1;
23      cout << "d   = " << (int)d << endl;
24  }
```

Program ten drukuje:

```
x   = -1
y   = 4294967295
y+1 = 0
c+1 = 0
d+1 = 256
d   = 0
```

Wartość zmiennej x to zgodnie z oczekiwaniem -1. Ale drukowanie wartości zmiennej y daje `4294967295` (nie jest to taka sobie przypadkowa liczba; jej wartość to $2^{32} - 1$). Po dodaniu do tej liczby jedynki otrzymujemy dokładnie zero (linia 14 programu drukuje '`y + 1 = 0`'). Wydawałoby się, że zmienne c i d mają te same wartości, więc i po dodaniu jedynki otrzymamy to samo. Ale linia 20 drukuje '`c + 1 = 0`', natomiast linia 21 drukuje '`d + 1 = 256`'. Jednak gdy wartość '`d + 1`' przypiszemy znów do zmiennej d i wydrukujemy jej wartość jako **int** (linia 23), dostaniemy 0. Jak widać, szczególnie łatwo jest pogubić się przy konwersjach od typów ze znakiem do typów bez znaku i odwrotnie — związane jest to z faktem, o którym wspominaliśmy, że zbiory wartości takich typów nie pozostają do siebie w relacji zawierania: nie można powiedzieć, który jest „węższy", a który „szerszy".

Najczęściej jednak nie musimy aż tak dokładnie analizować tego typu wyrażeń, jeśli trzymamy się podstawowych typów i staramy się pisać kod w sposób czytelny i prosty. Trzeba tylko pamiętać, że typy całkowite węższe od **int** zawsze promowane są co najmniej do typu **int**. Dotyczy to w szczególności wartości znakowych (typu **char**). Użyte jako argument operatorów lub argument w wywołaniu funkcji są przekształcane do wartości całkowitej typu **int** równej kodowi ASCII danego znaku. Na przykład po przypisaniu

```
int k = 3 + 'a';
```

wartość k wynosi 100, bo kod ASCII małej litery 'a' jest 97.

Można z tej własności skorzystać do napisania prostej funkcji odczytującej ciąg znaków aż do napotkania nie-cyfry i interpretującej ten napis jako dodatnią liczbę

całowitą:

P66: *konw.cpp* Konwersje znak → liczba

```
#include <iostream>
using namespace std;

int konwert(char* nap) {
    int w = 0, i = 0, c;
    while (c = nap[i++], c >= '0' && c <= '9')
        w = 10*w + c - '0';
    return w;
}

int main() {
    char tab1[] = "123a";
    char tab2[] = "456 1";
    char tab3[] = " 56";

    cout << "tab1 -> " << konwert(tab1) << endl;
    cout << "tab2 -> " << konwert(tab2) << endl;
    cout << "tab3 -> " << konwert(tab3) << endl;
}
```

Wynikiem jest

```
tab1 -> 123
tab2 -> 456
tab3 -> 0
```

Funkcja **konwert** pobiera jako argument tablicę znaków w postaci wskaźnika do pierwszego jej elementu. Następnie, w pętli **while** przetwarzane są kolejne znaki tego napisu. Wewnątrz nawiasów okrągłych, gdzie jest miejsce na wyrażenie logiczne sterujące pętlą, mamy tu wyrażenie przecinkowe. Wyrażenie będące lewym argumentem wczytuje kolejny znak napisu do zmiennej `c`, po czym zwiększa aktualną wartość indeksu. Wartością całego wyrażenia przecinkowego jest wartość prawego argumentu, a tu mamy sprawdzenie, czy wczytany znak jest cyfrą. Co jest bowiem np. wartością wyrażenia `c >= '0'`? Przed dokonaniem porównania obie strony tej nierówności są przkształcane do typu **int**. Zatem wartością zmiennej znakowej `c` będzie kod ASCII wczytanego znaku. Podobnie wartością `'0'` (z apostrofami!) będzie po konwersji do typu **int** kod ASCII *znaku* '0' — jest to liczba 48, ale wcale nie musimy o tym wiedzieć. Podkreślmy jeszcze raz, bo jest to źródłem wielu błędów: wartością liczbową zmiennej znakowej `'0'` jest kod ASCII znaku zero (czyli 48), a *nie* liczba zero. Literałem znaku o kodzie ASCII równym zero jest '\0', łącznie z apostrofami i odwrotnym ukośnikiem.

Kody ASCII kolejnych cyfr, 0, ..., 9, są kolejnymi liczbami całkowitymi (zeru odpowiada 48, jedynce — 49 itd., ale na szczęście nie musimy tych kodów pamiętać). Tak więc warunek `c >= '0' && c <= '9'` sprawdza, czy kod ASCII znaku `c` jest

jednocześnie większy lub równy od kodu znaku '0' i mniejszy lub równy od kodu znaku '9', czyli czy jest to znak odpowiadający cyfrze. Jeśli nie, pętla zostanie przerwana.

Powtórnie wykorzystujemy ten mechanizm w następnej linii: wyrażenie `c-'0'` ma wartość *liczbową* równą liczbie reprezentowanej przez znak `c`. Jeśli np. `c` jest znakiem '4', to `'4'-'0'` jest tym samym co $52 - 48$, czyli 4 (tym razem *liczbowo* cztery). Zatem, jeśli kolejny wczytany znak jest cyfrą, to dotychczasowa wartość zmiennej `w` jest mnożona przez dziesięć, co odpowiada przesunięciu cyfr o jedną pozycję w lewo, i dodawana jest na pozycji jedności wartość liczbowa odpowiadająca wczytanej cyfrze. Pętla kończy się, gdy kolejnym znakiem nie jest cyfra. Dlatego `tab2` zostanie zinterpretowana jako 456; wczytywanie zostanie zakończone, gdy napotkany zostanie odstęp pomiędzy cyfrą 6 a 1. Dla `tab3` otrzymamy 0, bo pętla nie wykona ani jednego obrotu: jako pierwszy znak zostanie bowiem wczytany odstęp, a więc nie-cyfra.

10.2 Porządek wartościowania

Wyrażenia mogą zawierać podwyrażenia, które trzeba obliczyć przed wykonaniem operacji. Na przykład może to być wywołanie funkcji

```
int k = fun1(x) + fun2(y);
```

Aby wykonać dodawanie, trzeba wywołać funkcje, które obliczą wartości argumentów tego dodawania. Ale czy najpierw zostanie wywołana funkcja **fun1** czy **fun2**? Czyli właśnie: jaki jest porządek wartościowania? Otóż w takich przypadkach *nie* jest on określony przez standard języka (tak jak *jest* określony w Javie). Najlepiej więc pisać program tak, aby wynik nie zależał od tego porządku. W przeciwnym przypadku wynik ten może być zaskakujący. Na przykład, co wypisze następujący program?

P67: ***porz.cpp*** Porządek wartościowania

```
#include <iostream>
using namespace std;

int zzz;      // globalne, inicjowane zerem

int fun1() {
    return zzz += 1;
}

int fun100() {
    return zzz += 100;
}

int main() {
    cout << fun1() << " " << fun100() << endl;   ①
}
```

Wydawać by się mogło, że w linii ① najpierw wywołana zostanie funkcja **fun1**. Dodaje ona do wartości globalnej zmiennej zzz jedynkę i zwraca tę podwyższoną wartość. Ponieważ zzz jako zmienna globalna jest zerowana natychmiast po utworzeniu, wywołanie **fun1** powinno zwrócić 1. Natomiast następujące po nim wywołanie funkcji **fun100** dodaje do wartości zzz liczbę 100 i zwraca otrzymaną wartość, która wobec tego powinna wynosić teraz 101. Tymczasem rezultat uruchomienia tego programu to

```
101 100
```

A zatem, najpierw wywołana została **fun100** i jej wynik zapamiętany na jakimś wewnętrznym stosie, zmienna zzz otrzymała wartość 100; następnie wywołana została funkcja **fun1**, która wobec tego zwróciła 101, co zostało wstawione do strumienia, po czym do tego strumienia została wstawiona zapamiętana wartość zwrócona przez **fun100**.

Z podobnych powodów należy unikać odwoływania się w instrukcji dwa razy do zmiennej, która w tej samej instrukcji ulega zmianie; np. zapis

```
tablica[i] = ++i;
```

jest mylący i może być różnie interpretowany: czy po lewej stronie podczas określania adresu pod jaki należy wpisać wynik przypisania wartość indeksu i jest już zwiększona, czy jeszcze nie? (Powinna być już zwiększona, ale lepiej takich konstrukcji nie stosować.)

— 11 —

Funkcje

Rozdział ten poświęcimy funkcjom. Na razie nie będziemy rozpatrywać funkcji będących składowymi klas — obiektem naszego zainteresowania w tym rozdziale będą tylko **funkcja globalna**. Omówimy też pojęcie wskaźników funkcyjnych i na kilku prostych przykładach nauczymy się, jak je definiować i używać.

Podrozdziały:

11.1 Wstęp

Funkcje to opisy czynności do wykonania. Generalnie, z punktu widzenia programu, funkcja jest definicją instrukcji, która na podstawie danych wejściowych (argumentów) dostarcza w miejscu jej użycia (wywołania) wartość określonego typu. Funkcje są więc sposobem na realizację tego samego czy podobnego zadania wielokrotnie, bez potrzeby wielokrotnego powtarzania w naszym kodzie tych samych sekwencji instrukcji.

Definiowanie funkcji jest podobne do definiowania funkcji w Javie i wielu innych językach. Jest jednak szereg ważnych różnic.

W Javie wszystkie funkcje są składowymi jakiejś klasy: albo metodami, albo jej funkcjami statycznymi. W C/C++ natomiast funkcje można definiować poza wszystkimi klasami (w C jest to jedyna możliwość). Zresztą, jak wiemy, w C/C++ można w ogóle nie definiować i nie używać żadnych klas czy struktur. Tak zdefiniowane funkcje nazywamy **funkcjami globalnymi** lub **funkcjami wolnymi**.

Przyzwyczaić się też trzeba do innej cechy języka C/C++ (nieznanej w Javie): użycie funkcji w tekście programu musi być leksykalnie poprzedzone jej definicją lub

przynajmniej deklaracją. Leksykalnie, a więc definicja lub deklaracja funkcji musi wystąpić w *tekście* programu wcześniej niż jej jakiekolwiek użycie. Jak mówiliśmy w rozdz. 2 (str. 7), dyrektywa `#include` włącza plik do tekstu przetwarzanego programu; można więc na przykład definiować czy deklarować funkcję we włączanym pliku nagłówkowym i używać tej funkcji w dalszej części pliku z programem.

11.2 Deklaracje i definicje funkcji

Definicja lub deklaracja dostarcza kompilatorowi informacji o funkcji: jaki jest jej typ zwracany, jaka jest liczba parametrów, jaki jest typ parametrów itd. W ten sposób, za każdym razem, gdy w dalszej części tekstu programu pojawia się użycie tej funkcji, kompilator może sprawdzić

- o którą z funkcji o tej samej nazwie chodzi (może być wiele funkcji o tej samej nazwie);
- czy wywołanie funkcji jest prawidłowe; czy na przykład ilość i typ argumentów odpowiada ilości i typom parametrów, czy typ wartości zwracanej przez funkcje jest w danym miejscu programu akceptowalny, itd. Jest to konieczne, na przykład, aby zdecydować, czy będa potrzebne konwersje argumentów lub wartości zwracanej.

W razie niezgodności kompilacja zostanie przerwana, co jest lepsze niż utworzenie bezsensownego kodu wynikowego. Jest to cecha C++; w tradycyjnym C wymogu wcześniejszego deklarowania funkcji nie było, co powodowało moc kłopotów przy testowaniu i uruchamianiu programów.

Dlaczego definicja *lub* deklaracja, co to w ogóle jest deklaracja funkcji i do czego się przydaje?

Wyobraźmy sobie następującą sytuację: definiujemy kolejno dwie funkcje, **fun1** i **fun2**. Funkcja **fun1** wywołuje w swej treści funkcję **fun2** i odwrotnie, funkcja **fun2** wywołuje w swej treści funkcję **fun1**:

```
1    void fun1(int k) {
2        // ...
3        fun2(k)
4        // ...
5    }
6
7    void fun2(int m) {
8        // ...
9        fun1(m)
10       // ...
11   }
```

W jakiej kolejności zdefiniować te funkcje? Jeśli zdefiniujemy je tak jak wyżej, to w linii 3 kompilacja zostanie przerwana, bo nieznana jest w niej jeszcze funkcja **fun2**;

odwrócenie kolejności nie pomoże, bo wtedy instrukcja wywołania **fun1** wewnątrz **fun2** spowoduje te same kłopoty.

Na szczęście jest wyjście z tej sytuacji. Kompilatorowi nie jest potrzebna definicja funkcji, to znaczy nie musi wiedzieć, co funkcja robi: musi tylko wiedzieć, ile i jakie ma parametry i jaki jest jej typ zwracany. Do tego wystarczy **prototyp** funkcji podany w **deklaracji**. Deklaracja ma formę nagłówka funkcji, po którym następuje średnik zamiast ciała (treści) funkcji. Na przykład poprawnymi deklaracjami funkcji są

```
string& fun1(char* c1, char* c2, bool b);
void    fun2(int k, double d[]);
Klasa*  fun3(Klasa* k1, Klasa* k2);
```

Nie mają one ciała (treści), a więc nie są definicjami. Ale zawierają informacje o nazwie, typie zwracanym, typie i liczbie parametrów (czyli właśnie *prototyp*). Jest to wszystko, czego potrzebuje kompilator, aby sprawdzić formalną prawidłowość ich użycia. Tak więc nasz pierwszy przykład skompiluje się gładko, jeśli przed definicją funkcji **fun1** umieścimy *deklarację* funkcji **fun2** (ze średnikiem na końcu):

```
1   void fun2(int);
2
3   void fun1(int k) {
4       // ...
5       fun2(k)
6       // ...
7   }
8
9   void fun2(int m) {
10      // ...
11      fun1(m)
12      // ...
13  }
```

Zauważmy, że w deklaracji (pierwsza linia) nie podaliśmy nazwy pierwszego i jedynego parametru formalnego funkcji **fun2** — tylko jego typ (**int**). Jest to całkowicie dopuszczalne: do sprawdzenia poprawności wywołania kompilator potrzebuje informacji o liczbie i typie parametrów funkcji, ale ich nazwy nie są do niczego potrzebne i są wobec tego przez kompilator pomijane. Zatem można ich w ogóle nie pisać (choć warto, bo umiejętnie dobrane nazwy są znakomitą formą komentarza). Oczywiście w *definicji* nazwa zwykle jest konieczna, ale nie musi być taka sama jaka została podana w deklaracji.

Na przykład, podane poprzednio trzy deklaracje moglibyśmy równie dobrze zapisać tak:

```
string& fun1(char*, char*, bool);
void    fun2(int, double[]);
Klasa*  fun3(Klasa*, Klasa*);
```

Funkcja zadeklarowana musi oczywiście być gdzieś również zdefiniowana (tylko raz). W przeciwnym razie powstanie błąd na etapie linkowania (łączenia, konsolidacji) programu. Zauważmy, że błędu *nie będzie*, jeśli zadeklarowana funkcja nie została w programie użyta — tak więc wolno deklarować funckje, które dopiero zamierzamy napisać, byle tylko nie próbować ich użycia. Definicja nie musi wystąpić w tym samym module (pliku). Wystarczy, że umieścimy ją w jakimś module składającym się na cały program. W innych modułach, w których funkcja ta jest używana, trzeba tylko zamieścić jej deklarację.

Ponieważ na razie nasze programy i tak mieszczą się w jednym pliku, szczegóły odłożymy do rozdz. 23.1 (str. 505).

Ta sama funkcja może być deklarowana wielokrotnie, nawet w tym samym pliku. Definicja natomiast powinna wystąpić tylko raz (ODR – ang. *One Definition Rule*); wyjątkiem są funkcje rozwijane, o których powiemy dalej.

Oczywiście wszystkie deklaracje, jeśli jest ich kilka, muszą być ze sobą zgodne, czyli definiować ten sam prototyp. Definicja również musi być zgodna z deklaracjami — to znaczy nagłówek funkcji musi być zgodny z prototypem. Jak mówiliśmy, ta zgodność nie musi dotyczyć nazw parametrów formalnych funkcji, które w ogóle nie mają znaczenia w deklaracjach i do prototypu nie należą.

Nagłówek określa prototyp, czyli zewnętrzne własności funkcji. W najprostszej postaci wygląda on tak:

```
Typ Nazwa ListaParam
```

gdzie Typ określa typ wartości zwracanej (przed nią mogą wystąpić modyfikatory, o których powiemy w dalszej części rozdziału), Nazwa oznacza nazwę funkcji, a ListaParam listę parametrów formalnych.

Typ wartości zwracanej.
Musi to być typ wbudowany (jak **int**, **char**, ...), typ zdefiniowany przez użytkownika, typ pochodny (**int***, **Osoba&** itd.), albo wreszcie typ **void**. Jeśli typem zwracanym nie jest **void**, to funkcję nazywamy **funkcją rezultatową**. Jeśli jest to **void**, to funkcja w ogóle nie zwraca żadnej wartości; taką funkcję nazywamy **funkcją bezrezultatową**. Typ wartości zwracanej zwany jest też po prostu **typem funkcji**.
Typem funkcji nie może być typ tablicowy i nie może nim być typ funkcyjny; może to natomiast być typ wskaźnikowy — w szczególności wskaźnik może wskazywać na tablicę lub funkcję — lub typ referencyjny.

Nazwa.
Nazwa może być dowolna, byle nie kolidowała z którymś ze słów kluczowych, składała się tylko z liter, cyfr i znaków podkreślenia. Nie może jednak zaczynać się od cyfry.

Lista parametrów formalnych.

Lista ta jest ujętą w okrągłe nawiasy listą oddzielonych przecinkami deklaracji pojedynczych parametrów funkcji w postaci `(typ1 nazwa1, typ2 nazwa2)`. Nie wolno stosować deklaracji zbiorczych: `(typ nazwa1, nazwa2)` — nie byłoby wtedy wiadomo, czy nazwa2 jest drugim parametrem typu **typ**, czy typem następnego, anonimowego (co jest dopuszczalne), parametru. Nazwy wszystkich parametrów muszą być różne.

Nazwy parametru można w ogóle nie podawać w deklaracjach; w definicjach również można nazwę pominąć, jeśli z argumentu wywołania skojarzonego z tym parametrem funkcja w ogóle nie korzysta, jak to często bywa w czasie tworzenia funkcji w jej wstępnych wersjach.

Lista parametrów może być pusta; obejmujących ją nawiasów pominąć jednak nie można. Jeśli lista parametrów jest pusta, to można zaznaczyć to przez wpisanie wewnątrz nawiasów słowa kluczowego **void**. Nie jest to jednak konieczne.

Pamiętajmy też, że poprzedzenie nazwy typu słowem kluczowym **const** odpowiada zmianie typu: typ **int*** i typ **const int*** to dwa *różne* typy. Jeśli typem parametru jest na przykład **const int***, to kompilator nie zgodzi się na zmianę wartości zmiennej wskazywanej poprzez nazwę wskaźnika przekazanego jako argument (co funkcja normalnie może zrobić; przekazana jej została, co prawda, kopia adresu, ale sam adres odpowiada oryginalnej zmiennej z funkcji wywołującej).

Część nagłówka funkcji składająca się z nazwy funkcji i listy typów jej parametrów (bez nazw tych parametrów) nazywa się czasem jej **sygnaturą**. Typu zwracanego zwykle w sygnaturze nie uwzględnia się. Tak więc na przykład sygnaturą funkcji o prototypie

```
double fun(double x, char* nap);
```

jest

```
fun(double, char*)
```

Definicja funkcji składa się z takiego samego nagłówka, tyle że teraz nie kończymy go średnikiem, tylko umieszczamy zaraz za nim, ujętą w nawiasy klamrowe, treść (ciało) funkcji, czyli sekwencję instrukcji do wykonania. Jeśli w ciele funkcji chcemy korzystać z argumentu przekazanego poprzez parametr funkcji, to oczywiście ten parametr musi mieć nazwę (którą w deklaracji mogliśmy pominąć). W programie może występować tylko jedna definicja funkcji, choć wiele deklaracji. Wyjątkiem są funkcje rozwijane, które mogą być definiowane wielokrotnie (patrz rozdz. 11.10, str. 179).

Ciało funkcji może być traktowane jak wnętrze instrukcji grupującej (złożonej). Do zakresu tej instrukcji grupującej należą również deklaracje zmiennych lokalnych opisane przez specyfikacje parametrów funkcji. Zmienne definiowane w ciele funkcji będą w czasie jej wykonywania lokalne — po wykonaniu funkcji są one usuwane. Zmiennymi *lokalnymi* są również zmienne wyspecyfikowane jako parametry formalne funkcji: będą one zainicjowane wartościami argumentów wywołania. Po zakończeniu wykonywania funkcji zmienne lokalne (z wyjątkiem tych zadeklarowanych jako **static**) będą usunięte.

Zmienne lokalne funkcji (w tym te deklarowane przez specyfikacje parametrów w nagłówku funkcji) w żaden sposób nie kolidują ze zmiennymi o tej samej nazwie w innych funkcjach. Mogą natomiast przesłaniać nazwy zmiennych globalnych, zadeklarowanych poza funkcjami i klasami. Jeśli tak jest, to niekwalifikowana nazwa występująca w ciele funkcji odnosi się zawsze do zmiennej lokalnej, natomiast dostęp do zmiennej globalnej o tej nazwie mamy poprzez operator zasięgu — czterokropek (patrz rozdz. 7.2 na str. 83).

Nie wolno definicji funkcji zagnieżdżać, to znaczy nie można w ciele jednej funkcji definiować innej funkcji (co jest dozwolone w innych językach, jak Pascal czy Fortran 90/95). W nowym standardzie C++11 można jednak wewnątrz funkcji definiować tak zwane funkcje lambda, o których za chwilę.

Definicja (i deklaracja) funkcji może w nowym standardzie C++11 mieć inną, alternatywną, postać, a mianowicie

```
auto f_nazwa(parametry) -> typ_zwracany { ciało }
```

Zamiast specyfikować typ zwracany *przed* nazwą funkcji, stawiamy tam słowo kluczowe **auto**, natomiast typ funkcji określamy zaraz *za* listą parametrów, poprzedzając go „strzałką” ('->'). Nie musimy zresztą tego typu określać jawnie, możemy użyć tu wyrażenia **decltype** (zob. rozdz. 4.1, str. 27). Zauważmy, że jeśli tak zrobimy, to możemy w **decltype** użyć nazw parametrów funkcji, bo w tym miejscu jesteśmy już w zakresie funkcji. Pokażmy to na przykładzie

P68: ***fundefnew.cpp*** Alternatywna postać deklaracji funkcji

```
#include <iostream>
using namespace std;

auto D(int a, double b) -> decltype(a*b);    ①

double A(int a, double b) {                  ②
    return a*b;
}

auto B(int a, double b) -> double {          ③
    return a*b;
}

auto C(int a, double b) -> decltype(a*b) {   ④
    return a*b;
}

double D(int a, double b) {                  ⑤
    return a*b;
}

int main() {
```

```
23      cout << A(4,2.5) << " " << B(4,2.5) << " "
24           << C(4,2.5) << " " << D(4,2.5) << endl;
25  }
```

Definiujemy tu szereg funkcji (**A**, **B**, **C**, **D**), które są właściwie identyczne — wszystkie po prostu zwracają iloczyn swoich argumentów. Definicja funkcji **A** (②) jest „normalna". W linii ③ użyliśmy natomiast nowej składni (**auto** przed nazwą, typ za listą parametrów i poprzedzony strzałką). W definicji funkcji **C** (④) zamiast podawać jawnie typ „poprosiliśmy" kompilator, aby sam określił, jaki powinien być typ iloczynu argumentów (oczywiście będzie to **double**). Widać też, że nowej składni możemy też użyć do deklarowania funkcji (①) — sama definicja (⑤) może być zapisana zarówno w nowej jak i starej składni (wtedy oczywiście typ zwracany musi się zgadzać z tym wydedukowanym przez kompilator z deklaracji).

Forma użyta w tym przykładzie w definicji funkcji **C** (④) okaże się niesłychanie użyteczna przy definiowaniu szablonów funkcji.

Powiedzmy na koniec, że deklaracje i definicje funkcji nie są instrukcjami wykonywalnymi (jak na przykład w Pythonie). Zatem kolejność, w jakiej je piszemy, nie ma znaczenia, dopóki spełniony jest warunek, że co najmniej jedna deklaracja poprzedza leksykalnie instrukcje, w których funkcja jest wykorzystywana.

11.3 Wywołanie funkcji

Wywołanie funkcji ma postać nazwy tej funkcji z podaną w nawiasach okrągłych listą oddzielonych przecinkami argumentów. Oczywiście liczba argumentów wywołania musi być dokładnie taka jak liczba parametrów w deklaracji i definicji funkcji. Nawet jeśli funkcja jest bezargumentowa, nawiasy musimy podać: w przeciwnym przypadku nazwa zostanie potraktowana nie jako wywołanie, ale jak nazwa wskaźnika wskazującej funkcję (więcej o wskaźnikach funkcyjnych powiemy w rozdz. 11.12 na str. 184).

Rolę argumentów mogą pełnić dowolne wyrażenia, których wartość jest typu zgodnego z zadeklarowanym typem odpowiedniego parametru funkcji. Nie musi być to typ identyczny z typem parametru: wywołanie będzie prawidłowe, jeśli wartość argumentu można niejawnie przekształcić na wartość zadeklarowanego typu parametru (patrz rozdz. 10.1). Jeśli przy takiej konwersji może być tracona informacja (czyli jest to konwersja zawężająca, a nie promocja), to zazwyczaj podczas kompilacji otrzymamy ostrzeżenia.

Jeśli funkcja zwraca rezultat, to ostatnią wykonywaną instrukcją musi być *zawsze* instrukcja **return** zwracająca wartość typu, jaki został zadeklarowany jako typ funkcji, albo typu, który może być niejawnie przekształcony do typu funkcji.

Jeśli funkcja jest bezrezultatowa, to instrukcja **return** może (choć nie musi) w niej występować — jeśli występuje, to nie może zawierać żadnego wyrażenia, którego wartość miałaby być zwrócona. Dla funkcji bezrezultatowych domniemywa się instrukcję **return** tuż przed nawiasem kończącym definicję ciała tej funkcji.

Proces wywoływania możemy sobie wyobrażać tak (szczegóły mogą zależeć od platformy i od kompilatora):

- Obliczne są wartości argumentów, po czym dokonywane są konwersje potrzebne do przekształcenia tych wartości w wartości typu odpowiedniego parametru funkcji (co, oczywiście, powinno być wykonalne — w przeciwnym razie program jest błędny).

- Tak uzyskane wartości kopiowane są na stos programu. Zauważmy, że jeśli argumentem była zmienna, to na stos kładziona jest *kopia* jej wartości (po ewentualnej konwersji — wartość kładziona na stosie musi już być *dokładnie* takiego typu jak zadeklarowany typ odpowiadającego temu argumentowi parametru funkcji). Tak więc funkcja *nie* uzyskuje dostępu do zmiennej, która pełniła rolę argumentu, a tylko do kopii jej wartości w momencie wywołania.

- Sterowanie przechodzi do funkcji: kopie wartości argumentów na stosie są utożsamiane ze zmiennymi lokalnymi odpowiadającymi poszczególnym parametrom funkcji. Zmienne lokalne (definiowane w funkcji) umieszczane są również na stosie.
  Wyjątek stanowią zmienne definiowane z modyfikatorem **static**. Takie zmienne tworzone są raz: podczas pierwszego wywołania. Przy kolejnych wywołaniach funkcja ma dostęp do *tej samej* zmiennej statycznej. Istnieje ona aż do zakończenia programu. Różni się od zmiennej globalnej tym, że jej zasięgiem jest zasięg funkcji (od miejsca deklaracji do końca funkcji).

- Po zakończeniu wykonywania funkcji część stosu ze zmiennymi lokalnymi (również tymi odpowiadającymi parametrom funkcji), jest usuwana, tak aby stan stosu odpowiadał temu sprzed wywołania (*zwijanie* stosu). Jeśli funkcja jest rezultatowa, to wynik jest zwracany do funkcji wywołującej (możemy sobie wyobrażać, że również poprzez stos, choć w praktyce stosuje się inne mechanizmy).

Rozpatrzmy przykład ilustrujący konwersje argumentów i rezultatu funkcji:

P69: ***function.cpp*** Konwersje argumentów i wartości zwracanej

```
#include <iostream>
using namespace std;

int d2i(double);
double i2d(int);

int main() {
    double x = 3.5;

    cout << "d2i(x) = " << d2i(x) << endl
         << "i2d(x) = " << i2d(x) << endl;
}

int d2i(double x) { return 3*x; }
double i2d(int k) { return 3*k; }
```

Obie funkcje, **d2i** i **i2d**, zwracają wartość otrzymanego argumentu pomnożoną przez trzy. Wynikiem powinna być liczba 10,5. Kompilator wypisał ostrzeżenia, ale program skompilował się i uruchomił:

```
cpp> g++ -o function function.cpp
function.cpp: In function `int main()':
function.cpp:11: warning: passing `double' for converting 1
                 of `double i2d(int)'
function.cpp: In function `int d2i(double)':
function.cpp:14: warning: converting to `int' from `double'
cpp> ./function
d2i(x) = 10
i2d(x) = 9
cpp>
```

Jak widać, obie funkcje zwracają inne rezultaty, jednak żadnego z nich nie uznalibyśmy za prawidłowy! Dlaczego tak się stało?

Parametr funkcji **d2i** jest typu **double**. Zatem wartość zmiennej `x`, która też jest typu **double**, została przekazana do funkcji jako **double**. W funkcji wartość ta została pomnożona przez trzy; wynikiem jest 10,5. Ale typem funkcji (typem jej wartości zwracanej) jest typ **int**, zatem przed zwróceniem wynik jest przekształcany do tego typu. Część ułamkowa jest więc „obcinana" i zwrócona wartość to liczba całkowita 10 (typu **int**).

W przypadku funkcji **i2d** parametr jest typu **int**. Zatem wartość argumentu, czyli 3,5, zostanie przekształcona do typu **int** i będzie wynosić 3. Po pomnożeniu przez 3 wynik będzie całkowity i równy 9. Funkcja jest typu **double**, więc przed zwróceniem wynik zostanie przekształcony do tego właśnie typu; tym niemniej jego wartość pozostanie 9.

Zauważmy, że w powyższym programie definicje obu funkcji umieściliśmy po funkcji **main**. Na początku programu są one jednak zadeklarowane, a zatem kompilator zna ich prototyp, gdy napotyka ich użycie wewnątrz funkcji **main**.

11.4 Argumenty domyślne

W C/C++ istnieje możliwość definiowania funkcji o domyślnych wartościach argumentów. Znaczy to, że wywołując taką funkcję nie trzeba podawać wszystkich argumentów: dla tych argumentów, których wartość nie została podana, przyjęta zostanie wartość domyślna.

Aby skorzystać z tej możliwości, deklarujemy (lub od razu definiujemy) funkcję, określając wartości domyślne dla parametrów funkcji bezpośrednio w nagłówku. Muszą to być *końcowe* parametry: jeśli któryś z parametrów ma wartość domyślną, to wszystkie parametry występujące na liście parametrów po nim też muszą mieć przypisane wartości domyślne. Wartości domyślne można przypisać tylko raz: jeśli zrobimy to w deklaracji, to *nie* powtarzamy tego w definicji ani innych deklaracjach. Na przykład, można zadeklarować funkcję **fun** o jednym argumencie obowiązkowym i dwoma z wartościami domyślnymi:

```
void fun(int i, int b = 0, double x = 3.14);
```

Ponieważ w deklaracji nazwy parametrów są i tak pomijane, tę samą deklarację możemy zapisać tak:

```
void fun(int, int = 0, double = 3.14);
```

W dalszej części programu musimy oczywiście podać definicję tej funkcji; tam już przypisania wartości domyślnych *nie* umieszczamy:

```
void fun(int i, int b, double x) {
    // ...
}
```

Teraz możemy wywoływać funkcję **fun** na różne sposoby, np.:

```
fun(3, 4, 7.5);
fun(3, 4);
fun(3);
```

W linii pierwszej podaliśmy wszystkie argumenty, więc żadna wartość domyślna nie zostanie użyta. W drugiej linii podaliśmy dwa argumenty: wartość pierwszego zostanie użyta do zainicjowania parametru pierwszego (i), wartość drugiego — parametru drugiego (b); tu wartość domyślna 0 zostanie zignorowana). Trzeciego argumentu nie podaliśmy, więc parametr trzeci (x) zostanie zainicjowany wartością domyślną (3,14). W trzeciej linii podaliśmy tylko jeden argument; dla dwóch pozostałych przyjęte zostaną wartości domyślne. Tak więc powyższe trzy wywołania równoważne są wywołaniom:

```
fun(3, 4, 7.5);
fun(3, 4, 3.14);
fun(3, 0, 3.14);
```

Zauważmy, że nie ma sposobu, aby podać wartość trzeciego argumentu nie podając drugiego, gdyż kolejne argumenty przypisywane są zawsze do kolejnych parametrów i, jeśli lista parametrów nie została wyczerpana, dla pozostałych użyte będą wartości domyślne. Nie da się również wywołać funkcji nie podając argumentów odpowiadających parametrom, dla których wartości domyślne nie zostały określone. W naszym przypadku na przykład, nie byłoby możliwe wywołanie funkcji **fun** bez żadnych argumentów, ponieważ argument odpowiadający pierwszemu parametrowi jest obowiązkowy.

Argument domyślny może być wyrażeniem, którego wartość zostanie obliczona przy wywoływaniu funkcji (takim wyrażeniem może być na przykład wyrażenie zawierające nazwę zmiennej globalnej lub wywołanie funkcji). Jeśli tak jest, to wartość tego wyrażenia znajdowana jest za każdym razem gdy funkcja jest wywoływana, ale w zakresie tej deklaracji, gdzie wartości domyślne zostały zdefiniowane. Na przykład w poniższym programie w zakresie, do którego należy deklaracja funkcji **kolo** (①), widoczną zmienną PI jest globalna zmienna o tej nazwie.

P70: ***wardom.cpp*** Wartości domyślne funkcji

```
#include <iostream>
using namespace std;

double PI = 3;

double kolo(double, double = PI);                    ①

int main() {
    double r = 1;

    cout << "1. PI = " << PI << " Pow = "
                       << kolo(r) << endl;           ②
    double PI = 3.14;
    cout << "2. PI = " << PI << " Pow = "
                       << kolo(r) << endl;           ③
    ::PI = 3.14;                                     ④
    cout << "3. PI = " << PI << " Pow = "
                       << kolo(r) << endl;
}

double kolo(double r, double pi) {
    return pi*r*r;
}
```

W pierwszym wywołaniu funkcji (linia ②) użyta zostanie wartość 3. W drugim wywołaniu (③), lokalna zmienna PI przysłania w funkcji **main** zmienną globalną o tej samej nazwie, ale wartość domyślna drugiego argumentu funkcji **fun** zostanie obliczona w zakresie globalnym, a zatem w dalszym ciągu będzie to wartość 3! Natomiast jeśli zmienimy wartość zmiennej globalnej (④), to ta nowa wartość zostanie użyta przy inicjowaniu parametrów domyślnych dla trzeciego wywołania, o czym przekonuje nas wydruk:

```
1. PI = 3 Pow = 3
2. PI = 3.14 Pow = 3
3. PI = 3.14 Pow = 3.14
```

Parametry funkcji zadeklarowane jako obowiązkowe można przedeklarować tak, aby miały wartości domyślne. Należy tylko pamiętać, aby każdy parametr deklarowany jako domyślny był w ten sposób deklarowany *tylko raz*. Zmienić status parametrów z obowiązkowego na parametr z wartością domyślną można tylko dla ostatnich obowiązkowych parametrów na liście.

Załóżmy, że w pliku nagłówkowym ***wardom1h.h*** zadeklarowana jest funkcja z czterema parametrami, w tym tylko ostatni ma wartość domyślną (równą 255):

```
void kolor(int r, int g, int b, int alpha = 255);
```

Rozpatrzmy teraz program

P71: ***wardom1.cpp*** Przedeklarowywanie wartości domyślnych

```
1 #include <iostream>
2 using namespace std;
3
4 #include "wardom1h.h"
5
6 void kolor(int, int , int = 0, int);          ①
7
8 int main() {
9     kolor(100,150,250,199);
10    kolor(100,150,250);
11    kolor(100,150);
12 }
13
14 void kolor(int red, int green, int blue, int alpha) {
15    cout << "Alpha = " << alpha << " (R,G,B) = (" << red
16         << "," << green << "," << blue << ")" << endl;
17 }
```

Na początku włączamy plik nagłówkowy ***wardom1h.h***. W zasięgu znajdzie się zatem deklaracja funkcji **kolor** z jednym (czwartym) argumentem domyślnym. Ostatnim na liście parametrem obowiązkowym jest parametr trzeci. Możemy więc zmienić jego status — robimy to przedeklarowując funkcję w linii ①. Zauważmy, że *nie* piszemy wartości domyślnej dla czwartego parametru: ten parametr *już jest* domyślny (ma wartość domyślną 255) i drugi raz przypisywać mu wartości domyślnej, nawet tej samej, nie wolno. Redeklaracja zmienia zatem status trzeciego parametru na domyślny i możliwe jest teraz wywołanie funkcji z czterema, trzema i dwoma argumentami; otrzymujemy

```
Alpha = 199 (R,G,B) = (100,150,250)
Alpha = 255 (R,G,B) = (100,150,250)
Alpha = 255 (R,G,B) = (100,150,0)
```

„Sztuczek" z przedeklarowywaniem funkcji i zmienianiem statusu ich parametrów lepiej jednak unikać, gdyż bardzo skutecznie zaciemniają kod.

Argumenty domyślne stosuje się często w konstruktorach klas.

11.5 Zmienna liczba argumentów

Zdarzają się funkcje, które nie mają w naturalny sposób określonej, z góry znanej liczby argumentów. Jako przykład możemy wyobrazić sobie funkcję wyznaczająca maksimum z dwóch, trzech, czterech... liczb, albo drukującą wartości pewnej nieustalonej z góry ilości argumentów. W C/C++ istnieje sposób na tworzenie tego

rodzaju funkcji, choć jest to nieco zawiłe i trudne w użyciu: można zastosować **funkcje o zmiennej liczbie argumentów** (ang. *variable-length argument list*). W C++ zwykle lepiej i prościej zastosować opisany dalej mechanizm przeciążania funkcji lub narzędzia z nowego standardu (np. tak zwane krotki lub listy inicjujące).

Funkcje tego typu deklarujemy ze znakiem wielokropka ('`...`') w miejsce parametrów, których liczby nie specyfikujemy. Przed wielokropkiem wymienione są wszystkie „normalne", obowiązkowe parametry. Na przykład deklaracja

```
int fun(char* ...);
```

deklaruje funkcję, którą można wywołać z obowiązkowym pierwszym argumentem (typu **char***) i dowolną liczbą dodatkowych argumentów. Tę samą deklarację można też zapisać z przecinkiem po ostatnim obowiązkowym argumencie:

```
int fun(char*, ...);
```

Jak napisać treść funkcji aby prawidłowo odczytać w niej wszystkie przekazane argumenty? Przede wszystkim należy dołączyć plik nagłówkowy ***cstdarg*** a następnie, wewnątrz funkcji:

1. Utworzyć zmienną typu **va_list**. Typ ten jest dostępny po dołączeniu pliku nagłówkowego ***cstdarg*** (nazwa **va_list** pochodzi od *variable-arguments list*). Załóżmy, że zmienna ta nazywa się `ap` (jest to tradycyjna nazwa takiej zmiennej, od *argument pointer*).

2. Wywołać (tylko raz) funkcję **va_start** podając jako jej pierwszy argument utworzoną wcześniej zmienną typu **va_list**, a jako drugi argument zmienną odpowiadającą *ostatniemu* parametrowi obowiązkowemu funkcji; w poniższym przykładzie wywołanie ma zatem postać `va_start(ap,typ)`.

3. Wartości kolejnych argumentów odczytywać wywołując funkcję **va_arg**: jej pierwszym argumentem powinna być zmienna `ap`, a drugim nazwa typu tego argumentu funkcji który chcemy odczytać. Wynika z tego zatem, że ten typ trzeba znać! Trzeba też wiedzieć, kiedy skończyć wczytywanie argumentów. Najczęściej informacja o liczbie i typie argumentów w aktualnym wywołaniu jest w jakiś sposób przekazywana poprzez pierwsze, obowiązkowe argumenty funkcji.

4. Po zakończeniu wczytywania argumentów wywołać funkcję **va_end** ze zmienną `ap` jako jedynym argumentem. Funkcja ta porządkuje stos; jeśli jej nie wywołamy, to program może się załamać.

Jako przykład rozpatrzmy program:

P72: ***varg.cpp*** Funkcje o zmiennej liczbie argumentów

```
#include <iostream>
#include <cstdarg>
using namespace std;

void typy(const char typ[] ...);

int main() {
    typy("SxS", "Jan", 0, "Maria");
    typy("issD", 17, "Jan", "Maria", 1.);
    typy("iDdsiI", 17, 19.5, 1.5, "OK", -1, 8);
}

void typy(const char typ[] ...) {
    int     i = 0, integ;
    char    c, *strin;
    double  doubl;

    va_list ap;

    va_start(ap,typ);

    while ( (c = typ[i++]) != '\0') {            ①
        switch (c) {
            case 'i':
            case 'I':
                integ = va_arg(ap,int);
                cout << "Liczba int    : " << integ << endl;
                break;
            case 'd':
            case 'D':
                doubl = va_arg(ap,double);
                cout << "Liczba double: " << doubl << endl;
                break;
            case 's':
            case 'S':
                strin = va_arg(ap,char*);
                cout << "Napis         : " << strin << endl;
                break;
            default:
                cout << "Nielegalny kod typu!!!!!" << endl;
                goto KONIEC;
        }
    }
    KONIEC:
```

```
45      cout << endl;
46
47      va_end(ap);                                          ②
48  }
```

Pierwszym, obowiązkowym argumentem funkcji **typy** jest napis, czyli tablica znaków, z których ostatni jest znakiem '\0'. Kolejne znaki tego napisu określają typy kolejnych argumentów wywoływanej funkcji: 'd' lub 'D' — typ **double**, 'i' lub 'I' — typ **int**, 's' lub 'S' — typ **char***, czyli wskaźnik do napisu. W ciele funkcji, po wykonaniu kroków 1 i 2 z podanego powyżej schematu postępowania, odczytujemy w pętli **while** kolejne argumenty. Najpierw (①) z napisu typ odczytujemy kolejny znak (aż będzie nim znak '\0'). Znak ten określa typ kolejnego argumentu do wczytania. Znając ten typ, za pomocą instrukcji **switch** przechodzimy do wczytywania kolejnego argumentu: wczytujemy go do zmiennej roboczej o odpowiednim typie za pomocą wywołania funkcji **va_arg** (patrz punkt 3 schematu). Jeśli odczytany znak nie odpowiada żadnemu ze spodziewanych typów, wypisywany jest komunikat i za pomocą instrukcji **goto** przerywane jest wykonywanie zarówno bloku **switch**, jak i pętli **while**. Tym niemniej funkcja **va_end** musi nawet wtedy być wywołana (②), aby umożliwić „posprzątanie" stosu i kontynuowanie programu. Przykład działania tego programu podany jest poniżej:

```
Napis        : Jan
Nielegalny kod typu!!!

Liczba int   : 17
Napis        : Jan
Napis        : Maria
Liczba double: 1

Liczba int   : 17
Liczba double: 19.5
Liczba double: 1.5
Napis        : OK
Liczba int   : -1
Liczba int   : 8
```

Przy pierwszym wołaniu powstaje błąd, gdyż użyta jest w napisie typ litera 'x', która nie odpowiada żadnemu typowi. Funkcja kończy swoje działanie, ale porządkuje stos i dalszy przebieg programu może być prawidłowy.

Ponieważ w deklaracji i definicji funkcji nie jest określony typ parametrów, wyłączona zostaje kontrola zgodności typów dla jej wywołań. Aby uprościć stosowanie funkcji o zmiennej liczbie parametrów, „krótkie" wartości całkowite awansowane są do typu **int**, a wartości **float** do typu **double**. Na przykład w drugim wywołaniu funkcji **typy** jednym z argumentówr jest '1.'. Gdybyśmy opuścili kropkę dziesiętną, literał odpowiadałby wartości całkowitej 1 i zostałby przekazany jako czterobajtowa wartość typu **int**. Ta zostałaby następnie odczytana jako **double**, a więc wartość ośmiobajtowa, co doprowadziłoby do trudno wykrywalnego błędu w programie.

Stosować funkcje o zmiennej liczbie argumentów należy tylko wtedy, gdy naprawdę są potrzebne i robić to trzeba bardzo ostrożnie.

W nowym standardzie C++11 istnieją inne, lepsze metody do przekazywania nieokreślonej z góry liczby danych do funkcji, o których powiemy w dalszej części.

11.6 Argumenty referencyjne

Zarówno w roli argumentów, jak i wartości zwracanej mogą występować referencje, o których mówiliśmy w rozdz. 4.7 na stronie 47. Co to znaczy, jeśli parametr funkcji jest zadeklarowany jako referencja?

Niech funkcja będzie zdefiniowana tak:

```
void fun(int& k) {
    k = k + 2;
}
```

Taki zapis oznacza, że argument typu **int** zostanie przesłany do funkcji przez referencję (referencję). Jeśli wywołamy tę funkcję podając jako argument pewną zmienną typu **int**, to w czasie wykonania funkcji nazwa parametru k będzie *inną nazwą* dokładnie *tej samej* zmiennej. A więc nie będzie kopiowania wartości na stos: zmienna lokalna w funkcji w ogóle nie będzie tworzona, nazwa k będzie odnosić się do *oryginału* zmiennej będącej argumentem wywołania. Jeśli powyższą funkcję wywołamy tak:

```
int m = 1;
fun(m);
cout << "m = " << m << endl;
```

to wewnątrz funkcji nazwa k będzie inną nazwą tej zmiennej, która w programie wywołującym nazywa się m. Ta właśnie zmienna zostanie przez funkcję powiększona o dwa. Zatem po wykonaniu funkcji zmienna m w funkcji wywołującej będzie zmieniona — jej wartość wynosić będzie teraz trzy!

Zauważmy, że samo wywołanie funkcji wygląda tak jak normalne wywołanie przez wartość. Przy wywołaniu przez wartość jednak to kopia wartości m byłaby przypisana do *lokalnej* dla funkcji zmiennej k. Modyfikacje tej lokalnej zmiennej nie odbiłyby się w żaden sposób na wartości oryginału, czyli zmiennej m z funkcji wywołującej.

Fakt, że patrząc na wywołanie funkcji nie jesteśmy w stanie stwierdzić, czy jest to wywołanie przez wartość czy przez referencję, jest dość nieszczęśliwy: może prowadzić do błędnej interpretacji kodu, jeśli nie sprawdzi się deklaracji lub definicji funkcji. Dlatego nie należy stosować takiej formy wywołania bez potrzeby. Z drugiej strony, w niektórych sytuacjach jest to sposób najwygodniejszy, a czasem wręcz konieczny.

Parametr zadeklarowany jako referencyjny jest inną nazwą zmiennej przekazanej jako argument. Zatem ta zmienna musi istnieć — jak podkreślaliśmy, w funkcji *nie jest* tworzona zmienna lokalna.

Co jednak będzie, jeśli jako argumentu użyjemy literału lub wyrażenia o określonej, co prawda, wartości, ale nie będącego l-wartością istniejącej zmiennej? Albo, co

będzie, jeśli przekażemy jako argument l-wartość istniejącej zmiennej, ale innego typu niż ten zadeklarowany na liście parametrów funkcji? Zauważmy bowiem, że nie może tu być mowy o niejawnych konwersjach: identyfikator zadeklarowany jako referencja skojarzona ze zmienną typu **double** nie może przecież być „inną nazwą" zmiennej typu **int**. Doprowadziłoby to szybko do kompletnego chaosu.

Zatem w obu przypadkach kompilator wykryje błąd. Jest jednak odstępstwo od tej reguły. Jeśli parametr referencyjny jest zadeklarowany z modyfikatorem **const**, takie wywołanie jest dopuszczalne. Ponieważ jednak zmienna lokalna nie jest dla parametrów referencyjnych tworzona, zostanie utworzona zmienna tymczasowa odpowiedniego typu i nazwa parametru w funkcji będzie inną nazwą tej zmiennej tymczasowej. Zauważmy, że to *nie* doprowadzi do chaosu: skoro parametr był **const**, czyli *'read-only'*, to i tak nie była możliwa zmiana wartości argumentu, zatem żadnych nieprzewidzianych skutków ubocznych taka konstrukcja nie powinna spowodować.

Parametry referencyjne to wygodny sposób przekazywania do funkcji obiektów o znacznych rozmiarach. Oczywiście zmienne typów wbudowanych są małe, ale zmienne typów (klas) definiowanych przez autora programu mogą być i bywają bardzo duże. Przekazywanie przez wartość powodowałoby konieczność kopiowania ich na stos, tworzenia zmiennych lokalnych itd. Użycie referencji eliminuje te niedogodności. Z drugiej strony, często nie jest naszą intencją, aby funkcja zmodyfikowała zmienną przesłaną jej przez referencję jako argument. Pamiętamy bowiem, że funkcja ma wtedy dostęp do oryginału, a nie do kopii wartości. Przed taką niezamierzoną zmianą możemy się uchronić deklarując parametr referencyjny z modyfikatorem **const**; zmienna będąca argumentem wywołania nie musi wtedy wcale być ustalona — skojarzenie argumentu typu **Typ** z parametrem typu **const Typ&** jest dopuszczalne (jest to prawidłowa, trywialna konwersja niejawna: patrz rozdz. 10.1 na stronie 147).

Inna sytuacja, gdy argumenty referencyjne przydają się, zachodzi wtedy, gdy właśnie zależy nam na tym, aby zmienna przekazywana do funkcji zmieniła swoją wartość. Na przykład chcielibyśmy, aby funkcja obliczyła i zwróciła jakieś dwie (albo więcej) wartości: poprzez normalny mechanizm zwracania wartości za pomocą **return** możemy otrzymać jedną z nich, ale jak dostać drugą? Można to łatwo rozwiązać dzięki parametrom referencyjnym. Oczywiście, inną możliwością jest użycie wskaźników: posyłamy do funkcji poprzez argument *adres* zmiennej, którą chcemy w funkcji zmienić. Ten adres zostanie przesłany przez wartość, a więc przesłana zostanie jego kopia, ale sam adres będzie wskazywał na istniejącą zmienną z funkcji wywołującej, a więc funkcja wołana będzie miała do niej dostęp. Zatem będzie mogła wartość tej zmiennej zmodyfikować.

Różne sposoby przesyłania argumentów do funkcji ilustruje poniższy program:

P73: ***vrp.cpp*** Przekazywanie argumentu przez wartość, referencję i wskaźnik

```
1 #include <iostream>
2 using namespace std;
3
4 void fun(int, int&, int*);
5
6 int main() {
```

```
7     int a = 1, b = 2, c = 3;
8
9     cout << "Przed: a = " << a << " b = " << b
10                          << " c = " << c << endl;
11    fun(a, b, &c);
12
13    cout << "Po   : a = " << a << " b = " << b
14                          << " c = " << c << endl;
15 }
16
17 void fun(int x, int& y, int* z) {
18    x = 2*x;
19    y = 3*y;
20   *z = 4*(*z);
21 }
```

Pierwszy argument przesłany jest przez wartość, a więc zmiana wartości zmiennej *lokalnej* x wewnątrz funkcji nie odbije się na wartości zmiennej a w funkcji **main**. Drugi argument przesyłany jest przez odniesienie (referencję), a więc w funkcji identyfikator y oznacza dokładnie tę samą zmienną, która w funkcji **main** nazywa się b. Ponieważ funkcja modyfikuje wartość y, więc automatycznie zmieniona jest i wartość zmiennej b w funkcji **main**. Trzeci argument przekazywany jest przez wskaźnik. Odpowiedni parametr funkcji **fun** zadeklarowany jest jako typu **int***. Jest to wskaźnik do **int**, a więc funkcja spodziewa się otrzymania (przez wartość) *adresu* zmiennej typu **int**. Dlatego trzecim argumentem wywołania jest *adres* zmiennej c (czyli wartość wyrażenia &c). Wewnątrz funkcji wyrażenie *z oznacza zmienną wskazywaną przez z, ale z zawiera przekazany jej adres zmiennej c z programu głównego; zatem *z w funkcji jest tym samym co c w programie głównym — w ten sposób modyfikacja *z modyfikuje oryginał, czyli zmienną c w programie głównym:

```
Przed: a = 1 b = 2 c = 3
Po   : a = 1 b = 6 c = 12
```

Jeśli nazwa parametru typu referencyjnego oznacza w funkcji tę samą zmienną, która została użyta jako odpowiedni argument podczas jej wywołania, to co będzie, jeśli parametrem tym jest referencja do tablicy? Czy wewnątrz funkcji znany na przykład będzie wymiar tej tablicy, tak jak jest znany w tej funkcji, gdzie tablica była deklarowana? Odpowiedź jest twierdząca. Spójrzmy na następujący przykład:

P74: ***tabref.cpp*** Referencja do tablicy

```
1 #include <iostream>
2 using namespace std;
3
4 void funtab(double[]);                    ①
5 void funref(double (&)[6]);               ②
6
```

```
7  int main() {
8      double tab[] = {1,2,3,4,5,6};
9      cout << "Wymiar double : " << sizeof(double)   << endl;
10     cout << "Wymiar double*: " << sizeof(double*)  << endl;
11     cout << "Wymiar tab w main: " << sizeof(tab)   << endl;
12     funtab(tab);
13     funref(tab);                                   ③
14 }
15
16 void funtab(double t[]) {
17     cout << "Wymiar t w funtab: " << sizeof(t) << endl;
18 }
19
20 void funref(double (&t)[6]) {                     ④
21     cout << "Wymiar t w funref: " << sizeof(t) << endl;
22 }
```

Parametrem funkcji **funtab** zadeklarowanej w linii ① jest tablica, czyli tak naprawdę wskaźnik do początku tej tablicy. Wewnątrz funkcji wymiar tablicy jest nieznany, zmienna `t` jest typu **double*** i jej rozmiar wynosi 4 (lub 8). Aby wymiar stał się znany, musielibyśmy przesłać go jako osobny argument. Parametrem funkcji **funref** (zadeklarowanej w linii ②) jest *referencja* do tablicy. Zauważmy tu deklarację: '`double (&)[6]`' jest tu nazwą typu *referencja do sześcioelementowej tablicy elementów typu* **double**. Nawias jest tu konieczny; gdyby go nie było, to typem byłaby *sześcioelementowa tablica odniesień (referencji) do zmiennych typu* **double** — coś takiego w ogóle nie istnieje!

Nie istnieje typ *tablica odniesień.*

Podobnie w linii ④ '`double (&t)[6]`' oznacza `t` *jest referencją do sześcioelementowej tablicy elementów typu* **double**, czyli inaczej: *identyfikator* `t` *jest inną nazwą sześcioelementowej tablicy elementów typu* **double**, *która istnieje w funkcji wołającej i została tam użyta jako argument wywołania.* Zauważmy, że wymiar *musi* być podany: tablica czteroelementowa to nie to samo, co tablica pięcioelementowa. Dlatego w tym przypadku funkcja **tabref** *zna* wymiar tablicy:

```
Wymiar double : 8
Wymiar double*: 8
Wymiar tab w main: 48
Wymiar t w funtab: 8
Wymiar t w funref: 48
```

Nie oznacza to jednak, że rozwiązaliśmy problem przekazywania rozmiaru tablicy do funkcji „żeby było tak jak w Javie": zauważmy bowiem, że argumentem wywołania funkcji **funref** może być *wyłącznie* tablica sześcioelementowa! Spróbujmy zmienić wymiar tablicy `tab` przez dodanie na przykład jednego elementu: funkcji **funtab** to nie przeszkadza, ale wywołanie z linii ③ w ogóle się nie skompiluje!

11.7 Referencje jako wartości zwracane funkcji

Odniesienie (referencja) może nie tylko być parametrem, ale i wartością zwracaną przez funkcję. Następująca deklaracja

```
int& fun(int);
```

oznacza, że funkcja zwraca pewną zmienną typu **int** przez referencję, czyli na przykład wyrażenie `fun(3)` jest inną nazwą pewnej istniejącej zmiennej, która została podana w funkcji w instrukcji **return**.

> Funkcja nie może zwracać przez referencję (ani przez wskaźnik) swoich zmiennych lokalnych, gdyż po powrocie do funkcji wywołującej zmienne lokalne funkcji już nie istnieją. Może natomiast zwrócić zmienną lokalną przez wartość: po powrocie z funkcji zmienna lokalna nie istnieje, ale kopia jej wartości, po ewentualnej konwersji do typu zadeklarowanego jako typ zwracany, przekazana zostaje do funkcji wywołującej (poprzez stos, rejestr procesora lub jeszcze inaczej; to już jest szczegół implementacyjny, którym nie musimy się zajmować).

To, że wyrażenie z wywołaniem funkcji jest traktowane jako nazwa istniejącej zmiennej, a więc jako l-wartość, oznacza, że wywołanie funkcji zwracającej referencję może pojawić się po lewej stronie przypisania, co wygląda trochę szokująco i nie stosuje się zbyt często; tym niemniej nie ma tu błędu. Przykładem może być funkcja **funmax** z poniższego programu:

P75: *varepo.cpp* Różne typy argumentów i wartości zwracanych

```
#include <iostream>
#include <cmath>
using namespace std;

double potegi(double&,double*);
int*   kwadrat(int*);
int&   funmax(int[],int);

int main() {

      // argumenty wskaźnikowe i referencyjne
    double u = 4, v;
    double szesc = potegi(u, &v);                    ①
    cout << "Szescian: " << szesc  << "; kwadrat: "
         << u << "; pierwiastek: " << v  << endl;

      // to też ma sens
    int i = 4;
```

```
19      cout << "20? : " << ++*kwadrat(&i)+3 << endl;          ②
20
21      // funkcja zwracająca referencję
22      int tab[] = {1,4,6,2};
23      cout << "Tablica przed: ";
24      for ( i = 0; i < 4; i++ ) cout << tab[i] << " ";
25      cout << endl;
26
27      funmax(tab,4) = 0;                                     ③
28
29      cout << "Tablica po   : ";
30      for ( i = 0; i < 4; i++ ) cout << tab[i] << " ";
31      cout << endl;
32  }
33
34  double potegi(double& u, double* v) {                      ④
35      double x = u;
36      u *= u;
37      *v = sqrt(x);
38      return u*x;
39  }
40
41  int* kwadrat(int* p) {                                     ⑤
42      *p *= *p;
43      return p;
44  }
45
46  int& funmax(int* tab, int ile) {                           ⑥
47      int i, ind = 0;
48      for ( i = 1; i < ile; i++ )
49          if ( tab[i] > tab[ind] ) ind = i;
50      return tab[ind];
51  }
```

Funkcja **potegi** ma jeden argument referencyjny i jeden wskaźnikowy. Przez referencję, jako pierwszy argument, przesyłamy liczbę dodatnią: funkcja oblicza jej kwadrat, sześcian i pierwiatek kwadratowy (④). Kwadrat argumentu jest w funkcji przypisywany do tej samej zmiennej u, która była pierwszym argumentem tej funkcji. Ponieważ argument jest referencyjny, w funkcji **main** po wywołaniu funkcji **potegi** wartość ta będzie nową wartością zmiennej u — stara wartość, wynosząca 4, zostanie zamazana.

Jako drugi argument wysyłamy do funkcji kopię adresu zmiennej v (①). W funkcji do zmiennej o tym adresie wpisywana jest wartość pierwiastka kwadratowego pierwszego argumentu. Tak więc po powrocie z funkcji zmienna v w programie głównym będzie miała wartość 2. Funkcja **potegi** zwraca, przez wartość, sześcian argumentu.

Po wywołaniu funkcji **potegi** sześcian pierwotnej wartości zmiennej u mamy w zmiennej szesc, kwadrat jest nową wartością zmiennej u, a pierwiastek jest wartością zmien-

nej v: potwierdza to wydruk z programu

```
Szescian: 64; kwadrat: 16; pierwiastek: 2
20? : 20
Tablica przed: 1 4 6 2
Tablica po   : 1 4 0 2
```

Dość karkołomna konstrukcja użyta jest w linii ②. Funkcja **kwadrat** (⑤) oblicza kwadrat argumentu przekazanego przez wskaźnik (do funkcji przekazujemy kopię *adresu* zmiennej i). Obliczona wartość jest wpisywana do zmiennej wskazywanej przez wskaźnik, a więc do zmiennej i z programu głównego. Jednocześnie wskaźnik do tej samej zmiennej jest zwracany w instrukcji **return** — zauważmy, że nie jest to wskaźnik do zmiennej lokalnej, ale do istniejącej w programie głównym zmiennej i. Po wywołaniu funkcji wartością wyrażenia `kwadrat(&i)` jest zatem wskaźnik do zmiennej i, której wartość uległa zmianie i wynosi teraz 16 ($= 4 \times 4$). Za pomocą operatora gwiazdki dokonujemy dereferencji, a więc wyrażenie `*kwadrat(&i)` jest równoważne zmiennej `i`, o wartości 16. Tę wartość zwiększamy o jeden za pomocą operatora zwiększenia i do wyniku dodajemy 3; zatem wartością całego wyrażenia `++*kwadrat(&i)+3` jest 20. Wbrew pozorom takie zawiłe konstrukcje dość często zdarzają się w realnych programach (choć może nie powinny).

Przyjrzyjmy się teraz funkcji **funmax** (⑥). Do funkcji w tradycyjny sposób wysyłamy tablicę i jej wymiar (③). Funkcja szuka największego elementu i zwraca tenże element *przez referencję*, czego z linii zawierającej instrukcję **return** nie widać, ale widać z nagłówka w deklaracji/definicji. Zatem wyrażenie `funmax(tab,4)` jest nazwą tej zmiennej, która jest największym elementem tablicy, czyli w naszym przykładzie tab[2]. Wyrażenie to stoi po lewej stronie przypisania, zatem zmienna ta jest zerowana, o czym przekonuje nas wydruk.

11.8 Funkcje rekurencyjne

Funkcje w C/C++ mogą wywoływać same siebie — takie funkcje nazywamy **funkcjami rekurencyjnymi**. Przy definiowaniu takich funkcji trzeba oczywiście zadbać, aby ciąg wywołań skończył się. Zatem przed wywołaniem samej siebie funkcja zwykle sprawdza pewien warunek i jeśli nie jest on spełniony, nie dokonuje już samowywołania.

Jako przykład rozpatrzmy funkcję **gcd** z poniższego programu. Oblicza ona i zwraca największy wspólny dzielnik (ang. *greatest common divisor*) swoich dwóch argumentów, które są liczbami naturalnymi (całkowitymi dodatnimi). Użyty algorytm to znana wszystkim modyfikacja algorytmu Euklidesa opisanego w *Elementach* jako Twierdzenie VII.2 (i będącego wypadkiem szczególnym wcześniejszego i bardziej ogólnego algorytmu, przypisywanemu Teajtetosowi):

P76: *gcd.cpp* Algorytm Euklidesa - realizacja rekurencyjna

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

```
3
4  int gcd(int a, int b) {
5      return b == 0 ? a : gcd(b, a % b);
6  }
7
8  int main() {
9      int x, y;
10
11     x = 732591; y = 1272585;
12     cout << "gcd(" << x << "," << y << ")\t= "
13          << gcd(x,y) << endl;
14
15     x = 732591; y = 1270;
16     cout << "gcd(" << x << "," << y << ")\t= "
17          << gcd(x,y) << endl;
18 }
```

Sama funkcja realizująca algorytm składa się w zasadzie z jednej instrukcji **return**. Dopóki zmienna lokalna b nie stanie się równa zeru, funkcja wywołuje się rekurencyjnie. Z analizy teoretycznej wynika, że dla prawidłowych danych (a więc liczb naturalnych, czyli, w szczególności, dodatnich) po skończonej liczbie kroków wartość b musi stać się równa zeru. Tak więc warunek zakończenia (tzw. warunek stopu) jest zapewniony i funkcja działa prawidłowo:

```
gcd(732591,1272585)     = 129
gcd(732591,1270)        = 1
```

„Samowywołanie" się funkcji jest w zasadzie normalnym wywołaniem: wszystkie lokalne zmienne są tworzone oddzielnie w każdym wywołaniu, mechanizm przekazywania argumentów jest taki sam jak dla funkcji nierekursywnych. Na przykład funkcja **gcd** kreuje zmienne lokalne a i b a następnie, obliczając wyrażenie za dwukropkiem, wywołuje tę samą funkcję **gcd** i czeka na zwrócenie wyniku. To następne „wcielenie" funkcji też tworzy swoje zmienne lokalne a i b, woła **gcd** i czeka na wynik, aby go zwrócić, itd. Kiedy w końcu b stanie się zero, funkcja zwróci a, które zostanie zwrócone przez poprzednie „wcielenie", które zostanie zwrócone przez poprzednie „wcielenie", itd.

Funkcje rekurencyjne trzeba stosować z umiarem i umiejętnie. Nieprzemyślane zastosowanie rekurencji może bowiem prowadzić do kombinatorycznej eksplozji liczby wywołań i rozmiaru stosu potrzebnego do realizacji rekurencji. Tak na przykład bywa, gdy w treści funkcji wywołanie samej siebie występuje dwukrotnie, jak w klasycznym przykładzie *nieprawidłowego* obliczania wartości wyrazów ciągu Fibonacciego. Definicja tego ciągu jest następująca:

$$F_n = \begin{cases} n & \text{dla } 0 \leqslant n < 2, \\ F_{n-1} + F_{n-2} & \text{dla } n \geqslant 2 \end{cases}$$

Naturalnym sposobem zaprogramowania takiej funkcji jest postać rekurencyjna jak w następującym programie (w linii ①, po dwukropku, mamy tu *dwukrotne* wywołanie

przez funkcję samej siebie):

P77: ***fib.cpp*** Ciąg Fibonacciego: zły przykład rekurencji

```
#include <iostream>
#include <iomanip>
using namespace std;

int licznik;

int fib(int n) {
    licznik++;
    return (n < 2) ? n : fib(n-1) + fib(n-2);     ①
}

int main() {
    cout << "\n  i       Fib(i)   # wywolan\n"
            "---------------------------" << endl;
    for (int i = 10; i <= 43; i += 3) {
        licznik = 0;
        int w = fib(i);
        cout << setw(3)  << i << setw(12) << w    ②
             << setw(12) << licznik << endl;
    }
}
```

W programie wprowadziliśmy zmienną globalną licznik zliczającą liczbę wywołań funkcji **fib** podczas znajdowania elementów ciągu Fibonacciego — zmienna ta jest zerowana przed przystąpieniem do wyliczania każdego kolejnego elementu ciągu. W ten sposób możemy wypisać nie tylko wartości kolejnych liczb Fibonacciego, ale również liczbę wywołań funkcji, jak były potrzebne do ich wyliczenia. Obliczenia zabrały kilka sekund; większość tego czasu zużyta została na obliczenie ostatniej wartości, czyli F_{43}. Poniższy wydruk z tego programu pokazuje, że do obliczenia F_{43} potrzebne było prawie półtora miliarda wywołań!

```
  i       Fib(i)   # wywolan
---------------------------
 10           55         177
 13          233         753
 16          987        3193
 19         4181       13529
 22        17711       57313
 25        75025      242785
 28       317811     1028457
 31      1346269     4356617
 34      5702887    18454929
 37     24157817    78176337
```

```
40    102334155    331160281
43    433494437   1402817465
```

Oczywiście wykonanie wersji „normalnej”, a więc tzw. iteracyjnej, tej funkcji zabiera ułamek sekundy i w ogóle nie zużywa miejsca na stosie. [Użyte w tym programie tzw. manipulatory (**setw** w linii ②) służą jedynie do przejrzystego sformatowania tabelki wyników — będą one omówione w rozdz. 16.3.2, str. 333].

11.9 Funkcje statyczne

Funkcje można deklarować/definiować z modyfikatorem **static**. Znaczenie tego modyfikatora jest jednak zupełnie inne dla funkcji globalnych, a więc nie będących składowymi klas, i dla funkcji składowych klas (ten ostatni przypadek omówimy w rozdz. 14.5, str. 275).

Wszystkie funkcje globalne są widoczne nie tylko w module (jednostce translacji), gdzie zostały zdefiniowane, ale we wszystkich tych modułach składających się na program, w których pojawia się ich deklaracja.

Natomiast funkcja globalna zadeklarowana/zdefiniowana w pewnym module z modyfikatorem **static** nie jest „eksportowana”: może być użyta tylko w tym module, podobnie jak to miało miejsce dla globalnych zmiennych statycznych — patrz rozdz. 7.3.1 o klasach pamięci na str. 85. Dzięki temu w innym module programu można użyć tej samej nazwy dla zupełnie innej funkcji statycznej: kolizji nazw nie będzie, bo w każdym module znana będzie tylko jedna funkcja o tej nazwie — ta pochodząca z tego modułu. Z tego mechanizmu praktycznie rzadko się korzysta; jeśli chodzi nam o zabezpieczenie się przed przypadkowymi kolizjami nazw, to lepszym i bardziej elastycznym mechanizmem są przestrzenie nazw (patrz rozdz. 23.2 na str. 510).

11.10 Funkcje rozwijane

Funkcje, zarówno globalne, jak i, o czym się przekonamy w późniejszych rozdziałach, będące składowymi klas, mogą być **rozwijane** (ang. *inlined*). O takich funkcjach mówi się też **funkcje otwarte** lub **wklejane**. Rozwijanie oznacza, że kod (instrukcje maszynowe) funkcji jest przez kompilator umieszczany w każdym miejscu pliku wykonywalnego, gdzie funkcja powinna, według treści programu, być wywołana. Zatem tak naprawdę taka funkcja w ogóle nie będzie wywoływana i, co więcej, w ogóle w kodzie wynikowym jako osobna funkcja nie będzie istnieć.

Gdybyśmy na przykład napisali trywialną funkcję typu

```
int fun(int m) {
    return 2*m;
}
```

to korzyść z niej byłaby wątpliwa: wywołanie funkcji wiąże się z zapamiętaniem aktualnego stanu rejestrów, wykonaniem skoku, powrotu itd.; jest to operacja dość kosztowna. Program może być nieco dłuższy, ale wykonywać się będzie szybciej, jeśli

mnożenie przez dwa umieścimy bezpośrednio w kodzie w każdym miejscu, gdzie tę funkcję wołaliśmy, a wywołania funkcji i samą funkcję w ogóle usuniemy. Z drugiej strony, umieszczanie takiego samego czy bardzo podobnego fragmentu kodu w wielu miejscach programu mogłoby być niezwykle uciążliwe, a co gorsza, czyniłoby ten program zupełnie nieczytelnym. Strach pomyśleć, co by było, gdybyśmy taki zastępujący wywołanie funkcji fragment musieli zmodyfikować: trzeba by to robić w każdym miejscu uważając na ewentualne konflikty nazw i kontekst, w jakim ten fragment pojawia się w różnych częściach programu. Aby to sobie ułatwić, możemy po prostu zadanie to zlecić kompilatorowi. W naszym programie definiujemy funkcję z modyfikatorem **inline**, na przykład

```
inline int fun(int m) {
    return 2*m;
}
```

Modyfikator **inline** mówi, że kompilator *nie* powinien tej funkcji kompilować tak jak innych funkcji, ale zastąpić każde jej wywołanie fragmentem kodu odpowiadającym jej treści. Wykonywalny plik wynikowy może być nieco dłuższy, ale wykonywać się będzie szybciej; łatwiej też będzie zmienić mnożenie przez dwa na mnożenie przez trzy — w tekście programu zrobimy to w jednym miejscu, a kompilator zadba o to, by w kodzie wykonywalnym zmiana zaszła we wszystkich miejscach gdzie tej funkcji użyliśmy.

W miejscu, gdzie następuje w programie wywołanie funkcji rozwijanej, musi być już znana definicja funkcji!

Definicja, a nie tylko deklaracja. Ponieważ kompilator musi zastąpić wywołanie funkcji jej treścią, sama deklaracja *nie* wystarczy. Pociąga to pewną niekonsekwencję w stosunku do tego, co mówiliśmy wcześniej: normalne funkcje, czyli funkcje nierozwijane, a więc **zamknięte**, deklarować można wiele razy, ale definiować tylko raz. W przypadku funkcji rozwijanych definicja (taka sama) musi istnieć w każdym module, gdzie funkcja jest używana, i to leksykalnie przed miejscem tego użycia. Zwykle najwygodniej jest zatem umieścić tę definicję w pliku nagłówkowym i włączać na początku każdego modułu, gdzie funkcja ma być użyta.

Pamiętać jednak trzeba, że dodanie do definicji modyfikatora **inline** nie gwarantuje, że funkcja rzeczywiście zostanie rozwinięta. Po prostu, jest to często dla kompilatora za trudne; jeśli istnieje prawdopodobieństwo błędu lub zniekształcenia możliwych intencji programisty — których przecież kompilator dokładnie nie zna — to funkcja *nie* będzie rozwijana: zostanie skompilowana jako normalna funkcja, czyli funkcja zamknięta. To, czy dana funkcja zdefiniowana z modyfikatorem **inline** będzie czy nie będzie rozwinięta, zależy w dużym stopniu od użytego kompilatora. Niektóre kompilatory informują o tym podczas kompilacji (choć większość tego nie robi).

Zwykle funkcja nie będzie rozwinięta, jeśli zawiera skomplikowane instrukcje warunkowe, pętle, szczególnie z „ifami” w środku, wywołania rekurencyjne itd.

Jako rozwijane (zadeklarowane jako **inline**) definiuje się najczęściej krótkie, proste funkcje, ale takie, które wywoływane są wielokrotnie, bo na przykład używane są

wewnątrz mocno zagnieżdżonej pętli. W zasadzie zazwyczaj *nie* rozwijamy funkcji aż do końcowego etapu pisania programu: wtedy dopiero sprawdzamy (za pomocą profilowania), które funkcje „zjadają" najwięcej czasu i te właśnie funkcje rozwijamy (obserwując, czy daje to jakiś zauważalny pozytywny efekt).

11.11 Przeciążanie funkcji

Funkcje można w C++ (ale nie w C) **przeciążać**. Oznacza to sytuację, gdy w tym samym zakresie są widoczne deklaracje/definicje kilku funkcji o tej samej nazwie. Oczywiście, aby na etapie kompilacji było wiadomo, o wywołanie której funkcji nam chodzi, wszystkie wersje funkcji muszą się wystarczająco różnić. Co to znaczy wystarczająco? Generalnie, muszą się różnić na tyle, aby można było jednoznacznie wybrać jedną z nich na podstawie wywołania.

Warunkiem koniecznym, choć niewystarczającym jest, aby funkcje o tej samej nazwie różniły się **sygnaturą**.

> Do sygnatury funkcji należy jej nazwa oraz liczba i typ parametrów *nie* licząc tych z wartościami domyślnymi. Typ funkcji, czyli typ wartości zwracanej, zwykle do sygnatury *nie* jest zaliczany.

Tak więc na przykład

```
int    fun(double x, int k =0);
double fun(double z);
```

to dwie deklaracje różnych funkcji, ale o takiej samej sygnaturze, a mianowicie o sygnaturze `fun(double)`. I rzeczywiście, wywołanie `fun(1.5)` byłoby najzupełniej legalnym i nie wymagającym żadnej niejawnej konwersji wywołaniem zarówno pierwszej, jak i drugiej z tych funkcji. Takie przeciążenie jest zatem nielegalne.

Natomiast

```
double fun(int);
double fun(unsigned);
```

to deklaracje funkcji różniących się sygnaturą. Wywołanie `fun(15)` jest wywołaniem pierwszej z nich, bo '15' jest literałem wartości typu **int** i do przekształcenia tej wartości do typu **unsigned** potrzebna byłaby konwersja. Zatem takie przeciążenie jest prawidłowe.

Z drugiej strony, różne sygnatury nie są jeszcze warunkiem dostatecznym na legalność przeciążenia. Widzimy to na przykładzie funkcji

```
void fun(int i);
void fun(int& i);
```

które mają różną sygnaturę, ale wywołanie `fun(k)`, gdzie `k` jest typu **int**, może być traktowane jako wywołanie zarówno pierwszej, jak i drugiej z nich. Zatem takie przeciążenie byłoby nieprawidłowe. Podobnie nieprawidłowe byłoby przeciążenie

```
void fun(int tab[]);
void fun(int * p);
```

lub

```
void fun(tab[3][3]);
void fun(tab[5][3]);
```

bo pierwszy wymiar tablicy wielowymiarowej nie ma znaczenia do określenia typu (jest i tak pomijany w tego rodzaju deklaracji/definicji). Natomiast

```
void fun(tab[3][3]);
void fun(tab[3][5]);
```

prawidłowo deklaruje dwie przeciążone funkcje **fun**, gdyż tablice wielowymiarowe różniące się wymiarem innym niż pierwszy *są* różnych typów i pomiędzy tymi typami nie ma niejawnej konwersji.

Argument typu **T** może być użyty przy wywołaniu funkcji z parametrem typu **T**, **const T** i **volatile T**, więc funkcje przeciążone nie mogą się różnić tylko typem takiego parametru. Zatem

```
int fun(int k);
int fun(const int k);
```

byłoby nielegalne.

Natomiast typy parametrów **T***, **volatile T*** i **const T*** (i analogicznie **T&**, **volatile T&** i **const T&**) są wystarczająco różne: patrząc na wywołanie funkcji kompilator może stwierdzić, czy użyta tam zmienna była ustalona lub ulotna czy nie; przeciążenie

```
int fun(int& k);
int fun(const int& k);
```

jest zatem prawidłowe.

Może się jednak zdarzyć, że to samo wywołanie funkcji pasuje do kilku jej przeciążonych wersji po ewentualnym dokonaniu dozwolonych konwersji. Jak rozstrzygnąć, która z tych funkcji będzie wywołana?

Proces poszukiwania takiej funkcji przebiega etapami i kończy się, gdy zostanie znalezione dopasowanie funkcji do wywołania. Tak więc sprawdzane są kolejno różne typy (stopnie) dopasowania:

Dopasowanie dokładne. Wszystkie typy argumentów są identyczne jak typy odpowiednich parametrów.

Dopasowanie po konwersji trywialnej. Do pełnego dopasowania wystarczy konwersja trywialna argumentu (ang. *minor conversion*). Konwersje trywialne to (`T` oznacza pewien typ, `fun` funkcję — o wskaźnikach funkcyjnych powiemy w rozdz. 11.12 na stronie 184):

Tablica 11.1: Konwersje trywialne

T	→	T&
T&	→	T
T[]	→	T*
fun()	→	(*fun)()
T	→	const T
T	→	volatile T
T*	→	const T*
T*	→	volatile T*

Dopasowanie po promocji. Do uzyskania dopasowania wystarczą standardowe promocje całościowe lub zmiennopozycyjne — patrz rozdz. 10.1 na stronie 147 — na przykład **char** → **int**.

Dopasowanie po innej konwersji niejawnej. Do uzyskania dopasowania wystarczą standardowe konwersje nie będące promocjami całościowymi lub zmiennopozycyjnymi — patrz rozdz. 10.1 — na przykład **int** → **double** lub odwrotnie.

Dopasowanie po konwersji zdefiniowanej przez użytkownika. Do uzyskania dopasowania potrzebne są konwersje zdefiniowane przez użytkownika — jak takie konwersje definiować, omówimy w rozdz. 20.1 na stronie 439.

Dopasowanie do funkcji o zmiennej liczbie parametrów. Ostatnia, rozpaczliwa próba dopasowania może być podjęta, jeśli wywołanie może pasować do funkcji o nieustalonej liczbie parametrów.

Znajdowanie dopasowania to skomplikowany proces, którego wynik nie zawsze jest intuicyjny. Rozpatrzmy program:

P78: ***match.cpp*** Dopasowywanie funkcji przeciążonych do wywołania

```
#include <iostream>
#include <string>
using namespace std;

string fun1(   int) { return "\'int\'\n";     }
string fun1(  char) { return "\'char\'\n";    }
string fun1(double) { return "\'double\'\n"; }

string fun2( short) { return "\'short\'\n";   }
string fun2(double) { return "\'double\'\n"; }

int main() {
    int    kin =   0;
    char   kch = '\0';
    float  kfl =   0;
    double kdo =   0;

    cout << "fun1(   int) -> " << fun1(kin);
```

```
19      cout << "fun1(  char) -> " << fun1(kch);
20      cout << "fun1( float) -> " << fun1(kfl);   ①
21      cout << "fun1(double) -> " << fun1(kdo);
22
23      cout << "fun2( float) -> " << fun2(kfl);   ②
24    //cout << "fun2(  char) -> " << fun2(kch);
25    //cout << "fun2(   int) -> " << fun2(kin);
26  }
```

Definiujemy trzy funkcje o tej samej nazwie **fun1**. Funkcje zwracają identyfikujący je napis. Wywołujemy je w funkcji **main** z argumentami różnych typów. Z wydruku

```
fun1(   int) -> 'int'
fun1(  char) -> 'char'
fun1( float) -> 'double'
fun1(double) -> 'double'
fun2( float) -> 'double'
```

widzimy, że zawsze wywołana jest ta funkcja, której parametr dokładnie pasuje do typu argumentu. Wyjątkiem jest wywołanie z argumentem typu **float** z linii ① — tu jednak nie dziwi nas, że wybrana została funkcja z parametrem typu **double**: wystarczyła tu jedna standardowa promocja zmiennopozycyjna (konwersja **float** → **int** też jest standardową konwersją, ale nie standardową promocją).

Ciekawsze są natomiast wywołania funkcji **fun2**. Istnieją jej dwie wersje: z argumentem typu **short** i **double**. Nie ma kłopotów z wywołaniem funkcji z argumentem typu **float** (②) — konwersja **float** → **double** jest standardową promocją. Ale zakomentowane wywołanie `fun2(kch)`, gdzie kch jest typu **char** jest błędne! Argument jest tu typu **char**, więc wydawałoby się, że ma „bliżej" do typu **short** niż do typu **double**. Standardową promocją całościową jest jednak promocja **char** → **int**; promocja **char** → **short** nią nie jest i wobec tego jest uznana za „dopasowanie po innej konwersji niejawnej", a więc takie samo jak **char** → **double**. Zatem kompilator zgłosi błąd uznając obie wersje funkcji **fun2** za tak samo dobre (albo tak samo złe).

Podobnie jest dla wykomentowanego wywołania tej samej funkcji z argumentem typu **int** (ostatnia linia). Wydaje się, że lepsza jest promocja **int** → **double**, będąca konwersją rozszerzającą, niż przekształcenie **int** → **short**, które jest konwersją zawężającą, a więc potencjalnie „gubiącą" informację. A jednak są one tak samo dobre/złe: ponieważ konwersja pierwsza jest, co prawda, promocją, ale nie całościową, obie znowu wpadają do tej samej kategorii „dopasowanie po innej konwersji niejawnej".

Oczywiście stosowanie takich nieczytelnych przeciążeń może prowadzić do trudno wykrywalnych błędów i generalnie powinniśmy ich unikać.

11.12 Wskaźniki funkcyjne

Bardzo ważnym pojęciem w C/C++ są wskaźniki funkcyjne. Pamiętamy, że zmienne wskaźnikowe to takie, których wartością jest adres obiektu: liczby, napisu, tablicy

itd. Definiując zmienną wskaźnikową musimy wskazać, na obiekty jakiego typu będzie ona wskazywać. Potrzebne jest to, między innymi, dlatego, aby można było stosować arytmetykę wskaźników opisaną już w rozdz. 4.6 na stronie 40.

Obiektami szczególnego rodzaju są funkcje. Tak jak wartość zmiennej zapisana jest gdzieś w pamięci komputera, a zatem ma określony adres, tak i funkcja, w postaci jej binarnego kodu, musi oczywiście być gdzieś w pamięci komputera obecna. Jest zatem sens mówić o jej adresie. Skoro tak, to ma również sens pojęcie wskaźnika do funkcji — **wskaźnika funkcyjnego**.

Pomiędzy „normalnymi" a funkcyjnymi wskaźnikami jest jednak głęboka różnica. Określając typ zmiennej wskazywanej dla zwykłego wskaźnika określamy jednocześnie rozmiar pojedynczego wskazywanego przez ten wskaźnik obiektu. Takie obiekty możemy układać w pamięci kolejno jeden po drugim tworząc tablice. Jeśli znamy adres jednego z nich, to znając rozmiar pojedynczego obiektu znamy położenie drugiego, trzeciego i, ogólnie, n-tego. W przypadku funkcji jednak tak nie jest. Każda funkcja jest inna i, oczywiście, jej binarny kod ma inny rozmiar. Zatem:

Dla wskaźników funkcyjnych nie obowiązują normalne zasady arytmetyki wskaźników.

Inna sprawa to kontrola typów. Generalnie *zmienna* to obszar pamięci przeznaczony na daną oraz informacja o typie tej danej. Informacja o typie jest potrzebna, aby móc prawidłowo zinterpretować binarną reprezentację zapisanej w pamięci danej, jak i po to, aby móc skontrolować poprawność operacji na niej. W przypadku funkcji kompilator również sprawdza czy ich użycie jest zgodne z ich specyfikacją wyrażoną w deklaracji czy definicji. Zatem w przypadku wskaźników funkcyjnych musi istnieć mechanizm pozwalający nie tylko na określenie adresu funkcji, ale też jej typu, liczby parametrów, typie tych parametrów itd.

Jak zatem zdefiniować wskaźnik do funkcji? Najlepiej zrozumieć to na podstawie przykładów. Przed ich analizą warto przypomnieć sobie zasady odczytywania definicji typów pochodnych, o których mówiliśmy w rozdz. 6.1 na stronie 73.

Czym wobec tego jest fun po następującej instrukcji:

```
int (*fun)(int);
```

Czytamy zgodnie z zasadami z rozdz. 6.1: ZMIENNA fun JEST — na prawo nawias zamykający, więc patrzymy w lewo — WSKAŹNIKIEM DO — wychodzimy z nawiasu, patrzymy w prawo — FUNKCJI Z JEDNYM PARAMETREM TYPU int ZWRACAJĄCEJ — patrzymy znów w lewo — WARTOŚĆ TYPU int. A zatem zmienna fun jest zmienną wskaźnikową przystosowaną do przechowywania adresów funkcji, ale nie dowolnych funkcji, tylko takich, które pobierają jeden argument typu int i zwracają wartość typu int. Podobnie jak po definicji

```
int *k;
```

zmienna wskaźnikowa k istnieje, ale na razie nie zawiera żadnego użytecznego adresu, tak i w naszym przypadku funkcyjny wskaźnik fun po takiej definicji już istnieje, ale nie zawiera adresu żadnej konkretnej funkcji.

Zauważmy też, że nie można było opuścić nawiasów wokół `*fun`. Gdybyśmy je opuścili, to dostalibyśmy

```
int *fun(int);
```

co przeczytalibyśmy tak: ZMIENNA fun JEST — na prawo nawias otwierający — FUNKCJĄ Z JEDNYM PARAMETREM TYPU **int** ZWRACAJĄCĄ — patrzymy w lewo — WSKAŹNIK DO — dalej w lewo — ZMIENNEJ TYPU **int**. A zatem byłaby to *deklaracja* funkcji **fun**.

Rozpatrzmy inny przykład:

```
double (*fun[3])(double);
```

Czytamy: ZMIENNA fun JEST — na prawo nawias kwadratowy otwierający — TRZYELEMENTOWĄ TABLICĄ — przechodzimy na lewo — WSKAŹNIKÓW DO — wychodzimy z nawiasu, patrzymy w prawo — FUNKCJI Z JEDNYM PARAMETREM TYPU **double** ZWRACAJĄCEJ — patrzymy znów w lewo — WARTOŚĆ TYPU **double**. A zatem zmienna fun jest tablicą trzech wskaźników do funkcji określonego typu.

Jeśli pfun jest wskaźnikiem funkcyjnym pewnego typu, to jak możemy wpisać do niego adres istniejącej funkcji? Podobnie jak do wskaźnika p typu **int*** wpisujemy adres istniejącej zmiennej k typu **int** pisząc

```
p = &k;
```

tak na zmienną pfun można przypisać

```
pfun = &fun;
```

gdzie fun jest nazwą pewnej funkcji odpowiedniego typu. Konwersja od fun do &fun jest konwersją trywialną, tak więc równie dobrze w przypisaniu możemy pominąć operator wyłuskania adresu i napisać po prostu

```
pfun = fun;
```

(czego, oczywiście, nie moglibyśmy zrobić dla zmiennych typu np. **int**) Pamiętać tylko trzeba, żeby *nie* napisać nawiasów przy nazwie funkcji — nawiasy pełnią rolę operatora wywołania funkcji, tak więc

```
pfun = fun();
```

spowodowałoby wywołanie funkcji **fun** i próbę przypisania rezultatu do zmiennej pfun, najprawdopodobniej z opłakanym skutkiem.

Rozpatrzmy przykład programu w którym stosujemy wskaźniki funkcyjne:

P79: ***wskfun.cpp*** Wskaźniki funkcyjne

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  const double PI = 4*atan(1.);
6
7  double nasza(double);
8
9  int main() {
10     double (*f)(double);                                    ①
11
12     f = sin;                                                ②
13     cout << "sin(PI/2) =  " << (*f)(PI/2) << endl;          ③
14
15     f = &cos;                                               ④
16     cout << "  cos(PI) = "  << f(PI)      << endl;          ⑤
17
18     f = nasza;                                              ⑥
19     cout << " nasza(3) =  " << f(3)       << endl;          ⑦
20 }
21
22 double nasza(double x) {
23     return x*x;
24 }
```

Na początku włączamy nagłówek ***cmath***, aby mieć dostęp do funkcji matematycznych, w tym przypadku funkcji **sin** i **cos**.

W linii ① definiujemy zmienną f, która jest wskaźnikiem funkcyjnym przystosowanym do wskazywania funkcji pobierających jeden argument typu **double** i zwracających wartość typu **double**. W linii ② inicjujemy wartość tego wskaźnika adresem bibliotecznej funkcji **sin** (sinus) — jak już wiemy, możemy opuścić operator pobrania adresu '&'. W linii ④ zmieniamy wartość wskaźnika — teraz wskazuje na funkcję **cos** (kosinus) — a następnie zmieniamy jego wartość jeszcze raz tak, aby wskazywał na zdefiniowaną przez nas samych funkcję **nasza** (⑥). Wszystkie te funkcje są oczywiście, i muszą być, tego samego typu **double** → **double**. Jak widzimy, przy przypisywaniu na wskaźnik funkcyjny można używać operatora wyłuskania adresu lub go opuścić.

W liniach ③, ⑤ i ⑦ aktualnie wskazywana przez wskaźnik f funkcja jest wywoływana. Za każdym razem jest to inna funkcja, o czym przekonuje nas wydruk:

```
sin(PI/2) =  1
  cos(PI) = -1
 nasza(3) =  9
```

Skoro f jest wskaźnikiem, to formalnie najpierw powinniśmy wyłuskać wskazywany obiekt (czyli funkcję) za pomocą operatora dereferencji '*', a następnie dokonać wy-

wołania: tak właśnie robimy w linii ③ (nawias wokół `*f` jest tu konieczny, gdyż operator wywołania funkcji, czyli nawiasy, ma wyższy priorytet od operatora dereferencji '*'). Jednak, jak widzimy w liniach ⑤ i ⑦, w przypadku wskaźników funkcyjnych dereferencja nie jest konieczna — można użyć identyfikatora wskaźnika funkcyjnego tak jak nazwy funkcji (oczywiście, dla „normalnych" wskaźników dereferencji nie można w ten sposób opuszczać).

Wskaźnik funkcyjny może też być parametrem funkcji. Deklaracja takiego parametru wygląda tak jak deklaracja wskaźnika funkcyjnego, tyle że w deklaracji funkcji można, choć nie trzeba, pominąć nazwę (w definicji jest ona jednak potrzebna). Tak więc

```
double fun( double (*f)(double), double a, double b) {
    return f(a) + f(b);
}
```

jest definicją funkcji **fun**, której pierwszym parametrem jest funkcja typu **double** → **double**, a dwoma następnymi są zwykłe parametry typu **double** (tu nazwane a i b). W treści funkcji **f** jest wywoływana z argumentem a i b. Tak zdefiniowaną funkcję możemy wywołać w programie na przykład tak (występująca tu funkcja **atan** oznacza biblioteczną funkcję arkus tangens):

```
#include <cmath>
const double PI = 4*atan(1.);
// ...
double result = fun(sin, 0, PI/2);
```

Jak wynika z treści funkcji **fun**, wynikiem ('result') powinna być liczba

$$\sin(0) + \sin(\pi/2) = 0 + 1 = 1$$

Oczywiście, jak zwykle, możemy pominąć nazwy parametrów, jeśli chodzi nam tylko o deklarację. Powyżej zdefiniowaną funkcję **fun** zadeklarować moglibyśmy zatem tak:

```
double fun(double(*)(double),double,double);
```

co wygląda dość ezoterycznie. Można uprościć sobie nieco życie wprowadzając synonim (alias) skomplikowanego typu za pomocą instrukcji **typedef** (o tej pożytecznej instrukcji mówiliśmy już w rozdz. 6.2 na stronie 76). W naszym przypadku moglibyśmy wprowadzić synonim typu *wskaźnik do funkcji z jednym parametrem typu* **double** *zwracającej* **double**:

```
typedef double (*FUNDtoD)(double);
```

i użyć go potem w deklaracji naszej funkcji **fun**:

```
double fun( FUNDtoD, double, double);
```

lub w deklaracjach/definicjach funkcyjnych zmiennych wskaźnikowych

```
#include <cmath>
// ...
FUNDtoD f = sin;
// ...
f = atan;
// ...
```

Takie funkcje od funkcji nie są niczym egzotycznym. Wyobraźmy sobie, że chcemy napisać funkcję obliczającą pierwiastki (miejsca zerowe) funkcji. Nie chcielibyśmy pisać osobnej takiej funkcji dla każdej możliwej funkcji, której pierwiastka szukamy. Opracujemy raczej ogólny algorytm, a konkretną funkcję, której pierwiastek chcemy policzyć, dostarczymy jako argument. Prostą realizację tego podejścia widzimy w następującym programie:

P80: ***root.cpp*** Funkcja jako parametr funkcji

```
#include <iostream>
#include <cmath>
#include <cassert>
using namespace std;

const double PI = 4*atan(1.);

typedef double (*FD2D)(double);                    ①

double root(FD2D, double, double);                 ②

double nasza(double x) {return 3 - x*(1 + x*(27 - x*9));}

int main() {
    double r;

    cout.precision(15);                            ③

    r = root(sin, 3, 4);                           ④
    cout << "sin         " <<  r    << endl
         << "dokladnie: " <<  PI   << endl << endl;

    r = root(cos, -2, -1.5);                       ⑤
    cout << "cos:        " <<  r    << endl
         << "dokladnie: " << -PI/2 << endl << endl;

    r = root(nasza, 0, 1);                         ⑥
    cout << "nasza:      " <<  r    << endl
         << "dokladnie: " <<  1./3 << endl;

    r = root([] (double x) -> double {             ⑦
```

```
32                  return 3-x*(1+x*(27-x*9));
33              }, 0, 1);
34     cout << "lambda:    " <<  r    << endl
35          << "dokladnie: " <<  1./3 << endl;
36 }
37
38 double root(FD2D fun, double a, double b) {
39     /* Znajdowanie pierwiastka funkcji metodą bisekcji.
40        fun(a) i fun(b) muszą być różnych znaków */
41     static const double EPS = 1e-15;
42     double f, s, h = b-a, f1 = fun(a), f2 = fun(b);
43
44     if (f1 == 0) return a;
45     if (f2 == 0) return b;
46     assert(f1*f2 < 0);                          ⑧
47
48     do {
49         if ((f = fun((s=(a+b)/2))) == 0) break;
50         if (f*f1 < 0) {f2 = f;b = s;}
51         else          {f1 = f;a = s;}
52     } while ((h /= 2) > EPS);
53
54     return (a+b)/2;
55 }
```

Typem pierwszego parametru funkcji **root** (②) jest funkcja (wskaźnik do funkcji) typu **double** → **double**. Typowi temu nadaliśmy nazwę (alias) **FD2D** (①). Oprócz wskaźnika funkcyjnego, funkcja **root** pobiera dwie liczby: są to wartości argumentów, pomiędzy którymi miejsce zerowe przekazanej funkcji będzie szukane. Dla prawidłowego działania algorytmu wymagane jest, aby w tych dwóch punktach wartości tej funkcji były *różnych* znaków. Aby nie dopuścić do produkowania bezsensownych wyników, warunek ten jest sprawdzany (⑧) za pomocą makra **assert** dostępnego dzięki dołączeniu pliku nagłówkowego ***cassert***. W argumencie **assert** podajemy wyrażenie logiczne. Wartość tego wyrażenia jest sprawdzana w czasie wykonania programu i jeśli rezultatem jest fałsz, program jest przerywany, a do standardowego strumienia błędów (***stderr***) wpisywany jest komunikat lokalizujący linię, gdzie to przerwanie nastąpiło. Sprawdzanie za pomocą **assert** można wyłączyć definiując dyrektywą preprocesora nazwę NDEBUG (np.: `#define NDEBUG` w pliku źródłowym lub `-DNDEBUG` jako opcja kompilacji).

Jeśli tylko znaki wartości funkcji **fun** w punktach a i b są różne, a funkcja jest ciągła, to nasz algorytm praktycznie zawsze powinien zadziałać i zwrócić znaleziony pierwiastek.

W programie głównym wywołujemy funkcję **root** cztery razy. Za pierwszym razem (④) posyłamy do niej jako pierwszy argument funkcję **sin** (sinus), a jako przedział do poszukiwania pierwiastka podajemy przedział $[3, 4]$. W tym przedziale funkcja **sin** ma dokładnie jedno miejsce zerowe, $x = \pi$, o czym my wiemy, ale czego nie wie oczywiście

funkcja **root**, pracowicie go szukając.

Następnie (⑤) obliczamy miejsce zerowe funkcji **cos** (kosinus) w przedziale $[-2, -1.5]$ (tym pierwiastkiem jest $-\pi/2$). W linii ⑥ szukamy miejsca zerowego funkcji **nasza** zdefiniowanej przed funkcją **main**. Funkcją tą jest wielomian

$$W(x) = 9x^3 - 27x^2 - x + 3$$

o pierwiastkach $x_{1,2} = \pm 1/3$ oraz $x_3 = 3$. Ponieważ pierwiastka poszukujemy w przedziale $[0, 1]$, powinniśmy znaleźć pierwiastek $x = 1/3$.

W końcu, w linii ⑦, szukamy zera funkcji równoważnej funkcji **nasza** ale zadanej tym razem w postaci funkcji lambda (więcej o takich funkcjach powiemy w rozdz. 11.13, str. 195).

Po znalezieniu miejsc zerowych drukujemy wyniki: pierwiastki znalezione przez funkcję **root** i pierwiastki dokładne, które w przypadku tych funkcji znamy. Wywołanie '`cout.precision(15)`' z linii ③ służy wyłącznie do tego, aby znalezione liczby wypisane zostały z dokładnością do 15 cyfr a nie, jak to jest domyślnie, z dokładnością do 6 cyfr znaczących. Więcej o sposobach formatowania wydruku w rozdz. 16.3.1 na stronie 328.

Jak widać,

```
sin         3.14159265358979
dokladnie: 3.14159265358979

cos:        -1.5707963267949
dokladnie: -1.5707963267949

nasza:      0.333333333333333
dokladnie: 0.333333333333333
lambda:     0.333333333333333
dokladnie: 0.333333333333333
```

pierwiastki zostały policzone prawidłowo, z dokładnością do 15 miesc znaczących.

Nic nie stoi na przeszkodzie, aby budować tablice funkcji (czyli właściwie wskaźników do funkcji). W poniższym programie najpierw definiujemy aliasy dla użytych w dalszej części typów (①, alias **TABNAM** jest tu zdefiniowany z użyciem nowej składni z C++11). Tak więc po

```
typedef double (*TABFUN[])(double);
```

TABFUN jest nazwą typu „tablica wskaźników do funkcji pobierających jeden argument typu **double** i zwracających wartość typu **double**". Nazwa **TABNAP** oznacza typ *tablica wskaźników do* **char**, czyli tablica C-napisów (będą to nazwy funkcji).

P81: ***tabfunc.cpp*** Tablice funkcji

```
1 #include <iostream>
2 #include <cmath>
```

```
3  using namespace std;
4
5  typedef double (*TABFUN[])(double);          ①
6  //typedef const char *TABNAP[];
7  using TABNAP = const char* [];  // C++11
8
9  void fundruk(TABFUN,TABNAP,double);
10
11 int main() {
12     const double PIover4 = atan(1.);
13
14     TABFUN tabfun = { sin,  cos,  tan };    ②
15     TABNAP tabnap = {"sin","cos","tan"};
16
17     cout << "sizeof(tabfun) = " << sizeof(tabfun) << endl
18          << "sizeof(tabnap) = " << sizeof(tabnap) << "\n\n";
19
20     for (int i = 0; i < 3; i++) {           ③
21         cout << "tabfun[" <<   i   << "](pi/4) = "
22              << tabfun[i](PIover4) << " ("
23              << tabnap[i]          << ")\n";
24     }
25
26     fundruk(tabfun,tabnap,PIover4);
27 }
28
29 void fundruk(TABFUN f, TABNAP t, double x) {
30     cout << "\n";
31     for (int i = 0; i < 3; i++) {
32         cout << "fundruk: " << t[i] << " "
33              << "wartosc: " << f[i](x) << endl;
34     }
35 }
```

W linii ② definiujemy obiekt typu **TABFUN**, czyli tablicę wskaźników do funkcji i inicjujemy ją adresami trzech istniejących funkcji. Co prawda, nie użyliśmy operatora wyłuskania adresu '&', ale został on domyślnie zastosowany. Tablica jest tablicą wskaźników, a nie funkcji jako takich — przekonuje nas o tym choćby wydrukowany rozmiar tej tablicy (24); odpowiada rozmiarowi trzech obiektów po osiem bajtów każdy.

```
sizeof(tabfun) = 24
sizeof(tabnap) = 24

tabfun[0](pi/4) = 0.707107 (sin)
tabfun[1](pi/4) = 0.707107 (cos)
tabfun[2](pi/4) = 1 (tan)
```

```
fundruk: sin wartosc: 0.707107
fundruk: cos wartosc: 0.707107
fundruk: tan wartosc: 1
```

Standard nie wymaga, aby wskaźniki funkcyjne miały ten sam rozmiar co inne wskaźniki — mogą one być zaimplementowane w zupełnie inny sposób. Dlatego nie ma żadnego sensu rzutowanie ich na jakikolwiek inny typ wskaźnikowy.

W pętli (③) wywołujemy kolejno funkcje z tablicy. Ponieważ element tabfun[i] jest wskaźnikiem do funkcji, który, jak wspomnieliśmy, jest konwertowany do funkcji, zapis `tabfun[i](Plover4)` jest wywołaniem funkcji, do której wskaźnik jest $(i+1)$-ym elementem tablicy tabfun; argumentem wywołania jest wartość zmiennej Plover4 (czyli liczba $\pi/4$).

Funkcja **fundruk** pobiera jako argumenty tablicę wskaźników funkcyjnych, tablicę napisów (wskaźników typu **char***) i liczbę typu **double**. W ciele funkcji wykonujemy te same wywołania co w funkcji **main**: funkcje które należy wywołać, zostały przekazane w tablicy, która wewnątrz funkcji **fundruk** nazywa się f.

Tak jak zwykłe wskaźniki, tak i wskaźnik do funkcji może być nie tylko parametrem, ale i wartością zwracaną funkcji.

P82: ***funret.cpp*** Wskaźnik do funkcji jako wartość zwracana

```
#include <iostream>
#include <cmath>
using namespace std;

typedef double (*FUNDtoD)(double);                          ①
typedef FUNDtoD TABFUN[];

FUNDtoD funmax(TABFUN,double);                              ②

double fun0(double x) { return log(x); }
double fun1(double x) { return x*x;    }
double fun2(double x) { return exp(x); }
double fun3(double x) { return sin(x); }
double fun4(double x) { return cos(x); }

int main() {
    TABFUN tabfun = { fun0, fun1, fun2, fun3, fun4 };        ③

    FUNDtoD fun = funmax(tabfun,1);                         ④

    int i;
    for (i = 0; i < 5; ++i)
        if (fun == tabfun[i]) break;

    cout.precision(14);
```

```
26      cout << "Najwieksza wartosc dla x=1 ma funkcja nr "
27           << i << ".\nWynosi ona " << fun(1) << endl;    ⑤
28  }
29
30  FUNDtoD funmax(TABFUN f, double x) {
31      double m = f[0](x),z;
32      int    k = 0;
33
34      for (int i = 1; i < 5; i++) {
35          if ( (z = f[i](x)) > m) {
36              m = z;
37              k = i;
38          }
39      }
40      return f[k];
41  }
```

Najpierw (①) definiujemy alias **FUNDtoD** dla typu *wskaźnik do funkcji* **double** → **double** i za jego pomocą w linii następnej definiujemy synonim **TABFUN** dla typu *tablica wskaźników do funkcji* **double** → **double**. Dzięki tym aliasom deklaracja funkcji **funmax** (②) jest w miarę czytelna: jest to funkcja, której pierwszym argumentem będzie tablica wskaźników funkcyjnych, a wartością zwracaną pojedynczy wskaźnik do funkcji. Zapisanie tego bez użycia instrukcji **typedef** prowadziłoby do niezbyt czytelnych wyrażeń. Zadaniem funkcji **funmax** jest znalezienie tej z pięciu funkcji, dla której wartość w punkcie x jest największa. Wskaźnik do tej właśnie funkcji jest wartością zwracaną przez **funmax**.

Definijemy zatem pięć prostych funkcji odpowiedniego typu. W funkcji **main** tworzymy tablicę wskaźników funkcyjnych (obiekt typu **TABFUN** — ③) i inicjujemy ją adresami naszych pięciu funkcji. Następnie posyłamy tablicę do funkcji **funmax** (④) i wartość zwracaną, która jest wskaźnikiem funkcyjnym, zapamiętujemy w zmiennej fun, która jest odpowiedniego typu **FUNDtoD**.

W pętli sprawdzamy, którą z naszych pięciu funkcji otrzymaliśmy jako wynik działania funkcji **funmax**. Wypisujemy numer funkcji zwróconej przez **funmax** i tę znalezioną funkcję wywołujemy dla argumentu 1 (⑤), aby się przekonać, że jest to rzeczywiście właściwa funkcja. W naszym przypadku powinna to być funkcja **exp(x)** (czyli e^x), która dla $x = 1$ przyjmuje wartość równą podstawie logarytmu naturalnego (w przybliżeniu 2.71828). Wydruk przekonuje nas, że tak jest w istocie:

```
Najwieksza wartosc dla x=1 ma funkcja nr 2.
Wynosi ona 2.718281828459
```

Uogólnieniem pojęcia funkcji w C++ jest *obiekt funkcyjny*. O takich obiektach będziemy mówić w rozdziale 24.2.2, str. 536.

11.13 Funkcje lambda

Począwszy od wersji C++11 wprowadzono możliwość definiowania tzw. **funkcji lambda**. Są to anonimowe funkcje, które można definiować lokalnie. Składnia jest następująca:

```
[ lista przechwytywania ] ( parametry ) -> typ_zwracany { ciało }
```

W nawiasie okrągłym podajemy listę parametrów, jak dla zwykłej funkcji. Po „strzałce" podajemy typ zwracany funkcji. W pewnych (a właściwie w większości) sytuacji można tę część opuścić, a kompilator sam ten typ wydedukuje — będzie to **decltype** zwracanego wyrażenia (o **decltype** pisaliśmy w rozdziale 4.1, str. 27). Jeśli w ciele funkcji nie ma instrukcji **return**, wydedukowanym typem będzie **void**.

Na początku wyrażenia mamy kwadratowe nawiasy, które mogą być puste — oznacza to wtedy, że wszystkie potrzebne dane funkcja otrzymuje poprzez argumenty wywołania. Można jednak umieścić w tych nawiasach, oddzielone przecinkami, symbole:

znak równości ('=')
: funkcja będzie miała dostęp do *kopii* wartości wszystkich zmiennych *lokalnych* z bieżącego zakresu, z wyjątkiem tych wymienionych jawnie (patrz niżej);

ampersand ('&')
: funkcja będzie miała dostęp do odniesień (referencji, a więc oryginałów) do wszystkich zmiennych lokalnych z bieżącego zakresu; znów — z wyjątkiem tych wymienionych jawnie;

`var`
: gdzie var jest nazwą zmiennej lokalnej: funkcja będzie miała dostęp do *kopii* wartości tej zmiennej;

`&var`
: gdzie var jest nazwą zmiennej lokalnej: funkcja będzie miała dostęp do referencji do tej zmiennej.

Na przykład, `[&,a]` znaczy, że w tworzonej funkcji lambda wszystkie zmienne będą widoczne poprzez referencje, a zmienna a przez jej obecną wartość. Natomiast `[=,&a,&b]` znaczy, że w lambdzie widoczne będą kopie wszystkich zmiennych lokalnych z wyjątkiem a i b, które będą widoczne „w oryginale".

Typem funkcji lambda jest nieokreślony przez standard typ, zwykle inny dla każdej lambdy. Co jednak ważne, typ ten jest konwertowalny do typu **function<Typ(Typy)>** (z nagłówka ***functional***), gdzie **Typ** jest typem zwracanym funkcji (być może **void**), a **Typy** to typy argumentów oddzielone przecinkami. Zwykłe wskaźniki funkcyjne też są konwertowane to tego rodzaju typów automatycznie. W wielu, ale nie wszystkich, przypadkach można się posłużyć słowem kluczowym **auto** aby uniknąć konieczności jawnego definiowania typu.

Rozpatrzmy przykład:

P83: ***lambdas.cpp*** Funkcje lambda

```
#include <iostream>
#include <functional>
using std::cout; using std::endl;

double square(double x) {
    return x*x;
}

void invoke(std::function<double(double)> f, double arg) {
    double res = f(arg);
    cout << "invoke(" << arg << ")=" << res << endl;
}

int main() {
      // pomocnicza funkcja lambda
    auto print =
        [](double p1, double p2, double p3,
           double arg, double val) -> void
        {
            cout << " a=" << p1 << " b=" << p2
                 << " c=" << p3 << " x=" << arg
                 << " res=" << val << endl;
        };

      // funkcja lambda a*x*x+b*x+c
    int a = 1, b = 1, c = 1;
      // lokalne zmienne przez wartość
    auto pol1 =
        [=](double x) -> double
        {
            double res = c+x*(b+x*a);
            print(a,b,c,x,res);
            return res;
        };
    cout << "pol1=" << pol1(2) << endl;
    a = b = c = 2;
    cout << "pol1=" << pol1(2) << endl << endl;

      // lokalne zmienne przez referencje
    auto pol2 =
        [&](double x) -> double
        {
            double res = c+x*(b+x*a);
            print(a,b,c,x,res);
```

```
45              return res;
46          };
47      cout << "pol2=" << pol2(2) << endl;
48      a = b = c = 1;
49      cout << "pol2=" << pol2(2) << endl << endl;
50
51        // a i c przez referencje, b i print przez wartość
52      auto pol3 =                                          ①
53          [&a,b,&c,print](double x) -> double
54          {
55              double res = c+x*(b+x*a);
56              print(a,b,c,x,res);
57              return res;
58          };
59      cout << "pol3=" << pol3(2) << endl;                  ②
60      a = b = c = 2;                                       ③
61      cout << "pol3=" << pol3(2) << endl << endl;          ④
62
63        // typ określony jawnie
64      std::function<double(double)> f = pol3;
65      invoke(f,2);
66        // konwersja zwykłych wskaźników funkcyjnych
67      invoke(square,2);
68      f = square;
69      invoke(f,2);
70        // lambda w argumencie
71      invoke([](double x) {return x*x*x;}, 3);
72
73        // dla void->void tylko nawiasy i ciało
74      [] {
75          cout << "Done" << endl;
76      }(); // zdefiniuj funkcję i od razu ją wywołaj
77  }
```

Funkcja **invoke** pobiera funkcję lambda (lub wskaźnik do funkcji odpowiedniego typu) i wywołuje przekazaną funkcję dla podanego argumentu. Na początku funkcji **main** (a więc *wewnątrz* funkcji) definiujemy pomocniczą funkcję lambda **print** z pustą listą przechwytywania (cała informacja będzie przekazywana poprzez argumenty). Zauważmy, że print samo jest tu zmienną lokalną. Funkcja ta jest potem wywoływana kilka razy w treści programu. Następnie używając słowa kluczowego **auto** definiujemy kilka prostych funkcji (**pol1**, **pol2**, **pol3** — wszystkie są implementacją tego samego wielomianu drugiego stopnia $ax^2 + bx + c$) używając różnych list przechwytywania: jedne zmienne lokalne są przekazywane przez wartość, a więc kopiowane są wartości jakie przyjmują w momencie definiowania funkcji, do innych funkcja będzie miała dostęp przez referencję, a więc będą w niej widoczne zmiany wartości odpowiednich zmiennych. Na przykład, w linii ① definiujemy lambdę z listą przechwytywania za-

wierającą aktualne wartości b (czyli 1) i **print** oraz referencje do a i c (które również mają wartość 1). Wywołując funkcję z $x = 2$ (linia ②), otrzymujemy 7. Następnie zmieniamy wartości zmiennych a, b i c — teraz wszystkie wynoszą 2 (linia ③). Jednak wartość b widziana przez funkcję w dalszym ciągu wynosi 1, bo zapamiętana została wartość przyjmowana w momencie definiowania lambdy. Z drugiej strony, zmienne a i c widziane są przez referencje, a więc ich zmiany będą widoczne w funkcji i wywołanie z linii ④ da rezultat 12.

W końcowej części programu demonstrujemy konwersje zwykłych wskaźników funkcyjnych i przekazywanie funkcji lambda i wskaźników funkcyjnych do innych funkcji (w tym przypadku do funkcji **invoke**). Widać, że wskaźnik jest niejawnie konwertowany do typu **std::function<double(double)>** (konwersja w drugą stronę nie zachodzi).

Ważne jest przeanalizowanie programu i zrozumienie otrzymanego rezultatu:

```
 a=1 b=1 c=1 x=2 res=7
pol1=7
 a=1 b=1 c=1 x=2 res=7
pol1=7

 a=2 b=2 c=2 x=2 res=14
pol2=14
 a=1 b=1 c=1 x=2 res=7
pol2=7

 a=1 b=1 c=1 x=2 res=7
pol3=7
 a=2 b=1 c=2 x=2 res=12
pol3=12

 a=2 b=1 c=2 x=2 res=12
invoke(2)=4
invoke(3)=27
Done
```

W nowszych wersjach standardu można parametry funkcji lambda deklarować z użyciem **auto**. Kompilator sam wtedy utworzy odpowiednie wersje funkcji dedukując typy parametrów na podstawie typów podanych argumentów. Na przykład program:

P84: *lambdagen.cpp* Dedukcja typów w lambdach

```
1 #include <iostream>
2
3 int main() {
4     using namespace std::literals;
5
6     auto pr = [] (auto e) {
```

```
7           std::cout << "Result is " << e << '\n';
8       };
9       auto f = [] (auto e1, auto e2) {
10          return e1 < e2 ? e1 : e2;
11      };
12      auto ri = f(3, 1);
13      pr(ri);
14      auto rs = f("Cindy"s , "Alice"s);
15      pr(rs);
16  }
```

drukuje

```
Result is 1
Result is Alice
```

Jak widzimy, jedna lambda „obsługuje" różne typy zwracane i typy argumentów (jest to możliwe dzięki przeciążeniom metody **operator()** w klasie reprezentującej lambdę). Zauważmy, że napisy `"Alice"` i `"Cindy"` zostały podane z literą 's' na końcu. Taki zapis oznacza, że napis ten ma być traktowany jako literał klasy **string**, a nie jako C-napis — wtedy typem tego literału byłby **const char*** (takiej składni można używać po włączeniu przestrzeni nazw ***std::literals***).

Wartości przekazywane do listy przechwytywania lambdy są trakowane jako stałe. Można jednak, poprzez użycie słowa **mutable** za listą parametrów, dopuścić ich zmiany. Każde wywołanie może wtedy zmieniać ich wartość i wartość ta zachowuje się w obiekcie reprezentującym lambdę pomiędzy wywołaniami. Pozwala to, na przykład, budować lambdy pełniące rolę generatorów: funkcji bezparametrowych, dla których każde wywołanie dostarcza kolejnej wartości pewnej sekwencji. W programie poniżej lambda fibo (①) będzie zwracać w kolejnych wywołaniach kolejne liczby ciągu Fibonacciego

$$F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, \ldots, F_n = F_{n-2} + F_{n-1}, \ldots$$

a lambda triangle (②) kolejne liczby trójkątne

$$t_0 = 0, t_1 = 1, t_2 = 3, t_3 = 6, \ldots, t_n = t_{n-1} + n, \ldots$$

P85: ***lambdamutable.cpp*** Funkcje lambda z opcją **mutable**

```
1  #include <iomanip>      // setw
2  #include <iostream>
3
4  int main() {
5      using std::cout; using std::endl; using std::setw;
6
7      auto fibo = [fp=-1, fn=1] () mutable  {       ①
```

```
8               int d = fp; fp = fn; return fn += d;
9           };
10
11      auto triangle = [t=0, i=0] () mutable {       ②
12              return t += i++;
13          };
14
15      for (size_t i = 0; i <= 10; ++i)
16          cout << setw(2) << i << ":" << setw(3) << fibo()
17               << setw(3) << triangle() << endl;
18  }
```

Zauważmy, że wartości fp i fn (i podobnie t oraz i) *nie* są zmiennymi lokalnymi z otaczającego zakresu: są definiowane i inicjowane bezpośrednio na liście przechwytywania a ich typ jest automatycznie dedukowany przez kompilator na podstawie wartości inicjujących. Program drukuje

```
 0:  0  0
 1:  1  1
 2:  1  3
 3:  2  6
 4:  3 10
 5:  5 15
 6:  8 21
 7: 13 28
 8: 21 36
 9: 34 45
10: 55 55
```

Zauważmy, że jeśli opcja **mutable** występuje, to nawiasy okrągłe są potrzebne nawet wtedy, gdy funkcja jest bezparametrowa. Typ zwracany natomiast można pominąć, jeśli może on być jednoznacznie wydedukowany przez kompilator.

11.14 Szablony funkcji

Często wiele funkcji, jakie musimy definiować, jest do siebie bardzo podobnych — różnią się tylko typem argumentów, a czynności, które mają wykonać, są identyczne. Wyobraźmy sobie na przykład funkcję, która z podanej poprzez argument tablicy wybiera element największy. Jeśli chcemy takiej funkcji użyć dla tablicy **int**ów, dla tablicy liczb typu **double**, a potem dla tablicy obiektów klasy **Osoba** (zakładając, że jest w niej zdefiniowana relacja większości), to przyjdzie nam pisać identyczną w zasadzie funkcję trzy razy: typ parametru musi być jawnie zadeklarowany w definicji funkcji, więc nie można użyć tej samej funkcji dla tablicy osób i tablicy liczb. Jeśli okaże się, że potrzebujemy takiej funkcji również dla tablic obiektów klasy **Zwierze**, to trzeba będzie dodać odpowiednią funkcję i zrekompilować moduł definiujący te funkcje.

I właśnie w takich sytuacjach przychodzą nam z pomocą szablony. Tworzymy **wzorzec** (czyli **szablon**), według którego kompilator sam utworzy tyle wersji danej funkcji, ile będzie trzeba; wszystkie będą w zasadzie takie same, ale różnić się będą typami parametrów, wartości zwracanej i/lub zmiennych lokalnych definiowanych wewnątrz funkcji. Można będzie te funkcje wygenerować nawet dla typów, które w chwili pisania szablonu jeszcze w ogóle nie istniały!

Składnię szablonów funkcji omówimy na następującym przykładzie:

P86: ***tmplt.cpp*** Szablon funkcji

```
#include <iostream>
#include <typeinfo>
using namespace std;

template <class T1, typename T2>                                    ①
int howmany(const T1* arr, T2 mn, T2 mx, int size) {                ②
    int count = 0;
    for (int i = 0; i < size; ++i)
        if (arr[i] >= mn && arr[i] <= mx) ++count;

    // test
    cout << "T1=" << typeid(T1).name() << " "                       ③
         << "T2=" << typeid(T2).name() << " ";

    return count;
}

int main() {
    double mnd = 0, mxd = 10;
    int    mni = 0, mxi = 10;
    double tabd[] = {-2, -1, 2, 5, 7, 11};
    int    tabi[] = {-2, -1, 2, 5, 7, 11};

    int ii = howmany(tabi,mni,mxi,6);                               ④
    cout << "res=" << ii << endl;

    int id = howmany(tabi,mnd,mxd,6);                               ⑤
    cout << "res=" << id << endl;

    int di = howmany(tabd,mni,mxi,6);                               ⑥
    cout << "res=" << di << endl;

    int dd = howmany(tabd,mnd,mxd,6);                               ⑦
    cout << "res=" << dd << endl << endl;

    int xx = howmany<double,double>(tabd,mni,mxi,6);                ⑧
```

```
37      cout << "res=" << xx << endl;
38  }
```

Słowo **template** (①) jest tu słowem kluczowym informującym kompilator, że to, co dalej nastąpi, będzie definicją wzorca funkcji (lub klasy), a nie definicją konkretnej funkcji. Po tym słowie, w nawiasach kątowych, występuje lista parametrów formalnych, oddzielonych przecinkami, każdy poprzedzony słowem kluczowym **class** lub **typename** – w tym kontekście te dwa słowa są synonimami, ale zalecane jest to drugie. Parametry formalne wzorca mogą mieć dowolne nazwy, choć często używa się w nich dużej litery T — w naszym przykładzie użyliśmy nazw **T1** i **T2**. Teraz następuje definicja samego wzorca funkcji, w której używamy nazw parametrów formalnych (u nas **T1** i **T2**) w miejscach, gdzie wymagana jest nazwa *typu* – w szczególności którejś z tych nazw moglibyśmy użyć do określenia typu zwracanego. Można też używać wyrażeń określających typy pochodne, jak **T1&** czy **T2***.

Analizując kod wzorca widzimy, że jest to coś w rodzaju definicji funkcji **howmany** (②), która pobiera wskaźnik do tablicy elementów typu **T1** i zwraca ilość takich elementów tej tablicy, które są większe od mn a mniejsze od mx. Z kolei mn i mx są typu **T2**. [Instrukcja ③ nie jest tu potrzebna, ale została dodana, aby podczas wykonania wypisywał się aktualny typ skojarzony z **T1** i **T2**. Więcej na temat operatora **typeid** powiemy w rozdz. 25 na temat RTTI, str. 553.] Oczywiście, kompilator nie może skompilować wzorca jako funkcji, choćby dlatego, że nie ma typów o nazwach **T1** czy **T2**. Zapamięta jednak, że taki wzorzec istnieje.

Jak możemy takiego wzorca użyć? W linii ④ wywołujemy funkcję **howmany** podając jako argumenty tablicę **int**ów oraz zmienne typu **int** jako mn i mx. Kompilator zauważy, że funkcji **howmany** w ogóle nie ma, ale jest szablon o tej nazwie. Spróbuje zatem dopasować we wzorcu typy **T1** i **T2** tak, aby pasowały do typów argumentów wywołania. Zauważy, że jeśli **T1** utożsamić z **int** i **T2** również z **int**, to wywołanie będzie dokładnie pasować do sygnatury szablonu. Dokona zatem podstawienia: **T1** zostanie wszędzie w ciele wzorca zastąpione przez **int** i to samo dla **T2**. Nazywa się to **konkretyzacją** (ang. *concretization*) wzorca. Otrzymana w ten sposób funkcja zostanie skompilowana (i użyta w czasie wykonania); będzie ona miała postać

```
int howmany(int* arr, int mn, int mx, int size) {
    //
    // ...
    //
}
```

Gdyby w dalszej części programu znów pojawiło się wywołanie funkcji **howmany** z argumentami tego samego typu, to żadna konkretyzacja wzorca już *nie zajdzie* — teraz odpowiednia funkcja już będzie istnieć.

Przejdźmy teraz do linii ⑤. Tu wywołanie *nie* pasuje dokładnie do już istniejącej wersji funkcji, bo argumenty drugi i trzeci są tym razem typu **double**. Konkretyzacja zatem zajdzie jeszcze raz: teraz dokładne dopasowanie otrzymamy po utożsamieniu **T1**→**int** oraz **T2**→**double**. A zatem kompilator wygeneruje następne przeciążenie funkcji **howmany**, tym razem w postaci

```
int howmany(int* arr, double mn, double mx, int size) {
    //
    // ...
    //
}
```

Podobna sytuacja zajdzie przy wywołaniach z linii ⑥ i ⑦: w pierwszym przypadku dopasowanie będzie odpowiadać utożsamieniu **T1**→**double** oraz **T2**→**int**, natomiast w drugim **T1**→**double** oraz **T2**→**double**.

Zauważmy jeszcze składnię wywołania z linii ⑧. Tu jawnie (nawiasy kątowe za nazwą funkcji) zażądaliśmy podstawienia za pierwszy parametr formalny wzorca (**T1**) typu **double** i za drugi (**T1**) również typu **double**. Taka też wersja funkcji (w tym przypadku już utworzona wcześniej) zostanie użyta, mimo, że argumenty drugi i trzeci są typu **int** (oczywiście nic złego się nie stanie, bo konwersja **int**→**double** jest konwersją standardową i nie powoduje żadnej straty informacji).

Wynik programu

```
T1=i T2=i res=3
T1=i T2=d res=3
T1=d T2=i res=3
T1=d T2=d res=3

T1=d T2=d res=3
```

pokazuje, że istotnie utworzone zostały cztery przeciążone wersje funkcji **howmany**. Tym, że typ **int** został nazwany **i**, a **double** to **d** nie należy się przejmować — są to zależne od kompilatora wewnętrzne nazwy (kody) typów.

Słowo kluczowe **class** na liście parametrów szablonu może sugerować, że **T1** musi odpowiadać typowi zdefiniowanemu za pomocą klasy. Tak jednak nie jest: jak widzieliśmy z tego prostego przykładu, wartościami odpowiadającymi tym parametrom mogą być dowolne typy, również wbudowane. Dlatego bardziej chyba czytelne jest używanie w tym miejscu słowa kluczowego **typename**.

Przypatrzmy się teraz następującemu programowi:

P87: ***tmpl.cpp*** Szablony funkcji

```
1  #include <iostream>
2  #include <typeinfo>
3  using namespace std;
4
5  template <typename T>                                    ①
6  T larger(T k1, T k2) {
7      cout << "T=" << typeid(T).name() << " ";             ②
8      return k1 < k2 ? k2 : k1;
9  }
10
```

```
11 double larger(double k1, double k2) {                    ③
12     cout << "Spec. double ";
13     return k1 < k2 ? k2 : k1;
14 }
15
16 template<>                                                ④
17 short larger<short>(short k1, short k2) {
18     cout << "Spec. short ";
19     return k1 < k2 ? k2 : k1;
20 }
21
22 template<>
23 long larger<long>(long k1, long k2) = delete;             ⑤
24
25 int main() {
26     short s1 = 4, s2 = 5;
27
28     cout << larger(1.5,2.5)     << endl;                 ⑥
29     cout << larger(111,222)     << endl;                 ⑦
30     cout << larger('a','d')     << endl;                 ⑧
31     cout << larger<int>(s1,s2) << endl;                  ⑨
32     cout << larger(30L,50L)     << endl;                 ⑩
33 }
```

Definiujemy tu szablon funkcji **larger** (①). Wzorzec zależy od jednego tylko parametru **T**. Funkcja opisywana wzorcem pobiera dwa argumenty tego samego typu, oznaczonego jako **T**, i zwraca przez wartość rezultat też typu **T**. Sama funkcja jest bardzo prosta: wartością zwracaną jest wartość większego z argumentów. Jak poprzednio, dodaliśmy linię (②) która wypisuje informację o aktualnym typie skojarzonym z **T** (za pomocą operatora **typeid** dostępnego po dołączeniu pliku nagłówkowego ***typeinfo*** — patrz rozdz. 25, str. 553).

Zwróćmy teraz uwagę na funkcję (*nie* wzorzec) **larger** typu **double** (③). Jej sygnatura dokładnie odpowiada szablonowi po podstawieniu **T**→**double**.

Podobnie dostarczamy też specyficznej wersji funkcji **larger** dla typu **short** (④). Tym razem jednak jawnie poinformowaliśmy kompilator, że to co robimy jest konkretyzacją wzorca dla pewnego konkretnego typu (w tym przypadku **short**). Zauważmy składnię — puste nawiasy kątowe po słowie **template** i jawne wskazanie typu parametru wzorca przy nazwie funkcji. Ta forma jest bardziej wskazana, bo pozwala kompilatorowi sprawdzić czy to co robimy rzeczywiście jest poprawną składniowo konkretyzacją wzorca.

W końcu linia ⑤ deklaruje konkretyzację naszego wzorca dla typu long jako nieistniejącą (**=delete** zamiast ciała funkcji — jest to konstrukcja wprowadzona w standardzie C++11).

Program się nie kompiluje:

```
tmpl.cpp: In function `int main()':
```

```
tmpl.cpp:32:27: error: use of deleted function
        'T larger(T, T) [with T = long int]'
```

Przyczyna leży w ostatniej linii (⑩). Funkcje usunięte (**deleted**) *są* brane pod uwagę, kiedy wybierana jest najlepsza funkcja kandydująca; tu będzie nią funkcja powstała z konkretyzacji szablonu po podstawieniu **T**→**long**, które daje dopasowanie dokładne. Wtedy dopiero kompilator „zda sobie sprawę", że ta funkcja jest usunięta — żadna inna kandydatura nie będzie już rozważana i kompilacja zakończy sie niepowodzeniem.

Po wykomentowaniu ostatniej linii kompilacja przebiega pomyślnie i wykonanie daje:

```
Spec. double 2.5
T=i 222
T=c d
T=i 5
```

Zwróćmy uwagę na niektóre aspekty tego programu ilustrujące ogólne zasady używania szablonów funkcji:

- W linii ⑥ wywołujemy funkcję **larger** z dwoma argumentami typu **double**. Odpowiada to dokładnie funkcji wygenerowanej z szablonu po podstawieniu **T** → **double**. Z drugiej strony, funkcję o odpowiedniej sygnaturze zdefiniowaliśmy też jawnie (③). Z wydruku widzimy, że jeśli możliwe jest użycie funkcji wygenerowanej z szablonu lub zdefiniowanej jawnie, to wybierana jest ta ostatnia.

- W linii ⑦ wywołujemy funkcję **larger** z dwoma argumentami typu **int**. Odpowiada to *dokładnie* funkcji wygenerowanej z szablonu po podstawieniu **T** → **int**. Odpowiada też, po zastosowaniu konwersji standardowej **int** → **double** dla argumentów, jawnie zdefiniowanej funkcji **larger** dla typu **double**. Wydruk wskazuje, że tym razem wybrana została wersja z szablonu, ponieważ otrzymujemy wtedy funkcję *dokładnie* odpowiadającą wywołaniu, podczas gdy istniejąca wersja z typem **double** wymagałaby konwersji argumentów.

- W linii ⑧ wywołujemy funkcję **larger** z dwoma argumentami typu **char**. Takie wywołanie mogłoby być „obsłużone" przez istniejącą już wersję z **int**ami, ale kompilator z niej nie skorzysta, bo z szablonu może wygenerować wersję która odpowiada wywołaniu bez żadnych konwersji.

- W linii ⑨ wstawiliśmy, w nawiasach kątowych, argument **int** dla szablonu. Taki zapis oznacza, że jawnie życzymy sobie wygenerowania wersji funkcji **larger** na podstawie szablonu o tej nazwie i po podstawieniu **T**→**int** (w naszym przypadku ta wersja już zresztą istnieje). Oczywiście, poprzez podstawienie **T**→**short** można wyprodukować z szablonu lepsze dopasowanie, ale tym razem sami jawnie określiliśmy której konkretyzacji użyć.

Nazwy typów pojawiające się w wydruku (**i**, **c**) są wewnętrznymi nazwami typów danych i mogą zależeć od implementacji (nie ma się co nimi przejmować, ważne jest, że możemy porównywać typy, a nie to, jak się nazywają).

Przypatrzmy się jeszcze definicji szablonu. W jego treści dla zmiennych k1 i k2 użyte jest porównanie za pomocą operatora '<'. Jeśli te zmienne są typu numerycznego, to oczywiście nie ma kłopotu: dla typów numerycznych operacja porównywania zdefiniowana jest „sama z siebie". Dla innych typów, zdefiniowanych za pomocą klas przez użytkownika, takie porównanie może nie być określone — użycie takiego typu do konkretyzacji szablonu spowodowałoby błąd kompilacji (choć, jak się przekonamy, nawet dla własnych typów *można* określić działanie operatora porównania).

Jako przykład na bardziej realistyczne zastosowanie szablonów, w poniższym programie definiujemy wzorzec **minmaxmed** funkcji pobierającej tablicę elementów pewnego typu i obliczającą element maksymalny, minimalny i medianę (taką wartość, od której połowa elementów tablicy jest mniejsza, a połowa większa). Tak jak poprzednio, typ elementów musi dopuszczać porównywanie. Minimum i maksimum zwracane jest poprzez argumenty przekazane przez referencję, a mediana jako wartość zwracana funkcji. Dodatkowo, również za pomocą szablonów definiujemy funkcje **pisztab** (do wypisywnia elementów tablicy), **inssort** (do porządkowania tablicy metodą sortowania przez wstawianie — *insertion sort*) oraz funkcję **test**, która wywołuje poprzednie funkcje.

P88: ***sorttempl.cpp*** Szablony funkcji

```
#include <iostream>
#include <cstring>    // memcpy
using namespace std;

template<typename T>
void pisztab(ostream&,const T[],int);

template<typename T>
void inssort(T[],int);

template<typename T>
double minmaxmed(const T[],int,T&,T&);

template<typename T>
void test(T[],int);

int main() {
    cout << "\n===tablica int===" << endl;
    int tabi[] = {9,7,2,6,6,2,7,9,2,9,5,2};
    test(tabi,sizeof(tabi)/sizeof(int));

    cout << "\n===tablica double===" << endl;
    double tabd[] = {9.5,2.5,6,7.5,9,2,5,2.5};
    test(tabd,sizeof(tabd)/sizeof(double));

    cout << "\n===tablica unsigned===" << endl;
```

```
27      unsigned tabu[] = {23,32,12,76,21,45,20,67};
28      test(tabu,sizeof(tabu)/sizeof(unsigned));
29  }
30
31  template<typename T>
32  void test(T tab[],int size) {
33      T min, max;
34
35      double mediana = minmaxmed(tab,size,min,max);
36
37      cout << "min = " << min << ", max = " << max
38           << ", mediana = "  <<  mediana   << endl;
39
40      cout << "Tablica  oryginalna: ";
41      pisztab(cout, tab, size);
42
43      inssort(tab, size);
44
45      cout << "Tablica posortowana: ";
46      pisztab(cout, tab, size);
47  }
48
49  template<typename T>
50  void pisztab(ostream& str, const T t[], int size) {
51      str << "[ ";
52      for (int i = 0; i < size; ++i) str << t[i] << " ";
53      str << "]" << endl;
54  }
55
56  template<typename T>
57  void inssort(T a[], int size) {
58      int i, indmin = 0;              // wartownik
59      for (i = 1; i < size; ++i)
60          if (a[i] < a[indmin]) indmin = i;
61      if (indmin != 0) {
62          T p = a[0];
63          a[0] = a[indmin];
64          a[indmin] = p;
65      }
66
67      for (i = 2; i < size; ++i) {  // sortowanie
68          int j = i;
69          T v = a[i];
70          while (v < a[j-1]) {
71              a[j] = a[j-1];
72              j--;
```

```
73          }
74          if (i != j ) a[j] = v;
75      }
76 }
77
78 template<typename T>
79 double minmaxmed(const T t[], int size, T& min, T& max) {
80      T* tab = new T[size];
81      memcpy(tab,t,size*sizeof(T));
82
83      inssort(tab, size);
84
85      min             = tab[0];
86      max             = tab[size-1];
87      double mediana = size%2 == 0 ?
88                         0.5*(tab[size/2] + tab[size/2-1])
89                       : tab[size/2];
90
91      delete [] tab;
92      return mediana;
93 }
```

Funkcja **minmaxmed** najpierw tworzy kopię przekazanej tablicy (aby nie zmodyfikować oryginału), sortuje tę kopię, znajduje wtedy łatwo szukane elementy, po czym usuwa roboczą tablicę. Z wydruku

```
===tablica int===
min = 2, max = 9, mediana = 6
Tablica  oryginalna: [ 9 7 2 6 6 2 7 9 2 9 5 2 ]
Tablica posortowana: [ 2 2 2 2 5 6 6 7 7 9 9 9 ]

===tablica double===
min = 2, max = 9.5, mediana = 5.5
Tablica  oryginalna: [ 9.5 2.5 6 7.5 9 2 5 2.5 ]
Tablica posortowana: [ 2 2.5 2.5 5 6 7.5 9 9.5 ]

===tablica unsigned===
min = 12, max = 76, mediana = 27.5
Tablica  oryginalna: [ 23 32 12 76 21 45 20 67 ]
Tablica posortowana: [ 12 20 21 23 32 45 67 76 ]
```

widzimy, że dzięki szablonom poradziliśmy sobie z tablicami różnych typów. Zauważmy, że ponieważ do kopiowania tablic użyliśmy, choć oczywiście nie musieliśmy, funkcji **memcpy**, wszystko będzie działać tylko dla typów, dla których kopiowanie obiektów „bit po bicie” daje właściwy rezultat! Zwróćmy uwagę, że wywołania funkcji określonych przez szablony następują tu również wewnątrz innych szablonów: na przykład szablonowa funkcja **test** wywołuje szablonowe funkcje **minmaxmed**, **inssort**

i **pisztab**.

W tym programie szablony, tak jak zwykłe funkcje, najpierw zostały zadeklarowane, a dopiero później zdefiniowane. Zarówno w deklaracji, jak i w definicji trzeba oczywiście użyć słowa **template**.

Zilustrujmy jeszcze jeden „trik" składniowy, pozwalający na „zrzucenie" na kompilator pełnej odpowiedzialności za wydedukowanie typu zwracanego funkcji. Od wersji standardu C++14, możemy typ zwracany funkcji zadeklarować jako **decltype(auto)** — kompilator wydedukuje wtedy typ zwracany analizując kod ciała funkcji i patrząc na wszystkie instrukcje **return** — wszystkie one, oczywiście, muszą zwracać wartości dokładnie tego samego typu. Spójrzmy na prosty przykład

P89: ***autoret.cpp*** Dedukcja typu zwracanego

```
#include <iostream>
#include <typeinfo>
using namespace std;

  // typ zwracany może być wydedukowany, gdy T i U znane
template<typename T, typename U>
decltype(auto) mul(T x, U y) {
    return x*y;
}

int main() {
    auto r1 = mul(2.0,7); // double*int -> double
    std::cout << r1 << " :: " << typeid(r1).name() << "\n";
    auto r2 = mul(2,7L);  //   int*long -> long
    std::cout << r2 << " :: " << typeid(r2).name() << "\n";
}
```

Z wydruku

```
14 :: d
14 :: l
```

widzimy, że

- w pierwszym wywołaniu typy argumentów są **T** → **double** i **U** → **int**; wydedukowanym typem zwracanym jest zatem **double**;
- w drugim wywołaniu typy argumentów są **T** → **int** i **U** → **long**; wydedukowanym typem zwracanym jest zatem **long**.

— 12 —

Zarządzanie pamięcią

Możliwość samodzielnego zarządzania pamięcią przez programistę jest z jednej strony siłą C/C++, z drugiej zaś komplikuje program i powoduje, że pisanie dużych programów wymaga więcej staranności i uwagi.

Pamiętać trzeba, że w C/C++ *nie ma* „odśmiecacza" pamięci (ang. *garbage collector*) znanego z Javy, Lispu, Smalltalk'a, Pythona i wielu innych języków obiektowych. Odpowiedzialność za zwalnianie pamięci zajmowanej przez niepotrzebne już obiekty zaalokowane w pamięci wolnej (stercie) spoczywa na programiście.

PODROZDZIAŁY:

12.1 Wstęp

W C/C++, tak jak i w wielu innych językach, dane mogą być zapisywane w kilku odrębnych obszarach pamięci. Szczegóły zależą od implementacji, ale generalnie najważniejszymi takimi obszarami są stos i sterta.

Na **stosie** zapisywane są zmienne lokalne definiowane w programie (bez użycia operatora **new**). Gdy sterowanie wchodzi do funkcji (lub, ogólniej, bloku) zmienne utworzone w tej funkcji są umieszczane na stosie. Po wyjściu sterowania z tego bloku, stos jest „zwijany": utworzone w nim zmienne są usuwane i stos powraca do stanu sprzed wejścia do funkcji/bloku. Zarządzanie stosem to bardzo delikatna sprawa i normalnie programista się tym nie zajmuje: „ręczna" ingerencja w strukturę stosu byłaby bardzo niebezpieczna. Nie ma zresztą żadnej potrzeby takiej ingerencji.

Drugi podstawowy obszar pamięci to **sterta** (ang. *heap*) lub, inaczej, obszar pamięci wolnej. Programista może alokować w tym obszarze miejsce na swoje dane, umieszczać je tam, następnie je zmieniać, odczytywać itd. W C/C++ dane te będą istnieć do końca działania programu i obszar pamięci wolnej, gdzie są zapisane, będzie zatem do końca programu oznaczony jako zajęty. Może to prowadzić do przepełnienia pamięci i przedwczesnego przerwania programu, albo jego znaczego spowolnienia (np. na skutek „swapowania").

Jak wspomnieliśmy, nie ma w C/C++ automatycznego „odśmiecania" pamięci

(choć istnieją implementacje posiadające tę cechę). Istnieją za to sposoby pozwalające „ręcznie” zwalniać zaalokowane wcześniej obszary pamięci wolnej (co w Javie jest praktycznie niemożliwe, ale też i mało potrzebne).

12.2 Przydzielanie pamięci — operator *new*

Pamięć na stercie można przydzielić (zaalokować), czyli zarezerwować po to, by zapisać tam jakieś dane, za pomocą operatora **new**. Operator ten występuje tylko w C++, choć alokować pamięć można i w C; jak to zrobić, powiemy w dalszej części tego rozdziału.

Operator **new** wymaga wskazania typu danej lub danych, które mają być w przydzielonej pamięci zapisane oraz informacji o ilości tych danych, czyli o wielkości obszaru pamięci, jaki ma być zarezerwowany.

W najprostszej formie operatora **new** można użyć do utworzenia w pamięci wolnej pojedynczej danej (zmiennej) pewnego typu. Składnia jest następująca:

```
int* pi = new int;
```

i oznacza polecenie utworzenia w pamięci wolnej zmiennej typu **int**, co wiąże się z zarezerwowaniem obszaru pamięci o odpowiednim rozmiarze i zaznaczeniu jej jako zajętej. Operator **new** znajduje i rezerwuje tę pamięć i zwraca adres początku zaalokowanego obszaru pamięci. Inicjalizuje też nowo utworzoną zmienną (w wypadku **int**a inicjalizacja oznacza „nic nie rób”, ale dla innych typów jest to nietrywialna operacja związana z wywołaniem konstruktora). Zwrócony adres możemy (i zwykle powinniśmy) zapamiętać: oczywiście w zmiennej odpowiedniego typu wskaźnikowego, tak jak w powyższym przykładzie (bo wartość zwracana przez **new** jest przecież *adresem*).

Zauważmy, że po tej instrukcji pi jest zmienną wskaźnikową *lokalną*, to znaczy zmienna ta jest umieszczana na stosie. W szczególności, zostanie ona usunięta po wyjściu sterowania z bloku, w którym była zdefiniowana!

Usunięcie wskaźnika do zaalokowanej pamięci nie zwalnia tej pamięci.
Pozostaje ona w dalszym ciągu „zajęta”.

Jeśli nie jesteśmy ostrożni, to w ten sposób stracimy możliwość odwołania się do nowo utworzonej na stercie anonimowej zmiennej; dostępna jest ona *jedynie* poprzez adres zawarty w zmiennej pi. Jeśli coś takiego się zdarzy, to następuje tzw. wyciek pamięci (ang. *memory leakage*) — zmienna na stercie istnieje, ale jest bezużyteczna, bo zgubiliśmy jej adres i nie wiemy gdzie ta zmienna jest!

W miejsce typu **int**, jak w powyższym przykładzie, może oczywiście wystąpić dowolny typ; również typ zdefiniowany przez użytkownika, a więc klasa. Gdyby była to klasa, np. o nazwie **Klasa**, to przydział pamięci na jeden obiekt tej klasy wyglądałby tak

```
Klasa* pk = new Klasa;
```

Jeśli klasa ta miałaby jakieś konstruktory wymagające danych, to moglibyśmy użyć składni (podobnej do tej z Javy)

```
Klasa* pk = new Klasa(12,8);
```

Co ciekawe, tej samej składni możemy użyć do utworzenia na stercie zmiennych typów podstawowych, np.

```
int* pi = new int(18);
```

utworzy na stercie zmienną typu **int** i zainicjuje ją wartością 18, zupełnie jakby **int** było nazwą klasy z jednoargumentowym konstruktorem. Ta cecha języka jest zamierzona: autor (Bjarne Stroustrup) dążył do tego, aby typy wbudowane i definiowane przez użytkownika były, jak to tylko możliwe, traktowane na równych prawach i podlegały tym samym regułom składniowym. W opisany wyżej sposób można też utworzyć dynamicznie *stałą*:

```
const int* stala = new const int(1);
```

Nie byłoby to możliwe bez podania w nawiasie inicjatora, gdyż, jak pamiętamy, stała musi być zainicjowana już w czasie tworzenia.

Jeśli niewygodnie nam operować na zmiennej bez nazwy, to można po jej utworzeniu nadać jej nazwę poprzez zdefiniowanie do niej odniesienia (referencji); na przykład w poniższym programiku rd jest referencją do (czyli inną nazwą) zmiennej anonimowej na stercie:

P90: *ref.cpp* Referencja do zmiennej na stercie

```
#include <iostream>
using namespace std;

int main() {
    double *pd = new double(4.5),
           &rd = *pd;

    cout << "*pd = " << *pd << endl;
    cout << " rd = " <<  rd << endl;
    *pd = 1.5;
    cout << "*pd = " << *pd << endl;
    cout << " rd = " <<  rd << endl;
    delete pd;
}
```

Do tej samej zmiennej na stercie możemy się zatem odnosić poprzez dereferencję wskaźnika, `*pd`, jak i referencję `rd` — zmieniamy wartość tej zmiennej poprzez `*pd`, a następnie drukujemy tę wartość odnosząc się do tej samej zmiennej poprzez obie nazwy:

```
*pd = 4.5
 rd = 4.5
*pd = 1.5
 rd = 1.5
```

Można również alokować pamięć na więcej niż jeden obiekt dowolnego typu. Składnia jest wtedy taka:

```
int* pi = new int[wym];
```

gdzie tym razem, po określeniu typu (w tym przypadku **int**), podajemy w nawiasach kwadratowych liczbę elementów danego typu, na które alokujemy pamięć. Wyrażenie wym powinno mieć typ **size_t** — jest to alias nadany za pomocą **typedef** pewnemu typowi całkowitemu bez znaku (zwykle **unsigned long**). Kolejne elementy w utworzonym obszarze pamięci będą zajmować kolejne fragmenty w *ciągłym* obszarze pamięci (jak dla tablic). Fundamentalne znaczenie ma fakt, że wyrażenie wym może być dowolnym wyrażeniem o dodatniej wartości całkowitej. Wymiar ten może zatem być wczytany, lub w jakiś sposób wyliczony, w trakcie działania programu — nie musi być znany już w czasie kompilacji czy ładowania programu, tak jak miało to miejsce dla zwykłych (czyli *statycznych*) tablic. Dlatego cały ten proces nazywamy *dynamicznym* przydziałem pamięci, a tablice tak utworzone **tablicami dynamicznymi**.

Jeśli na przykład aktualną wartością wym jest 40, to w powyższym przykładzie zarezerwowane zostanie 160 bajtów (4×40) i adres początku tego obszaru pamięci zostanie zwrócony przez **new** i zapamiętany w zmiennej wskaźnikowej pi.
Tak przydzielona pamięć *jest* inicjowana, choć dla niestatycznych zmiennych typów prostych inicjalizacja oznacza „nic nie rób” (inaczej będzie dla tablic obiektów klas — wtedy elementy tablicy są tworzone za pomocą konstruktora domyślnego). Jak widzieliśmy, zmiennej pi można teraz używać tak jak nazwy tablicy, zgodnie z odpowiedniością pomiędzy wskaźnikami i tablicami. Moglibyśmy na przykład nadać sensowne wartości danym w nowo przydzielonym obszarze pamięci za pomocą pętli

```
for (int i = 0; i < wym; ++i) pi[i] = 2*i;
```

Nie należy mylić nawiasów okrągłych i kwadratowych w obu formach użycia operatora **new**: nawiasy okrągłe zawierają dane potrzebne do inicjalizacji tworzonej *pojedynczej* zmiennej; nawiasy kwadratowe zawierają wyrażenie o wartości całkowitej mówiące o liczbie tworzonych elementów.

Zauważmy też, że tworzone tablice dynamiczne można, jak statyczne, inicjować za pomocą inicjatorów „klamrowych”:

```
int* p = new int[5]{1, 2, 3, 4, 5};
```

Można również alokować dynamicznie pamięć na tablice wielowymiarowe, ale są one wtedy tylko „półdynamiczne”. Oznacza to, że tylko jeden, a mianowicie pierwszy, wymiar może *nie być* stałą kompilacji. Rozpatrzmy przykład:

P91: ***poldyn.cpp*** Wielowymiarowe tablice „półdynamiczne”

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
```

```
5     const int DIM = 3;
6     cout << "Podaj pierwszy wymiar: ";
7     int size;
8     cin >> size;
9     int (*t)[DIM] = new int[size][DIM];          ①
10
11    for (int i = 0; i < size; ++i)
12        for (int j = 0; j < DIM; ++j)
13            t[i][j] = 10*i + j;
14
15    int* p = reinterpret_cast<int*>(t);          ②
16
17    for (int i = 0; i < DIM*size; ++i)
18        cout << p[i] << " ";
19    cout << endl;
20
21    cout << "t[0]         : " << t[0] << endl;   ③
22    cout << "t[1]         : " << t[1] << endl;
23    cout << "sizeof(t[0]): " << sizeof(t[0]) << endl; ④
24 }
```

W linii ① alokujemy pamięć na tablicę size×DIM, gdzie size nie jest znane z góry, gdyż jest wczytywane z klawiatury w trakcie wykonania. Natomiast drugi (i ewentualne następne) wymiar musi być stałą kompilacji. Zauważmy typ zmiennej t: jest to wskaźnik do trzyelementowej tablicy **int**'ów. Zatem obiektem wskazywanym jest tu nie „coś" typu **int**, ale cała tablica **int**'ów, w tym przypadku o wymiarze 3 (bo tyle wynosi DIM). Zatem t[0] jest taką tablicą i ma rozmiar 12 bajtów (3×4). Odpowiada pierwszemu wierszowi tablicy. Zatem t[1] też jest taką tablicą, odpowiadającą drugiemu wierszowi i powinno leżeć w pamięci o 12 bajtów dalej. Że tak jest rzeczywiście, przekonuje nas wydruk tego programu (0x88d01c−0x88d010=C w układzie szesnastkowym, czyli 12 w układzie dziesiętnym; konkretne adresy mogą być oczywiście inne, ale różnica powinna być właśnie taka). Zauważmy, że drukując t[0] (③) drukujemy tablicę, a ta jest konwertowana do wskaźnika wskazującego na początek tablicy — dlatego otrzymujemy adresy.

```
Podaj pierwszy wymiar: 5
0 1 2 10 11 12 20 21 22 30 31 32 40 41 42
t[0]         : 0x88d010
t[1]         : 0x88d01c
sizeof(t[0]): 12
```

W linii ② tworzymy zmienną wskaźnikową typu **int*** i wpisujemy tam adres początku całej tablicy t; o operatorze **reinterpret_cast** będziemy jeszcze mówić, na razie powiedzmy tylko, że jest tu konieczny ze względu na kontrolę typów: typem t nie jest bowiem **int*** (równie dobrze mogliśmy użyć tradycyjnej formy rzutowania typów '(**int***)'). Traktując następnie p jak jednowymiarową tablicę liczb całkowitych drukujemy kolejne wartości elementów tablicy. Widać, że są one zgodne z tym, co

wpisaliśmy poprzedzającej pętli i że rzeczywiście są ułożone w pamięci wierszami (co nie jest sprawą obojętną, na przykład dla wydajności operacji na macierzach — w Fortranie dane ułożone byłyby kolumnami).

Przydział pamięci może się nie powieść, na przykład jeśli zażądaliśmy zarezerwowania zbyt dużej jej ilości. Jak się przekonamy, w takich sytuacjach w języku C zwracany jest wtedy wskaźnik pusty (NULL). W C++ natomiast generowany jest wtedy wyjątek typu **bad_alloc** (z nagłówka *new*) który możemy przechwycić i obsłużyć zapobiegając załamaniu programu. O obsłudze wyjątków powiemy więcej w rozdz. 22 (str. 491), ale poniższy przykład powinien być zrozumiały przynajmniej dla tych, którzy uczyli się już Javy lub Pythona:

P92: *allo.cpp* Błąd przy alokacji pamięci

```
#include <iostream>
#include <new>
#include <iomanip>
using namespace std;

int main() {
    const size_t mega = 1024*1024, step = 200*mega;

    for (size_t size = step; ;size += step) {
        try {
            char* buf = new char[size];
            delete [] buf;
        }
        catch(bad_alloc) {
            cout << "NIE UDALO SIE: "  << setw(4)
                 << size/mega << " MB" << endl;
            return 1;
        }
        cout << "    udalo sie: "  << setw(4)
             << size/mega << " MB" << endl;
    }
}
```

W nieskończonej pętli alokujemy i natychmiast zwalniamy za pomocą operatora **delete** (patrz następny podrozdział) coraz większy obszar pamięci. W pewnym momencie żądamy tej pamięci za dużo. Zadanie nie może być wykonane, więc zgłaszany jest wyjątek, który przechwytujemy (fraza **catch**), drukujemy komunikat i kończymy program poprzez wywołanie **return** w funkcji **main**. Program ten wygenerował następujący wydruk:

```
    udalo sie:  200 MB
    udalo sie:  400 MB
    udalo sie:  600 MB
    udalo sie:  800 MB
```

```
    udalo sie: 1000 MB
    udalo sie: 1200 MB
    udalo sie: 1400 MB
    udalo sie: 1600 MB
    udalo sie: 1800 MB
    udalo sie: 2000 MB
    udalo sie: 2200 MB
    udalo sie: 2400 MB
    udalo sie: 2600 MB
    udalo sie: 2800 MB
NIE UDALO SIE: 3000 MB
```

Wydruk nie oznacza, że komputer ma rzeczywiście 3 GB pamięci — odliczony jest obszar zajęty a doliczony obszar wymiany (ang. *swap*).

Zauważmy jeszcze, że alokowanie pamięci na jeden egzemplarz obiektu

```
int* pi = new int;
```

nie jest równoważne alokowaniu pamięci na tablicę jednoelementową

```
int* pi = new int[1];
```

Przydział pamięci na tablice implementowany jest inaczej niż przydział pamięci na pojedyncze obiekty, te dwie instrukcje mają więc inny skutek. Różnica przejawia się między innymi podczas zwalniania tak przydzielonej pamięci (patrz następny podrozdział).

12.3 Zwalnianie pamięci — operator *delete*

Zarezerwowaną za pomocą **new** pamięć należy zwolnić, gdy nie będzie już potrzebna. Wykonuje się to za pomocą operatora **delete**. Jeśli zaalokowaliśmy, za pomocą **new**, pamięć na pojedynczy obiekt, to składnia jest następująca:

```
delete pi;
```

Argumentem operatora (w przykładzie powyżej jest to wartość zmiennej wskaźnikowej pi) musi być wyrażenie, którego wartością jest dokładnie ten sam adres, który został nam „przysłany” przez operator **new**.

Operatora **delete** nie wolno zastosować do adresu który nie został uprzednio zwrócony przez operator **new**.

Natychmiast po wykonaniu tej instrukcji pamięć, którą do tej pory zajmował obiekt wskazywany przez pi, jest zwalniana i może być użyta przez system do zapisania innych danych. Z tego powodu, choć nie tylko z tego,

operatora **delete** nie wolno zastosować dwa razy do tego samego adresu.

Zwróćmy uwagę, że sama zmienna pi *nie* jest usuwana, ani nawet nie jest zmieniana jej wartość — to pamięć wskazywana przez tę zmienną jest zwalniana!

Inna jest składnia przy zwalnianiu pamięci przydzielonej na tablicę, czyli poprzez **new** z podaniem wymiaru w nawiasach kwadratowych. Aby tak zaalokowaną pamięć zwolnić, należy użyć operatora **delete[]**

```
delete [] pi;
```

Wartość wyrażenia, które jest argumentem **delete** (w powyższym przykładzie pi) musi być adresem zwróconym uprzednio przez operator **new** alokującego tablicę, a nie pojedynczy obiekt. W nawiasach kwadratowych nie podajemy żadnego wymiaru — musiał być podany podczas alokowania tej pamięci za pomocą **new** i jest zapamiętywany. Dlatego, jak wspominaliśmy, alokowanie pamięci na jeden egzemplarz obiektu jest czym innym niż alokowanie pamięci na tablicę jednoelementową.

Jeśli alokujemy pamięć na pojedynczy obiekt, to trzeba ją zwolnić za pomocą **delete**. Jeśli alokujemy pamięć na tablicę (nawet jednoelementową), to trzeba ją zwolnić za pomocą **delete[]**.

Zwalnianie pamięci przydzielonej przez „nietablicowe” **new** za pomocą „tablicowego” **delete** lub odwrotnie jest błędem niewychwytywanym przez kompilator, a powodującym nieprzewidywalne, ale raczej opłakane, skutki.

Operator **delete**, w obu formach — zwykłej i tablicowej, jest tak zdefiniowany, że jego użycie nie powoduje żadnego skutku, jeśli podany adres jest pusty (czyli **nullptr**, albo po prostu zero). Dlatego, aby uniknąć przypadkowego wielokrotnego zwalniania tego samego obszaru pamięci, co powoduje najczęściej katastrofalny błąd wykonania, można, po użyciu **delete**, w odpowiednią zmienna wskaźnikową wpisać adres pusty

```
1    int* pi = new int[40];
2    // ...
3    delete [] pi;
4    pi = 0;
5    //
6    delete [] pi; // teraz nieszkodliwe
```

W linii 3 zwalniamy pamięć przydzieloną w linii 1 i natychmiast zerujemy zmienną pi. Teraz, jeśli w dalszej części programu spróbujemy jeszcze raz zwolnić tę samą pamięć, nic złego się nie stanie.

Jako przykład rozważmy następujący problem. Chcemy napisać funkcję **minmaxmed** znajdującą element najmniejszy, największy i medianę z danych zawartych w tablicy (mediana to taka wartość, że połowa elementów jest od niej nie większa, i połowa jest od niej nie mniejsza). Choć istnieją szybsze metody znajdowania mediany, najprościej zrozumieć następującą. Najpierw sortujemy tablicę, czyli przestawiamy

jej elementy tak, aby uzyskać kolejność niemalejącą. Wtedy pierwszy element jest na pewno minimalnym, ostatni maksymalnym, a mediana środkowym lub średnią z dwóch środkowych, jeśli liczba elementów jest parzysta.

Problem w tym, że aby ten algorytm zrealizować, musimy poprzestawiać elementy tablicy, co być może nie jest dopuszczalne, bo będzie ona jeszcze potrzebna w takiej formie, w jakiej jest. Możemy zatem stworzyć tablicę roboczą, przekopiować tam naszą tablicę i szukać wyniku używając tylko tej tablicy roboczej. Realizuje to funkcja **minmaxmed** poniższego programu:

P93: *mediana.cpp* Alokowanie tablic

```
#include <iostream>
using namespace std;

void   pisztab(ostream&,const int[],size_t);
void   inssort(int[],size_t);
double minmaxmed(const int[],size_t,int&,int&);

int main() {
    int tab[] = {7,2,6,4,7,5}, min, max;                    ①
    size_t size = sizeof(tab)/sizeof(tab[0]);

    double mediana = minmaxmed(tab,size,min,max);

    cout << "min = " << min << ", max = " << max
         << ", mediana = "  <<  mediana   << endl;

    cout << "Tablica  oryginalna: ";
    pisztab(cout, tab, size);                               ②

    inssort(tab, size);                                     ③

    cout << "Tablica posortowana: ";
    pisztab(cout, tab, size);                               ④
}

void pisztab(ostream& str, const int t[], size_t size) {
    str << "[ ";
    for (size_t i = 0; i < size; ++i) str << t[i] << " ";
    str << "]" << endl;
}

void inssort(int a[], size_t siz) {
    size_t indmin = 0;
    for (size_t i = 1; i < siz; ++i)
        if (a[i] < a[indmin]) indmin = i;
    if (indmin != 0) {
```

```
37          int p = a[0];
38          a[0] = a[indmin];
39          a[indmin] = p;
40      }
41      for (size_t i = 2; i < siz; ++i) {
42          size_t j = i;
43          int v = a[i];
44          while (v < a[j-1]) { a[j] = a[j-1]; j--; }
45          if (i != j ) a[j] = v;
46      }
47  }
48
49  double minmaxmed(const int t[], size_t size,
50                   int& min, int& max) {
51      int* tab = new int[size];                              ⑤
52
53      // byłoby lepiej za pomocą memcpy...
54      for (size_t i = 0; i < size; ++i) tab[i] = t[i];       ⑥
55
56      inssort(tab, size);                                    ⑦
57
58      min = tab[0];
59      max = tab[size-1];
60
61      double mediana = size%2 == 0 ?
62                  0.5*(tab[size/2] + tab[size/2-1])
63                : tab[size/2];
64      delete [] tab;                                         ⑧
65      return mediana;
66  }
```

Funkcja **minmaxmed** otrzymuje tablicę (wskaźnik) t — odpowiadający jej parametr deklarujemy z modyfikatorem **const**, aby nie dopuścić do przypadkowego zniszczenia oryginalnej tablicy. Następnie (⑤) alokujemy pamięć na nową tablicę o takim rozmiarze jak t i przekopiowujemy tam wszystkie elementy (⑥). Dopiero tę tablicę sortujemy (⑦), znajdujemy rezultaty i wpisujemy je do min i max, które były przekazane przez referencje, a więc zmiana ich wartości będzie widoczna w funkcji wywołującej. Na końcu zwracamy przez wartość medianę.

Przed wyjściem z funkcji musimy zwolnić zarezerwowaną pamięć (⑧). Zmienna wskaźnikowa tab jest bowiem lokalna; po wyjściu z funkcji przestanie istnieć i nie będzie już żadnego sposobu, aby odnieść się do zaalokowanej tablicy, w szczególności usunąć ją. Pamięć, jaką zajmuje ta tablica robocza byłaby „zajęta" do końca programu, choć bezużyteczna. Zauważmy, że taki wyciek pamięci powstawałby *za każdym wywołaniem* funkcji **minmaxmed** i straty pamięci kumulowałyby się.

Użyta w programie funkcja **inssort** (③ i ⑦) realizuje znany, choć w ogólnym przypadku nie najszybszy, algorytm sortowania, tzw. *sortowanie przez wstawianie*

(ang. *insertion sort*), a konkretnie jego wersję z „wartownikiem". Algorytm ten jest niesłychanie efektywny dla tablic prawie posortowanych, a więc takich, do których uporządkowania wystarcza niewiele przestawień — sytuacje takie zdarzają się zaskakująco często w praktyce.

W programie głównym tworzymy tablicę (①) i wysyłamy ją do funkcji **minmaxmed**. Następnie drukujemy wyniki. W lini ② drukujemy zawartość tablicy, aby przekonać się, że istotnie nie uległa zmianie w funkcji **minmaxmed**. Tę tablicę następnie sortujemy (③) i drukujemy jeszcze raz (④), aby wygodniej było sprawdzić prawidłowość otrzymanych wyników

```
min = 2, max = 7, mediana = 5.5
Tablica  oryginalna: [ 7 2 6 4 7 5 ]
Tablica posortowana: [ 2 4 5 6 7 7 ]
```

Zauważmy, że funkcja **pisztab** drukująca zawartość tablicy napisana jest tak, że może pisać do dowolnego strumienia reprezentowanego obiektem klasy **ostream**. My posyłamy do funkcji obiekt cout, bo innego jeszcze nie znamy, ale dzięki temu zabiegowi ta sama funkcja mogłaby być użyta na przykład przy pisaniu do pliku (patrz rozdz. 16 na stronie 323).

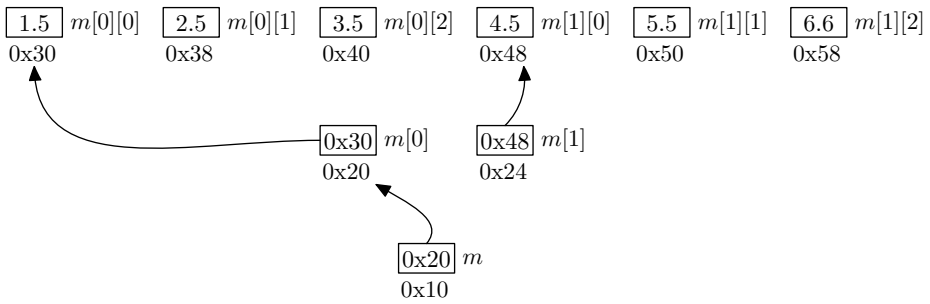
12.4 Dynamiczne tablice wielowymiarowe

Pokazaliśmy, jak można dynamicznie alokować pamięć na tablice jednowymiarowe lub wielowymiarowe, ale takie w których tylko jeden, mianowicie pierwszy, wymiar nie jest z góry znany. Często jednak zachodzi potrzeba dynamicznego tworzenia tablic wielowymiarowych, na przykład dwuwymiarowych, ale takich, w których wszystkie wymiary są określane dynamicznie w trakcie wykonania programu. Pokażemy teraz jeden ze sposobów, w jaki można to zadanie zrealizować.

Załóżmy, że chcemy zaalokować miejsce na tablicę liczb typu **double** o wymiarach 2×3. Chcielibyśmy odnosić się do tej tablicy za pomocą normalnej składni z użyciem indeksów w nawiasach kwadratowych. Na przykład m[1][2] powinno oznaczać element z wiersza drugiego i kolumny trzeciej — indeksy liczymy jak zwykle od zera — tablicy (macierzy) m.

Co oznacza m[1][2]? Wiemy, że jest to skrócony zapis wyrażenia '`*(m[1] + 2)`' (patrz rozdz. 5.3 na stronie 56). Zatem m[1] musi być wskaźnikiem typu **double***. A zatem m jest tablicą wskaźników, której kolejne elementy wskazują na początki kolejnych wierszy. A co to jest `m[1]`? To z kolei '`*(m + 1)`'; skoro wartością tego wyrażenia ma być wskaźnik typu **double***, to samo m musi być wskaźnikiem do wskaźnika, a więc mieć typ **double****.

Możemy to przedstawić na rysunku:



W górnym rzędzie mamy sześć elementów naszej tablicy liczb typu **double**. Liczby wewnątrz prostokątów oznaczają wartości tych elementów, tu przykładowo 1.5, 2.5 itd. Na prawo podane są nazwy zmiennych będących elementami tej tablicy. Poniżej podaliśmy przykładowe adresy kolejnych elementów — odległe od siebie o osiem, bo liczby typu **double** zajmują zwykle osiem bajtów (uwaga: w układzie szesnastkowym, w którym zwykle podaje się adresy, $38_{16} + 8_{16} = 40_{16}$, co w układzie dziesiętnym jest równoważne $56 + 8 = 64$). Górny rząd zatem reprezentuje naszą dwuwymiarową tablicę w postaci „wypłaszczonej"; w ciągłym obszarze pamięci po kolei umieszczone są kolejne wiersze tablicy.

W rzędzie środkowym mamy dwuelementową tablicę *wskaźników*. Wartościami są adresy początków wierszy właściwej tablicy liczb. Elementy tej tablicy, jako wskaźniki, są czterobajtowe (lub ośmiobajtowe na platformie 64-bitowej): adresy kolejnych elementów odległe są zatem o cztery bajty.

W dolnym rzędzie mamy pojedynczą zmienną m, której wartością jest z kolei adres tablicy ze środkowego rzędu, a więc adres tablicy wskaźników typu **double***. Zatem jest to wskaźnik do wskaźnika: typ m to **double****.

Zobaczmy, jak można to zrealizować w programie:

P94: ***matrix2dim.cpp*** Dynamiczna tablica dwuwymiarowa

```
#include <iostream>
using namespace std;

double** allocMatrix2D(size_t,size_t);
void     deleteMatrix2D(double**&);

int main() {

    size_t dim1 = 2, dim2 = 3;  // NIE stałe!

      // alokowanie
    double** matrix2d = allocMatrix2D(dim1,dim2);      ①

      // wpisywanie wartości
    for (size_t i = 0; i < dim1; ++i)                  ②
        for (size_t j = 0; j < dim2; ++j)
```

```
17              matrix2d[i][j] = 3*i+j+1.5;
18
19        // drukowanie
20      for (size_t i = 0; i < dim1; ++i) {
21          for (size_t j = 0; j < dim2; ++j)
22              cout << matrix2d[i][j] << " ";
23          cout << endl;
24      }
25
26        // usuwanie
27      deleteMatrix2D(matrix2d);                                   ③
28  }
29
30  double** allocMatrix2D(size_t dim1, size_t dim2) {              ④
31      double** matrix2d = new double*[dim1];
32      double*  dumm     = new double[dim1*dim2];
33
34      for (size_t i = 0; i < dim1; ++i)
35          matrix2d[i] = dumm + i*dim2;
36
37      return matrix2d;
38  }
39
40  void deleteMatrix2D(double**& matrix2d) {                       ⑤
41      delete [] matrix2d[0];
42      delete [] matrix2d;
43      matrix2d = 0;
44  }
```

Dwywymiarowa tablica (macierz) jest tworzona przez funkcję **allocMatrix2D** (④). Zmienne dim1 i dim2 oznaczają wymiary tablicy: dim1 jest liczbą wierszy, dim2 liczbą kolumn.

Najpierw tworzymy zmienną matrix2d typu **double**** i wpisujemy tam adres zaalokowanej tablicy wskaźników typu **double***. Liczba tych wskaźników odpowiada liczbie wierszy w macierzy. Na rysunku zmienna m odpowiada zmiennej macierz2d, a tablica ze środkowego rzędu odpowiada tablicy wskaźników wskazywanej przez matrix2d.

Następnie tworzymy właściwą tablicę liczb. Ich ilość to iloczyn wymiarów. Adres zawarty w dumm to adres początku tej tablicy (na rysunku jest to adres 0x30).

Teraz w pętli wypełniamy tablicę wskaźników ze środkowego rzędu na rysunku: do kolejnych elementów wpisujemy adresy początków wierszy tablicy liczb. Są one od siebie odległe o tyle wielokrotności długości jednej zmiennej typu **double**, ile wynosi liczba kolumn (u nas jest to dim2), czyli jak długi jest jeden wiersz.

Do funkcji wywołującej zwracamy wartość zmiennej matrix2d, która może być używana zgodnie ze składnią macierzową.

W programie głównym używamy tej macierzy: wpisujemy wartości i drukujemy

testowe wyniki

```
1.5 2.5 3.5
4.5 5.5 6.5
```

Po użyciu naszej macierzy, w linii ③ usuwamy ją wywołując funkcję **deleteMatrix2d** (⑤). W funkcji tej musimy wywołać operator **delete** dwa razy, bo tyle razy użyliśmy operatora **new** konstruując macierz. Najpierw zwalniamy tablicę liczb — jej adres to adres początku pierwszego wiersza (o numerze 0), a zatem jest to adres zawarty w macierz2d[0]. Potem zwalniamy tablicę wskaźników wskazywaną przez macierz2d. Kolejność jest tu istotna: gdybyśmy usunęli najpierw tablicę wskaźników, to do tablicy liczb nie mielibyśmy już dostępu. Na zakończenie zerujemy na wszelki wypadek wskaźnik macierz2d. Przesłaliśmy go do funkcji przez referencję, aby to wyzerowanie było skuteczne również w funkcji wywołującej.

W podobny sposób można budować tablice więcej niż dwuwymiarowe. Na przykład poniższy program pokazuje, jak można to zrobić dla macierzy dwu-, trzy- i czterowymiarowych (tym razem są to macierze liczb całkowitych):

P95: *macierze.cpp* Dynamiczne tablice wielowymiarowe

```
#include <iostream>
using namespace std;

int**    allocMatrix2D(int,int);
void     deleteMatrix2D(int**&);

int***   allocMatrix3D(int,int,int);
void     deleteMatrix3D(int***&);

int**** allocMatrix4D(int,int,int,int);
void     deleteMatrix4D(int****&);

int main() {
   int dim1 = 7, dim2 = 9, dim3 = 12, dim4 = 5;

    // Macierze dwuwymiarowe /////////////////

    // alokowanie
    int** matrix2d = allocMatrix2D(dim1,dim2);

    // test
    for ( int i = 0; i < dim1; i++ )
        for ( int j = 0; j < dim2; j++ )
            matrix2d[i][j] = i+j+2;

    cout << "Macierz dwuwymiarowa" << endl
         << " Element srodkowy: "
```

```
28          << matrix2d[dim1/2][dim2/2] << endl
29          << "     Powinno byc : "
30          << dim1/2 + dim2/2 +2 << endl;
31     cout << " Element ostatni : "
32          << matrix2d[dim1-1][dim2-1] << endl
33          << "     Powinno byc : "
34          << dim1 + dim2 << endl << endl;
35
36     // usuwanie
37     deleteMatrix2D(matrix2d);
38
39     // Macierze trzywymiarowe ///////////////////////
40
41     // alokowanie
42     int*** matrix3d = allocMatrix3D(dim1,dim2,dim3);
43
44     // test
45     for ( int i = 0; i < dim1; i++ )
46         for ( int j = 0; j < dim2; j++ )
47             for ( int k = 0; k < dim3; k++ )
48                 matrix3d[i][j][k] = i+j+k+3;
49
50     cout << "Macierz trzywymiarowa" << endl
51          << " Element srodkowy: "
52          << matrix3d[dim1/2][dim2/2][dim3/2] << endl
53          << "     Powinno byc : "
54          << dim1/2 + dim2/2 + dim3/2 + 3 << endl;
55     cout << " Element ostatni : "
56          << matrix3d[dim1-1][dim2-1][dim3-1] << endl
57          << "     Powinno byc : "
58          << dim1 + dim2 + dim3 << endl << endl;
59
60     // usuwanie
61     deleteMatrix3D(matrix3d);
62
63     // Macierze czterowymiarowe ////////////////////////
64
65     // alokowanie
66     int**** matrix4d = allocMatrix4D(dim1,dim2,dim3,dim4);
67
68     // test
69     for ( int i = 0; i < dim1; i++ )
70         for ( int j = 0; j < dim2; j++ )
71             for ( int k = 0; k < dim3; k++ )
72                 for ( int m = 0; m < dim4; m++ )
73                     matrix4d[i][j][k][m] = i+j+k+m+4;
```

```
74
75      cout << "Macierz czterowymiarowa" << endl
76           << " Element srodkowy: "
77           << matrix4d[dim1/2][dim2/2][dim3/2][dim4/2]
78           << endl << "     Powinno byc : "
79           << dim1/2 + dim2/2 + dim3/2 + dim4/2 + 4 << endl;
80      cout << " Element ostatni : "
81           << matrix4d[dim1-1][dim2-1][dim3-1][dim4-1]
82           << endl << "     Powinno byc : "
83           << dim1 + dim2 + dim3 + dim4 << endl << endl;
84
85      // usuwanie
86      deleteMatrix4D(matrix4d);
87  }
88
89  int** allocMatrix2D(int dim1, int dim2) {
90      int** matrix2d = new int*[dim1];
91      int*  dumm     = new int[dim1*dim2];
92      for ( int i = 0; i < dim1; i++ )
93          matrix2d[i] = dumm + i*dim2;
94      return matrix2d;
95  }
96
97  void deleteMatrix2D(int**& matrix2d) {
98      delete [] matrix2d[0];
99      delete [] matrix2d;
100     matrix2d = 0;
101 }
102
103 int*** allocMatrix3D(int dim1, int dim2, int dim3) {
104     int*** matrix3d = new int**[dim1];
105     int**  dumm     = new int*[dim1*dim2];
106     int*   d        = new int[dim1*dim2*dim3];
107     for ( int i = 0; i < dim1; i++ ) {
108         matrix3d[i] = dumm + i*dim2;
109         for ( int j = 0; j < dim2; j++ )
110             dumm[i*dim2+j] = d + (i*dim2+j)*dim3;
111     }
112     return matrix3d;
113 }
114
115 void deleteMatrix3D(int***& matrix3d) {
116     delete [] matrix3d[0][0];
117     delete [] matrix3d[0];
118     delete [] matrix3d;
119     matrix3d = 0;
```

```
120 }
121
122 int**** allocMatrix4D(int dim1,int dim2,int dim3,int dim4) {
123     int**** matrix4d = new int***[dim1];
124     int***  dumm    = new int**[dim1*dim2];
125     int**   dum     = new int*[dim1*dim2*dim3];
126     int*    d       = new int[dim1*dim2*dim3*dim4];
127     for ( int i = 0; i < dim1; i++ ) {
128         matrix4d[i] = dumm + i*dim2;
129         for ( int j = 0; j < dim2; j++ ) {
130             dumm[i*dim2+j] = dum + (i*dim2+j)*dim3;
131             for ( int k = 0; k < dim3; k++ )
132                 dum[(i*dim2+j)*dim3+k] =
133                     d + ((i*dim2+j)*dim3+k)*dim4;
134         }
135     }
136     return matrix4d;
137 }
138
139 void deleteMatrix4D(int****& matrix4d) {
140     delete [] matrix4d[0][0][0];
141     delete [] matrix4d[0][0];
142     delete [] matrix4d[0];
143     delete [] matrix4d;
144     matrix4d = 0;
145 }
```

Zasada tworzenia i usuwania tablic wielowymiarowych jest podobna do tej, którą omówiliśmy dla macierzy dwuwymiarowych, choć konstrukcje, jak widać, stają się szybko niepokojąco „wielopiętrowe”... Wynik tego programu to:

```
Macierz dwuwymiarowa
 Element srodkowy: 9
     Powinno byc : 9
 Element ostatni : 16
     Powinno byc : 16

Macierz trzywymiarowa
 Element srodkowy: 16
     Powinno byc : 16
 Element ostatni : 28
     Powinno byc : 28

Macierz czterowymiarowa
 Element srodkowy: 19
     Powinno byc : 19
 Element ostatni : 33
```

```
Powinno byc : 33
```

12.5 Zarządzanie pamięcią w C

Operatory **new** i **delete** są charakterystyczne dla języka C++. W C podobne zadania realizowane są za pomocą innych funkcji — aby z nich korzystać, należy dołączyć plik nagłówkowy ***cstdlib***. Ponieważ stosuje je się bardzo często w istniejącym kodzie, omówimy je tu pokrótce. Nawet pisząc w C++ używa się czasem funkcji z C, gdyż bywają efektywniejsze, choć są często bardziej niebezpieczne i trudniejsze w programowaniu.

Funkcja **malloc** (ang. *memory allocation*) przydziela obszar w pamięci wolnej. Jej prototyp ma postać

```
void* malloc(size_t size);
```

Typ **size_t** jest aliasem typu całościowego bez znaku, który może zależeć od implementacji (zwykle jest tożsamy z **unsigned long**). Funkcja **malloc** alokuje size bajtów pamięci dynamicznej i zwraca wskaźnik do początku zarezerwowanego obszaru jako wskaźnik typu **void***. Nie ma tu rozróżnienia między alokowaniem pamięci na pojedynczy obiekt i na tablicę; zawsze musimy podać liczbę *bajtów*. Funkcja **malloc** jest rzeczywiście funkcją, wywołuje się ją zatem podając argument w nawiasach okrągłych. Zwraca surowy adres w pamięci, bez żadnej informacji o typie obiektów, jakie będą tam wpisane. A zatem przed przypisaniem tego adresu do zmiennej wskaźnikowej pewnego typu należy dokonać odpowiedniej konwersji, czego nie musieliśmy robić w przypadku operatora **new**. Na przykład

```
1  int* k = (int*) malloc(sizeof(int));
2  *k = 5;
3  // ...
4  int size;
5  cin >> size;
6  int* m = (int*) malloc(size*sizeof(int));
7  for (int i = 0; i < size; ++i)
8      m[i] = 2*i;
```

Zauważmy, że nawet alokując pamięć na pojedynczą zmienną w linii pierwszej tego przykładu musieliśmy jawnie podać liczbę potrzebnych bajtów. W przypadku niepowodzenia funkcja **malloc** zwraca wskaźnik zerowy (NULL, odpowiednik **nullptr**). Przydzielona pamięć nie jest w żaden sposób inicjowana.

Za pomocą innej funkcji, **calloc**, możemy zaalokować obszar pamięci i od razu zainicjować go zerami. Prototyp ma postać

```
void* calloc(size_t count, size_t size);
```

Parametr count oznacza liczbę elementów, jakie alokowany obszar ma pomieścić, a size rozmiar (w bajtach) jednego elementu. Zatem, aby zaalokować tablicę liczb całkowitych i zainicjować ją zerami, można napisać

```
int dim;
cin >> dim;
int* tab = (int*) calloc(dim,sizeof(int));
```

Zauważmy, że wciąż nie ma tu informacji o typie elementów — zwracany jest wskaźnik typu **void***.

Istnieje również funkcja **realloc** zmieniająca rozmiar bloku pamięci dynamicznej wcześniej zaalokowanego:

```
void* realloc(void* ptr, size_t size);
```

której pierwszym argumentem jest wskaźnik zwrócony wcześniej przez **malloc**, **calloc** lub **realloc**, a size jest nowym rozmiarem. Jeśli rozmiar size jest większy niż ten który został podany przy wywołaniu funkcji **malloc**, to zostanie zarezerwowany nowy obszar o zwiększonym rozmiarze, a zawartość poprzedniego bloku pamięci zostanie do niego skopiowana, po czym stary obszar zostanie zwolniony. Funkcja zawsze zwraca adres nowego obszaru, który jednak może pokrywać się ze starym, jeśli size wcale nie jest większe od dotychczasowego rozmiaru.

W przypadku niepowodzenia **realloc** zwraca wskaźnik zerowy (NULL, **nullptr**), a dotychczasowy blok pamięci pozostaje nienaruszony.

W tradycyjnym C do zwalniania pamięci dynamicznej służy funkcja **free**

```
void free(void* ptr);
```

Wartością argumentu ptr musi być adres zwrócony wcześniej przez funkcję **malloc**, **calloc** lub **realloc** podczas alokacji (lub realokacji).

Obszar pamięci zaalokowany za pomocą operatora **new** (**new[]**) należy zwalniać za pomocą **delete** (**delete[]**). Jeśli do przydzielenia pamięci użyto funkcji **malloc**, **calloc** lub **realloc**, to zwolnić ją trzeba za pomocą funkcji **free**.

Podobnie jak nie należy mieszać **new** i **delete** tablicowych z nietablicowymi, nie wolno też mieszać funkcji przydzielających pamięc z C z funkcjami zwalniającymi pamięć z C++ i odwrotnie.

12.6 Funkcje operujące na pamięci

Standardowa biblioteka w nagłówku ***cstring*** (lub ***string.h***) dostarcza wielu funkcji użytecznych do manipulowania zawartością pamięci. Niektóre z nich to:

void* memcpy(void* target, const void* source, size_t len) — kopiuje len bajtów od pozycji wskazywanej przez source do obszaru pamięci rozpoczynającego się od adresu w target; zwraca target. Obszar źródłowy i docelowy nie mogą się przekrywać. Tak więc jeśli chcemy przekopiować tablicę liczb całkowitych tab o wymiarze size do nowo utworzonej tablicy t, możemy to zrobić tak

```
int* t = new int[size];
memcpy(t,tab,size*sizeof(int));
```

co jest dla długich tablic znaczenie szybsze niż kopiowanie w pętli element po elemencie.
Zauważmy, że kopiowanie jest „z prawa na lewo" — pierwszy argument wskazuje miejsce przeznaczenia, a drugi obszar źródłowy!

void* memmove(void* target, const void* source, size_t len) — jest podobna do **memcpy**, ale obszar docelowy *może* się częściowo pokrywać z obszarem źródłowym; jest wolniejsza od funkcji **memcpy**.

void* memchr(const void* str, int znak, size_t len) — poszukuje bajtu (znaku) będącego najmłodszym bajtem argumentu znak w len bajtach poczynając od pozycji w pamięci wskazywanej przez str. Zwraca wskaźnik do znalezionego bajtu (znaku) lub wskaźnik pusty (NULL, **nullptr**) jeśli poszukiwanie nie zakończyło się sukcesem.

int memcmp(const void* p, const void* q, size_t len) — porównuje leksykograficznie pierwsze len bajtów ciągów bajtów rozpoczynających się na pozycjach wskazywanych przez p i q; zwraca całkowitą wartość ujemną jeśli ciąg wskazywany przez p jest leksykograficznie wcześniejszy niż ciąg wskazywany przez q, 0 jeśli ciągi są identyczne, oraz całkowitą wartość dodatnią jeśli p jest leksykograficznie późniejsze niż q.

void* memset(void* p, int znak, size_t len) — wypełnia len bajtów pamięci najmłodszym bajtem wartości znak poczynając od pozycji wskazywanej przez wskaźnik p. Zwraca p.

Poniższy prosty programik demonstruje użycie tych funkcji:

P96: *rotate.cpp* Funkcje operujące na pamięci

```
#include <iostream>
#include <cstring>   // memcpy, memmove
using namespace std;

template <typename T>
T* rotate_left(T arr[], size_t size, size_t shift) {

    if ((shift %= size) == 0) return arr;          ①

    T* aux = new T[shift];

    memcpy(aux,arr,shift*sizeof(T));               ②
    memmove(arr,arr+shift,(size-shift)*sizeof(T));
    memcpy(arr+size-shift,aux,shift*sizeof(T));

    delete [] aux;
    return arr;
```

```
18 }
19
20 template <typename T>
21 void writeArr(const char* mes, const T arr[], size_t size) {
22     cout << mes << ": " << "[ ";
23     for (size_t i = 0; i < size; ++i)
24         cout << arr[i] << " ";
25     cout << "]" << endl;
26 }
27
28 int main() {
29     char arrc[] = {'a','b','c','d','e','f'};
30     writeArr("tab. znakow",arrc,6);
31     rotate_left(arrc,6,8);
32     writeArr("   rot. o 8",arrc,6);
33     rotate_left(arrc,6,1);
34     writeArr("  potem o 1",arrc,6);
35
36     cout << endl;
37
38     int arri[] = {1,2,3,4,5,6,7,8,9};
39     writeArr("tab. int'ow",arri,9);
40     rotate_left(arri,9,7);
41     writeArr("   rot. o 7",arri,9);
42 }
```

Funkcja **rotate_left** przesuwa elementy tablicy o wymiarze size o shift pozycji w lewo w taki sposób, że elementy „wychodzące" z lewej strony pojawiają się po prawej (rotacja). Aby zabezpieczyć się przed przypadkiem shift > size w linii ① brana jest reszta z dzielenia shift przez size. W linii ② kopiujemy shift pierwszych elementów do tablicy pomocniczej, następnie przesuwamy pozostałe elementy w lewo za pomocą **memmove**, po czym wstawiamy (kopiujemy) zapamiętane elementy po prawej stronie tablicy (nie zapominając o usunięciu tablicy pomocniczej). Wydruk z tego programu:

```
tab. znakow: [ a b c d e f ]
   rot. o 8: [ c d e f a b ]
  potem o 1: [ d e f a b c ]

tab. int'ow: [ 1 2 3 4 5 6 7 8 9 ]
   rot. o 7: [ 8 9 1 2 3 4 5 6 7 ]
```

12.7 Lokalizujący przydział pamięci

Za pomocą operatora **new** można przydzielić pamięc o z góry określonej lokalizacji. Istnieje do tego celu specjalna forma operatora **new** — dostępna po dołączeniu pliku nagłówkowego ***new*** — o następującej składni:

```
new (adres) Typ;
new (adres) Typ[wymiar];
```

W ten sposób przydzielamy na pojedynczy obiekt lub tablicę obszar pamięci rozpoczynający się od adresu będącego wartością wyrażenia adres. W zasadzie obszar ten powinien mieścić się w jakimś obszarze pamięci przydzielonym wcześniej za pomocą „zwykłego” **new**. Jest to operacja szybka, ponieważ tak naprawdę nie wymaga alokowania pamięci; kompilator zakłada, że programista wie co robi. Ponieważ tak naprawdę przydziału pamięci nie ma, obszar ten powinien zostać zwolniony przez wywołanie **delete** ze wskaźnikiem takiego typu i zawierającym ten adres, który był użyty w „prawdziwym” **new**, a nie w **new** lokalizującym!

W poniższym przykładzie alokujemy (①) tablicę arr znaków (czyli bajtów) o wymiarze pozwalającym pomieścić tam trzy sz-elementowe tablice różnych typów (**string**, **double** i **int**). Wewnątrz obszaru pamięci przez nią zajmowanego „alokujemy” trzy osobne tablice, obliczając za każdym razem (patrz na przykład ②) pod jakim adresem powinny się rozpoczynać, aby pomieścić się w dostępnej pamięci:

P97: ***new.cpp*** Lokalizujący przydział pamięci

```
#include <iostream>
#include <string>

int main() {
    using std::string; using std::cout; using std::endl;
    int sz = 3;

    char* arr = new char[sz*(sizeof(string) +          ①
                             sizeof(double)+sizeof(int))];

    string* nam = new (arr) string[sz]{"Sue", "Kim", "Joe"};
    double* wei = new (arr+sz*sizeof(string))          ②
                            double[sz]{55.5, 61.2, 81.5};
    int* hei = new (arr+sz*(sizeof(string)+sizeof(double)))
                            int[sz]{170, 165, 183};
    for (int i = 0; i < sz; ++i)
        cout << nam[i] << " " << wei[i] << " "
             << hei[i] << endl;
    delete [] arr;                                     ③
}
```

Wydruk

```
Sue 55.5 170
Kim 61.2 165
Joe 81.5 183
```

świadczy o tym, że tablice zostały prawidłowo zaalokowane i zainicjowane. Pamiętamy

oczywiście, aby zwalniając pamięć (③) użyć adresu otrzymanego przez „prawdziwe” **new**!

Opisany rodzaj przydzielania pamięci stosuje się też często aby wykorzystać na nowe dane wcześniej zaalokowane, a zawierające dane już niepotrzebne, obszary pamięci — jest to *znacznie* szybsze niż zwalnianie i potem przydzielanie pamięci od nowa.

— 13 —

C-struktury i unie

W języku C++, jak w każdym języku obiektowym, mamy możliwość definiowania własnych typów danych, wraz z określeniem operacji, jakie na tych danych można wykonywać. Służą do tego (jak w Javie) **klasy**, występujące w C++, z powodów głównie historycznych, pod dwoma nazwami: klas i struktur.

Jednak typy złożone istnieją również w czystym C. Mają one postać unii i struktur. W C++ implementacja struktur została rozszerzona (i jest w zasadzie taka jak dla klas), ale warto zdawać sobie sprawę, jak wyglądają struktury zgodne z implementacją w C. Tego typu struktury, które będziemy nazywać C-strukturami, są bardzo często używane nawet w programach napisanych w C++, głównie ze względu na to, że C-strukturami są często typy zdefiniowane w standardowych bibliotekach.

Podrozdziały:

13.1 C-struktury

Struktura definiuje nowy typ danych. Jest to typ złożony: pojedyncza zmienna typu strukturalnego może zawierać wiele danych (liczb, napisów, wskaźników, itd.). W odróżnieniu od tablic, dane te nie muszą być tego samego typu (jak elementy tablicy).

C-struktura jest kolekcją nazwanych składowych, które mogą być różnych typów, również innych typów strukturalnych.

Definicja C-struktury może mieć następującą postać:

```
struct Nazwa {
    Typ1 sklad1;
    Typ2 sklad2;
    ...
};
```

Nazwy Typ1 i Typ2 są tu nazwami typów (innych niż nazwa typu strukturalnego który właśnie jest definiowany, czyli tutaj **Nazwa**). Nazwy sklad1 i sklad2 dowolnymi indentyfikatorami **pól** tej struktury. Każda zmienna tego typu strukturalnego będzie

zawierać **składowe** o typach i nazwach odpowiadających polom struktury. Na razie jednak jest to tylko definicja typu: nie istnieją jeszcze żadne tego typu zmienne.

Średnik na końcu, po nawiasie zamykającym, jest konieczny!

W C++ po takiej samej definicji nazwą typu jest po prostu **Nazwa**. Natomiast w C nazwą tego typu jest **struct Nazwa** — konieczne jest zatem powtórzenie słowa kluczowego **struct** przy późniejszym użyciu nazwy tego typu. Można tego uniknąć stosując standardową „sztuczkę" ze specyfikatorem **typedef**:

```
typedef struct Nazwa {
    // definicja struktury
} Nazwa;
```

W ten sposób nazwa **Nazwa** staje się (już jednowyrazowym) aliasem pełnej nazwy **struct Nazwa**.

Kiedy już zdefiniowaliśmy typ, możemy definiować zmienne tego typu; składnia jest dokładnie taka sama jak składnia deklaracji/definicji zmiennych typów wbudowanych (jak **double** czy **int**):

```
struct Nazwa a, b;   /* C i C++ */
Nazwa c, d;          // tylko C++
```

Można też definiować zmienne tego typu bezpośrednio za definicją, a przed średnikiem:

```
struct Nazwa {
    Typ1 skladowa1;
    Typ2 skladowa2;
} a, b, c, d;
```

definiuje typ strukturalny i od razu definiuje cztery zmienne tego typu. Ten ostatni zapis daje możliwość tworzenia obiektów struktur anonimowych:

```
struct {
    Typ1 skladowa1;
    Typ2 skladowa2;
} a, b;
```

Za pomocą takiej konstrukcji stworzyliśmy dwa obiekty strukturalne o nazwach a i b. Każdy z nich zawiera dwie składowe odpowiadające dwóm polom w definicji struktury. Do składowych tych można się odnosić w sposób opisany poniżej, jak do składowych zwykłych struktur posiadających nazwę. Nie da się już jednak zdefiniować innych obiektów tego samego typu, bo typ ten nie ma żadnej nazwy i nie ma jak się do niego odwołać!

Każdy pojedynczy obiekt typu zdefiniowanego jako C-struktura zawiera tyle składowych i takich typów, jak to zostało określone w definicji tej C-struktury. Jeśli zdefiniowany (utworzony) jest obiekt, to do jego składowych odnosimy się za pomocą operatora wyboru składowej. Operator ten ma dwie formy (patrz pozycje 4 i 5 w

tabeli operatorów na str. 123), w zależności od tego, czy odnosimy się do obiektu poprzez jego nazwę, czy poprzez nazwę wskaźnika do niego. Jeśli a jest nazwą *obiektu*, to do jego składowej o nazwie sklad odnosimy się za pomocą operatora „kropki":

```
a.sklad
```

Ta sama reguła stosowałaby się, gdyby a było nazwą *referencji* do obiektu — pamiętajmy jednak, że w czystym C odnośników (referencji) nie ma. Natomiast jeśli pa jest *wskaźnikiem* do pewnego obiektu struktury, to do tej samej składowej odnosimy się za pomocą operatora „strzałki":

```
pa->sklad
```

(„strzałka" to dwuznak złożony z myślnika i znaku większości; odstęp między tymi dwoma znakami jest niedopuszczalny).

Ta sama zasada dotyczy nie tylko struktur, które mają postać C-struktur, ale wszystkich struktur i klas w C++.

Zauważmy, że forma `pa->sklad` jest w zasadzie notacyjnym skrótem zapisu `(*pa).sklad`, gdyż `*pa` jest właśnie l-wartością obiektu wskazywanego przez pa. Zatem jeśli a jest identyfikatorem obiektu struktury posiadającej pole x typu **double**, a pa wskaźnikiem do tego obiektu, to następujące instrukcje są równoważne:

```
1    a.x      = 3.14;
2    (&a)->x  = 3.14;
3    pa->x    = 3.14;
4    (*pa).x  = 3.14;
```

W linii 2 i 4 nawiasy są potrzebne, gdyż operatory wyboru składowej (kropka i „strzałka") mają wyższy priorytet niż operatory dereferencji i wyłuskania adresu ('`*`' i '`&`').

Zapamiętajmy zatem:

> Zapis `a.b` oznacza składową o nazwie b obiektu o nazwie a. Zapis `pa->b` oznacza składową o nazwie b obiektu *wskazywanego* przez wskaźnik o nazwie pa.

Tworząc obiekt typu C-struktury można go od razu zainicjować. Składnia jest podobna do tej, jakiej używamy do inicjowania tablicy:

```
Nazwa ob = {wyr_1, wyr_2};
```

gdzie wyr_1 i wyr_2 są wyrażeniami, których wartości mają posłużyć do zainicjowania składowych obiektu w kolejności takiej, w jakiej zadeklarowane zostały odpowiednie pola struktury. Inicjatorów może być mniej niż pól; składowe odpowiadające pozostałym polom zostaną wtedy zainicjowane zerami odpowiedniego typu. Słowo

kluczowe **struct** w powyższej instrukcji może być pominięte w C++, natomiast jest niezbędne w C.

W poniższym przykładzie definiujemy strukturę **Sam** (①) opisującą, w bardzo uproszczony sposób, samochody. Ma ona dwa pola: predk i roczn typu, odpowiednio, **double** i **int**. Dwie zmienne typu **Sam** od razu definiujemy: zmienną skoda i zmienną fiat. Tę drugą również inicjujemy podając w nawiasach klamrowych wartości inicjatorów w kolejności odpowiadającej kolejności pól w definicji struktury **Sam**. Zmienna skoda jest utworzona, ale nie została zainicjowana, więc wartości jej składowych są na razie nieokreślone. Obie zmienne są globalne.

P98: *cstru.cpp* C-struktury

```
#include <iostream>
using namespace std;

struct Sam {                                        ①
    double predk;
    int    roczn;
} skoda, fiat{100, 1998};

void pr(const char*,const Sam*);

int main() {
    Sam toyota, *mojsam = &toyota, *vw;             ②

    cout << "Obiekty \'Sam\' maja rozmiar "         ③
         << sizeof(Sam) << " bajtow\n";
    skoda.predk =  120;                             ④
    skoda.roczn = 1995;

    toyota.roczn  = 2012;                           ⑤
    mojsam->predk =  180;

    vw = new Sam{175,2003};                         ⑥

    pr("Skoda ",&skoda);
    pr("Fiat  ",&fiat);
    pr("Toyota",&toyota);
    pr("mojsam", mojsam);
    pr("VW    ", vw);

    delete vw;
}

void pr(const char *nazwa, const Sam *sam) {
    cout << nazwa << ": predkosc " << sam->predk
         << ", rocznik " << sam->roczn << endl;
```

```
36 }
```

Wewnątrz funkcji **main** definiujemy jeszcze jeden obiekt typu **Sam** o nazwie toyota. Prócz tego definiujemy dwie zmienne wskaźnikowe: mojsam zainicjowaną adresem obiektu toyota oraz vw, która na razie nie jest zainicjowana (linia ②).

W linii ③ drukujemy rozmiar obiektów typu **Sam**. Może to być 12 bajtów: 8 bajtów na składową predk typu **double** i 4 bajty na składową roczn typu **int**. Na wielu platformach preferowane są rozmiary będące wielokrotnością 8 bajtów; wtedy każdy obiekt będzie zawierał dodatkowe, nieużywane bajty, tzw. **padding** (wypełnienie). Na komputerze autora ten właśnie przypadek ma miejsce, o czym przekonuje nas wydruk:

```
Obiekty 'Sam' maja rozmiar 16 bajtow
Skoda : predkosc 120, rocznik 1995
Fiat  : predkosc 100, rocznik 1998
Toyota: predkosc 180, rocznik 2012
mojsam: predkosc 180, rocznik 2012
VW    : predkosc 175, rocznik 2003
```

W liniach ④ i następnej wpisujemy wartości do składowych obiektu skoda. Ponieważ skoda jest identyfikatorem *obiektu*, a nie wskaźnika, używamy operatora „kropki”.

W liniach ⑤ i następnej wpisujemy wartości do składowych obiektu toyota. W linii ⑤ odnosimy się do tego obiektu poprzez jego nazwę, więc używamy notacji z kropką; w linii następnej do *tego samego* obiektu (patrz linia ②) odnosimy się poprzez wskaźnik mojsam, zatem używamy notacji ze „strzałką”.

W linii ⑥ alokujemy na stercie (poprzez **new**) nowy obiekt typu **Sam** i zwrócony adres wpisujemy do wskaźnika vw. Obiekt jest od razu inicjowany poprzez użycie składni z listą inicjatorów w nawiasach klamrowych; jest to dozwolone w nowym standardzie C++11 (zauważmy jednak, że użycie tu znaku równości przed otwierającą klamrą byłoby błędem).

Następnie drukujemy informacje o naszych zmiennych za pomocą funkcji **pr**. Funkcja ta ma drugi parametr typu wskaźnikowego **const Sam***, a więc trzeba do niej wysłać adres obiektu; dlatego jako argument wpisujemy albo zmienną wskaźnikową (bo jej wartością jest właśnie adres), albo, jeśli używamy nazwy obiektu, wyłuskujemy jego adres za pomocą operatora ’&’.

C-struktury są bardzo często używane w funkcjach bibliotecznych. Na przykład, włączając plik nagłówkowy ***sys/timeb.h*** mamy do dyspozycji funkcję **ftime** (funkcje udostępniane przez nagłówek ***sys/timeb.h*** nie należą do standardu, ale są zaimplementowane w Linuksie, w wersji C++ firmy Microsoft i w większości innych implementacji tego języka). Funkcja **ftime** wymaga argumentu w postaci *adresu* obiektu struktury **timeb** zdefiniowanej następująco (ta definicja jest już widoczna po włączeniu pliku nagłówkowego ***sys/timeb.h***):

```
struct timeb {
    time_t          time;
    unsigned short  millitm;
    short           timezone;
```

```
    short           dstflag;
};
```

Korzystając z funkcji systemowych, funkcja wypełnia ten obiekt danymi. Składowa time będzie liczbą sekund od początku epoki, czyli od 1 stycznia 1970 roku, do chwili obecnej. Jej typ, **time_t**, odpowiada pewnemu typowi całkowitemu ze znakiem. Zwykle jest to **long**, a więc maksymalna wartość tej składowej dla maszyn 32-bitowych, na których **sizeof**(**long**) jest 4, wynosi $2^{31} - 1 = 2147483647 \approx 2.1 \cdot 10^9$ i zostanie osiągnięta w roku 2038 (jeden rok to w przybliżeniu $10^7\pi$ sekund). Składowa millitm to liczba milisekund od początku ostatniej pełnej sekundy. Składowe timezone i dstflag nie są używane. Za pomocą funkcji **ftime** możemy, choć nie jest to najlepsza metoda, mierzyć czas wykonania pewnych fragmentów programu:

P99: *czas.cpp* Wykorzystanie struktur bibliotecznych

```
#include <iostream>
#include <cmath>          // sin, cos
#include <sys/timeb.h>    // ftime
using namespace std;

int main() {
    timeb start, teraz;                                          ①
    double res = 0;

    ftime(&start);                                               ②

    for (int i = 0; i <= 90000000; ++i) {
        if (i%10000000 == 0) {
            ftime(&teraz);                                       ③
            time_t sec = teraz.time - start.time;
            int msec = teraz.millitm;
            msec -=  start.millitm;
            if (msec < 0) {
                --sec;
                msec += 1000;
            }
            cout << "Po " << i << " iteracjach: "
                 << sec << "s and " << msec << "ms\n";
        }
        res = cos(res+sin(i));
    }
    cout << "Bezuzyteczny wynik: " << res << endl;
}
```

W linii ① tworzymy dwie zmienne typu **timeb**: zmienną start i teraz. W linii ② pierwszą z nich wypełniamy aktualnymi danymi wywołując funkcję **ftime**, a następnie

z drugą z nich robimy to samo w pętli (③), co 10 milionów obrotów, za każdym razem obliczając czas, jaki upłynął do tego momentu od chwili uruchomienia programu:

```
Po 0 iteracjach: 0s and 0ms
Po 10000000 iteracjach: 0s and 886ms
Po 20000000 iteracjach: 1s and 769ms
Po 30000000 iteracjach: 2s and 663ms
Po 40000000 iteracjach: 3s and 555ms
Po 50000000 iteracjach: 4s and 438ms
Po 60000000 iteracjach: 5s and 321ms
Po 70000000 iteracjach: 6s and 205ms
Po 80000000 iteracjach: 7s and 90ms
Po 90000000 iteracjach: 7s and 974ms
Bezuzyteczny wynik: 0.953078
```

Zmienna res jest tu obliczana tylko po to, aby dać programowi coś do roboty...

Nic nie stoi na przeszkodzie, aby składowymi struktury były obiekty innej struktury. Oczywiście, składowymi struktury *nie* mogą być obiekty *tej samej* struktury, bo one też musiałyby zawierać obiekty tej struktury, które z kolei też musiałyby zawierać ... i tak *ad infinitum*...

Wyobraźmy sobie, że chcemy opisywać punkty na płaszczyźnie. Naturalną reprezentacją takich obiektów byłaby struktura o dwóch polach odpowiadających współrzędnym kartezjańskim punktu

```
struct Punkt {
    double x, y;
};
```

Gdybyśmy teraz potrzebowali trójkąta, moglibyśmy zdefiniować go jako strukturę złożoną z trzech punktów odpowiadających jego wierzchołkom

```
struct Trojkat {
    Punkt A, B, C;
};
```

Ponieważ w C-strukturach nie ma metod i konstruktorów, operacje na tych obiektach musielibyśmy zdefiniować za pomocą globalnych funkcji. Na przykład w poniższym programie definiujemy funkcję realizującą obrót punktu na płaszczyźnie kartezjańskiej wokół początku układu współrzędnych. Taki obrót o kąt ϕ opisany jest wzorami na współrzędne punktu po obrocie (zmienne primowane) w funkcji współrzędnych przed obrotem (zmienne nieprimowane):

$$x' = x\cos\phi - y\sin\phi \qquad y' = x\sin\phi + y\cos\phi$$

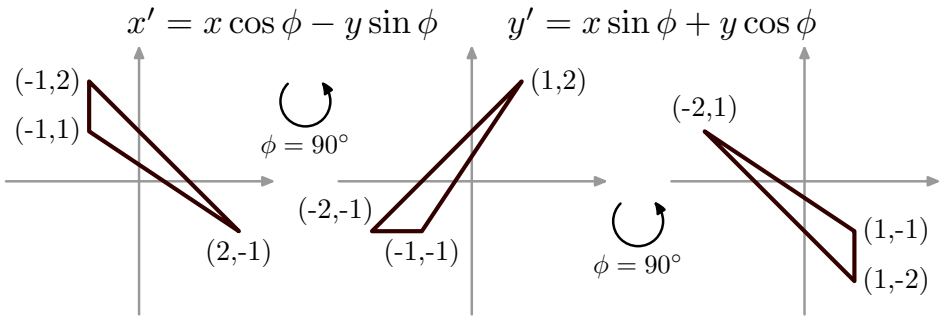
Odpowiada temu funkcja **rot** zadeklarowana w linii ③ poniższego programu:

P100: ***obroty.cpp*** Struktury jako składowe struktur

```
#include <iostream>
#include <cmath>     // sin, cos
using namespace std;

struct Punkt {
    double x, y;
};

struct Trojkat {
    Punkt A, B, C;
};

void info(const Punkt*);    ①
void info(const Trojkat*); ②
void rot(Punkt*,double);    ③
void rot(Trojkat*,double); ④

int main() {
    Punkt A;
    A.x = -1;
    A.y =  2;

    Punkt B = { -1, 1 };

    Punkt C = { 2 };
    C.y = -1;

    Trojkat T = { A, B };
    T.C = C;

    cout << "Wyjsciowe punty: ";
    info(&A); info(&B); info(&C);
    cout << "\nTrojkat: ";
    info(&T);
    cout << endl;

    rot(&A,90); rot(&B,90); rot(&C,90);                    ⑤
    cout << "A, B, C po obrocie o 90 stopni:\n    ";
    info(&A); info(&B); info(&C);
    cout << endl;

    rot(&T,90); rot(&T,90);                                ⑥
    cout << "T po obrocie dwa razy o 90 stopni:\n    ";
    info(&T);
```

```
45
46      rot(&T,180);                                                    ⑦
47      cout << "T po obrocie o nastepne 180 stopni:\n    ";
48      info(&T);
49  }
50
51  void info(const Punkt* pP) {
52      cout << "(" << pP->x << ", " << pP->y << ") ";
53  }
54
55  void info(const Trojkat* pT) {
56      cout << "A="; info(&pT->A);                                     ⑧
57      cout << "B="; info(&pT->B);
58      cout << "C="; info(&pT->C);
59      cout << endl;
60  }
61
62  void rot(Punkt* pP, double phi) {
63      static double conver = atan(1.)/45;
64      phi = phi*conver; // stopnie -> radiany
65
66      double c = pP->x;
67      pP->x = pP->x * cos(phi) - pP->y * sin(phi);
68      pP->y =      c * sin(phi) + pP->y * cos(phi);
69  }
70
71  void rot(Trojkat* pT, double phi) {
72      rot( &pT->A, phi);                                              ⑨
73      rot( &pT->B, phi);
74      rot( &pT->C, phi);
75  }
```

Funkcja ta pobiera adres punktu, dzięki czemu może pracować na oryginale poprzez wskaźnik (dlatego w jej definicji używamy „strzałek” a nie kropek). Jako przykład wybierzmy punkty $A = (-1, 2)$, $B = (-1, 1)$, $C = (2, -1)$, jak po lewej stronie rysunku:



Punkty tworzymy na początku programu albo inicjując je od razu całkowicie, albo tylko częściowo, albo w ogóle nie inicjując i wpisując potem wartości składowych „ręcznie”.

Następnie definiujemy trójkąt o wierzchołkach w punktach opisanych zmiennymi A, B i C. Zauważmy, że przypisanie wartości tych obiektów składowym obiektu T powoduje kopiowanie tych wartości. Po zainicjowaniu obiekt T jest niezależny od obiektów A, B i C. Drukując informację o punktach i trójkącier widzimy, że odpowiada ona sytuacji po lewej stronie rysunku.

Następnie, w linii ⑤, dla każdego z punktów wywołujemy funkcję **rot**, która oblicza ich nowe współrzędne po obrocie o 90°; ponieważ funkcja pracuje na oryginale, stare współrzędne są zamazywane. Punkty zatem uległy zmianie — ich współrzędne odpowiadają teraz sytuacji ze środkowej części rysunku. Zauważmy, że sam trójkąt T nie zmienił się; jego składowe są bowiem kopiami oryginalnych punktów sprzed zmiany.

```
Wyjsciowe punty: (-1, 2) (-1, 1) (2, -1)
Trojkat: A=(-1, 2) B=(-1, 1) C=(2, -1)

A, B, C po obrocie o 90 stopni:
    (-2, -1) (-1, -1) (1, 2)
T po obrocie dwa razy o 90 stopni:
    A=(1, -2) B=(1, -1) C=(-2, 1)
T po obrocie o nastepne 180 stopni:
    A=(-1, 2) B=(-1, 1) C=(2, -1)
```

W linii ⑥ obracamy z kolei trójkąt. Zauważmy, że funkcja do obracania trójkątów też nazywa się **rot**, dokładnie tak jak funkcja do obracania punktów (③ i ④); ma jednak parametry innego typu, więc „nie myli się” kompilatorowi z funkcją **rot** dla punktów (podobne przeciążenie zastosowaliśmy dla dwóch funkcji **info**: dla punktów ① i dla trójkątów ②). Funkcja **rot** dla trójkątów jest bardzo prosta i nie wymaga żadnej wiedzy z trygonometrii: po prostu wywołuje funkcję **rot** dla punktów posyłając do niej adresy punktów, które są składowymi obiektu typu **Trojkat**, dla którego sama została wywołana. Użyta w liniach ⑧ i ⑨ konstrukcja `&pT->A` jest prawidłowa i oznacza adres obiektu A (punktu) będącego składową obiektu typu **Trojkat** wskazywanego przez wskaźnik pT — czytelniejszy może byłby tu zapis `&(pT->A)`, który oznacza to samo, bo priorytet operatora wyboru składowej przez wskaźnik (’`->`’) jest wyższy od priorytetu wyłuskania adresu (’`&`’).

Po dwukrotnym obrocie o kąt 90°, czyli w sumie o kąt 180°, trójkąt powinien być (i jest, jak widać z wydruku) ulokowany jak w prawej części rysunku. Po kolejnym obrocie o 180° (⑦) trójkąt wraca do pozycji wyjściowej przedstawionej po lewej stronie rysunku i w ostatniej linii wydruku.

Składową struktury lub klasy nie może, z oczywistych względów, być obiekt tejże klasy. Nic jednak nie stoi na przeszkodzie, aby taką składową był *wskaźnik* do obiektu tej struktury/klasy. Jest to przypadek bardzo często spotykany i użyteczny, ponieważ umożliwia tworzenie *list* obiektów. W każdym obiekcie struktury zawarta jest wtedy składowa będąca adresem następnego obiektu na liście. Mówimy wtedy o liście jednokierunkowej (ang. *singly-linked* — stosuje się też listy dwukierunkowe (ang. *doubly-*

linked), w których każdy obiekt „pamięta” adres zarówno swego następnika, jak i poprzednika.

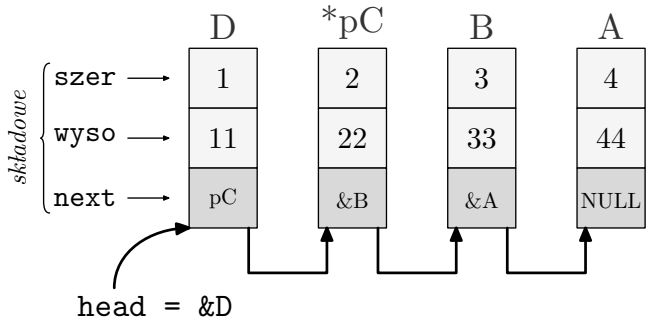
Zobaczmy, jak można taką prostą listę utworzyć. Konstruujemy strukturę **Wezel** zawierającą pewną liczbę pól z jakąś informacją; w przykładzie poniżej są to dwie liczby typu **double** — polom tym nadajemy nazwy szer i wyso. Prócz tego definiujemy dodatkowe pole next typu **Wezel***. Składowe obiektów odpowiadające temu polu będą właśnie zawierać adresy obiektów następnych na liście. Umawiamy się, że element ostatni, który już następnika nie ma, ma w tej składowej wskaźnik pusty **nullptr**. Inna często spotykana konwencja polega na tym, że element ostatni zawiera wskaźnik wskazujący na samego siebie lub na element pierwszy — w tym ostatnim przypadku otrzymujemy listę cykliczną.

P101: ***simplista.cpp*** Prosta lista

```
#include <iostream>
using namespace std;

struct Wezel {
    double szer;
    double wyso;
    Wezel *next;
};

void wstaw_dane(Wezel* n, double s, double w, Wezel* next);
void drukuj_liste(const Wezel* n);
void drukuj_liste_odwrotnie(const Wezel* n);

int main() {
    Wezel A = {4, 44, nullptr};          ①
    Wezel B, D, *head;

    Wezel* pC = new Wezel;               ②

    wstaw_dane(&B,3,33,&A);
    wstaw_dane(pC,2,22,&B);
    wstaw_dane(&D,1,11,pC);

    head = &D;                           ③

    drukuj_liste(head);
    drukuj_liste_odwrotnie(head);

    delete pC;
}

void wstaw_dane(Wezel* n, double s, double w, Wezel* next) {
    n->szer = s;
```

```
34      n->wyso = w;
35      n->next = next;
36  }
37
38  void drukuj_liste(const Wezel* n) {
39      for ( ; n; n = n->next )
40          cout << n->szer << " " << n->wyso << "; ";
41      cout << endl;
42  }
43
44  void drukuj_liste_odwrotnie(const Wezel* n) {
45      if (n == nullptr) return;   // pusta lista
46      if (n->next != nullptr)                ④
47          drukuj_liste_odwrotnie(n->next);
48      cout << n->szer << " " << n->wyso << "; ";
49  }
```

W linii ① tworzymy obiekt A typu **Wezel** i od razu inicjujemy jego trzy składowe; taka składnia inicjowania dozwolona jest jak wiemy dla C-struktur, choć nie dla bardziej skomplikowanych struktur dopuszczanych przez C++. W linii następnej definiujemy dwa obiekty struktury — B i D — oraz wskaźnik do obiektów tej struktury head. W linii ② alokujemy na stercie jeszcze jeden obiekt; adres zwrócony przez **new** przypisujemy do pC. Następnie wypełniamy utworzone obiekty danymi za pomocą funkcji **wstaw_dane**. Pobiera ona adres obiektu do wypełnienia, dwie liczby typu **double** jako dane oraz adres innego obiektu, który ma być wartością składowej next i wskazywać na obiekt następny na liście. Zauważmy, że aby pobrać adres, stosujemy operator wyłuskania adresu '&' dla A, B i D, ale nie dla obiektu wskazywanego przez pC: pC jest zmienną wskaźnikową i już zawiera właśnie adres obiektu; sam obiekt wskazywany jest na stercie i w ogóle nie ma nazwy. Widzimy, że B zawiera adres A jako następującego po nim; dla obiektu wskazywanego przez pC następnym jest B, a dla D następnym jest *pC, czyli obiekt wskazywany przez pC. Sytuacja ta przedstawiona jest na rysunku:



Zatem pierwszym elementem listy jest obiekt D (taki element, a właściwie wskaźnik do niego, nazywamy *głową* – ang. *head* – listy). Ostatnim natomiast jest obiekt A (jest to tzw. *ogon*, ang. *tail*; jego składowa next zawiera wskaźnik pusty **nullptr**).

W linii ③ zapamiętujemy w zmiennej head adres głowy listy. Zauważmy, że aby

operować na liście, wystarczy nam ten adres: mając dostęp do pierwszego elementu, możemy w jego składowej next znaleźć adres następnego itd.

Jako przykład w programie zdefiniowane są dwie funkcje: **drukuj_liste** i **drukuj_liste_odwrotnie**. Obie pobierają wyłącznie adres głowy listy. Pierwsza z nich przebiega po liście w pętli **for** drukując wartości danych; za każdym obrotem pętli wartość wskaźnika do obiektu z listy n zamieniana jest na 'n->next', aż wartością n stanie się **nullptr**, co nastąpi po dotarciu do elementu A.

Ciekawsza jest druga funkcja, **drukuj_liste_odwrotnie**. Jej zadaniem jest wydrukować informacje o elementach listy w odwrotnej kolejności: od ogona do głowy. Nie da się tego zrobić tak jak w poprzednim przypadku, bo nie możemy przechodzić listy do tyłu: elementy „nie znają" swych poprzedników. Rozwiązaniem jest rekurencja: z funkcji **main** wywołujemy **drukuj_liste_odwrotnie** dla głowy, czyli obiektu D. Funkcja jednak nie wypisuje informacji od razu, bo w linii ④ następuje wywołanie dla następnika, a następnie dla jego następnika itd.; w końcu dla A (ogon listy) warunek z linii ④ jest nieprawdziwy, zatem funkcja wypisuje informację i wraca do wywołania dla B, wypisuje informacje o B i wraca do wywołania dla *pC itd. Widzimy, że wędrówkę do tyłu zapewnia nam zwijanie stosu podczas powrotów z wywołań funkcji:

```
1 11; 2 22; 3 33; 4 44;
4 44; 3 33; 2 22; 1 11;
```

W powyższym przykładzie niektóre węzły były tworzone na stosie a inne na stercie (przez **new**). Nie jest to dobra praktyka: normalnie wszystkie węzły powinny być tworzone na stercie (patrz rozdz. 13.2 na stronie 252.

Obiekt typu C-struktury może pełnić rolę zarówno argumentu funkcji, jak i wartości zwracanej przez funkcję. Trzeba jednak pamiętać, że przesyłanie obiektów do i z funkcji może być bardzo nieefektywne, jeśli obiekty są duże. W takich przypadkach należy raczej przesyłać wskaźniki (lub referencje) i starać się nie przenosić i nie kopiować samych obiektów. To samo dotyczy oparacji na tablicach obiektów: często lepiej jest operować raczej na tablicach wskaźników do obiektów.

Na przykład w poniższym programie obiekty struktury **Krol** zajmują co najmniej 44 bajty (40 na imie i 4 na ur), a wskaźniki do nich tylko 4 (lub 8):

P102: *krol.cpp* Sortowanie na wskaźnikach

```cpp
#include <iostream>
#include <iomanip>   /* setw */
using namespace std;

struct Krol{
    int   ur;
    char  imie[40];
};
```

```
 9
10 void insertionSort(Krol*[],int);
11
12 int main() {
13     Krol zygmunt   = {1467,               "Zygmunt Stary"},
14          michal    = {1640, "Michal Korybut Wisniowiecki"},
15          wladyslaw = {1351,          "Wladyslaw Jagiello"},
16          anna      = {1523,            "Anna Jagiellonka"},
17          jan       = {1459,               "Jan Olbracht"};
18
19     Krol* krolowie[] = { &zygmunt, &michal, &wladyslaw, ①
20                          &anna,    &jan};
21
22     const int ile = sizeof(krolowie)/sizeof(Krol*);
23
24     cout << "sizeof(Krol ) = " << sizeof(Krol ) << endl;
25     cout << "sizeof(Krol*) = " << sizeof(Krol*) << endl
26                                                << endl;
27     insertionSort(krolowie, ile);
28
29     for ( int i =0; i < ile; i++ )                        ②
30         cout << setw(28) << krolowie[i]->imie
31              << setw(5)  << krolowie[i]->ur << endl;
32 }
33
34 void insertionSort(Krol* a[], int wymiar) {
35     if ( wymiar <= 1 ) return;
36
37     for ( int i = 1 ; i < wymiar ; ++i ) {
38         int j = i;
39         Krol* v = a[i];
40         while ( j >= 1 && v->ur < a[j-1]->ur ) {          ③
41             a[j] = a[j-1];                                ④
42             j--;
43         }
44         a[j] = v;
45     }
46 }
```

Zatem, jeśli chcemy posortować królów według dat ich urodzenia, lepiej posortować tablicę *wskaźników* do obiektów ich reprezentujących, niż same te obiekty. Sortowanie bowiem łączy się z kopiowaniem i przenoszeniem elementów; lepiej jest więc kopiować wskaźniki niż czterdziestocztero-bajtowe obiekty. W linii ① budujemy zatem tablicę wskaźników i przesyłamy ją do funkcji sortującej. W funkcji sortującej, w linii ③, widzimy, że kryterium sortowania jest wartość składowej `ur` obiektu wskazywanego przez wskaźnik będący elementem tablicy. Natomiast przestawiamy i kopiujemy tylko

wskaźniki (jak w linii ④), a nie wskazywane przez nie obiekty. Mimo to cel został osiągnięty: możemy wypisać imiona królów posortowane według roku urodzenia (②):

```
sizeof(Krol ) = 44
sizeof(Krol*) = 8

          Wladyslaw Jagiello 1351
                Jan Olbracht 1459
             Zygmunt Stary 1467
           Anna Jagiellonka 1523
 Michal Korybut Wisniowiecki 1640
```

Użyty tu manipulator **setw** służy jedynie ładniejszemu sformatowaniu wydruku. Funkcja sortująca jest implementacją algorytmu *sortowania przez wstawianie* „bez wartownika".

Można, a czasem trzeba, zadeklarować strukturę (unię, klasę) bez jej definiowania. Nie jest wtedy znany rozmiar obiektów tej struktury, więc jest to typ **niekompletny**. Taka deklaracja nazywa się zapowiadającą. Dzięki niej możemy korzystać z nazwy tego typu wszędzie tam, gdzie nie jest wymagany rozmiar obiektu, a więc np. do definiowania wskaźników do obiektów tego typu. Oczywiście, tylko definiowania, ale nie np. inicjowania, bo żeby taki wskaźnik zainicjować, musiałby istnieć obiekt, a ten możemy utworzyć dopiero po pełnym zdefiniowaniu struktury, kiedy będzie znany rozmiar jej obiektów.

Sytuacja, kiedy taka deklaracja zapowiadająca jest konieczna zachodzi, gdy jedna struktura zawiera, jako pola, wskaźniki do drugiej struktury, a ta druga – wskaźniki do tej pierwszej. Tak więc fragment

```
1   struct AA {
2       BB *b; // NIE
3       // ...
4   };
5
6   struct BB {
7       AA *a;
8       // ...
9   }
```

jest błędny, gdyż w linii drugiej nie wiadomo, co to takiego **BB**. Zmiana kolejności definicji struktur **AA** i **BB** nie pomoże, bo wtedy w definicji struktury **BB** nie wiadomo będzie, co to jest **AA**. Rozwiązaniem jest *deklaracja* struktury: należy przed definicją struktury **AA** zapowiedzieć, że **BB** jest strukturą, której definicja będzie podana później. Od tej chwili można już definiować wskaźniki do obiektów typu **BB**, choć nie można jeszcze definiować samych obiektów. Deklaracja zapowiadająca ma postać

```
struct Nazwa;
```

gdzie **Nazwa** jest nazwą zapowiadanej, ale na razie nie definiowanej, struktury. To samo dotyczy innych typów złożonych, tzn. klas i unii. Tak więc w naszym przykładzie powinniśmy byli napisać

```
struct BB;

struct AA {
    BB *b;
    // ...
};

struct BB {
    AA *a;
    // ...
}
```

i teraz w definicji struktury **AA** wiadomo, że **BB** jest nazwą typu, który będzie zdefiniowany później. Definicja **BB** nie jest jednak potrzebna, aby określić rozmiar przyszłych obiektów klasy **AA**, bo zawierać one będą tylko wskaźniki do obiektów **BB**, a rozmiar wskaźników jest kompilatorowi znany.

Jako przykład rozpatrzmy program:

P103: *zona.cpp* Deklaracje zapowiadające

```
#include <iostream>
#include <cstring>   // strcpy
using namespace std;

struct Husband;
struct Wife;

void prinfo(const Husband*);
void prinfo(const Wife*);

struct Wife {
    Husband *hus;
    char name[20];
} honoratka = { 0, "Honoratka" } ;

struct Husband {
    Wife *wif;
    char name[20];
} antoni  = { 0 } ;

int main() {
    strcpy(antoni.name, "Antoni");          ①

```

```
24      antoni.wif    = &honoratka;
25      honoratka.hus = &antoni;
26
27      Husband zenobiusz  = { 0, "Zenobiusz" };
28      Wife    celestynka = { 0, "Celestynka"};
29
30      prinfo(&antoni);
31      prinfo(&honoratka);
32      prinfo(&zenobiusz);
33      prinfo(&celestynka);
34  }
35
36  void prinfo(const Husband *h) {
37      cout << "Mezczyzna: " << h->name;
38      if ( h->wif )
39          cout << "; zona "
40               << h->wif->name << "\n";          ②
41      else
42          cout << "; (kawaler)\n";
43  }
44
45  void prinfo(const Wife *w) {
46      cout << "Kobieta:   " << w->name;
47      if ( w->hus )
48          cout << "; maz "
49               << (*((*w).hus)).name << "\n";    ③
50      else
51          cout << "; (panna)\n";
52  }
```

Na początku deklarujemy dwie struktury: **Husband** i **Wife**. Dzięki temu możemy zadeklarować funkcje **prinfo** których typem parametru są wskaźniki do obiektów **Husband** i **Wife** (nie możemy na razie zdefiniować tych funkcji, bo w jej treści używać będziemy nazw pól tych struktur, a te nie są na razie znane). Zauważmy, że deklarujemy *dwie* funkcje o tej samej nazwie. Różnią się one jednak wystarczająco typem parametru, więc takie przeciążenie jest legalne.

Dalej definiujemy struktury **Husband** i **Wife**, przy czym mamy tu właśnie do czynienia z przypadkiem, gdy jedna struktura zawiera, jako pole, wskaźnik do drugiej struktury, a ta druga – wskaźnik do tej pierwszej. Zatem deklaracja zapowiadająca była tu niezbędna. Zauważmy też, że nawet po tej deklaracji niemożliwe byłoby, aby struktura **Wife** zawierała jako pole *obiekt* (a nie wskaźnik) typu **Husband**, a struktura **Husband** obiekt typu **Wife**. Oznaczałoby to bowiem, że obiekt typu **Wife** zawierałby obiekt typu **Husband**, który z kolei zawierałby obiekt typu **Wife**, który z kolei . . . i tak *ad infinitum*. Funkcja **strcpy** z linii ① służy do kopiowania C-napisów i poznamy ją dokładniej w rozdz. 17.1 na stronie 355. Dalszy ciąg programu jest już zrozumiały; jego rezultat:

```
Mezczyzna: Antoni; zona Honoratka
Kobieta:   Honoratka; maz Antoni
Mezczyzna: Zenobiusz; (kawaler)
Kobieta:   Celestynka; (panna)
```

Zwróćmy jeszcze uwagę na linie ② i ③. W linii ② wyrażenie

```
h->wif->name
```

oznacza wartość składowej name obiektu wskazywanego przez wskaźnik będący składową wif obiektu wskazywanego przez wskaźnik h. W ten sposób mając wskaźnik do męża wyłuskujemy imię jego żony. Linia ③ zawiera analogiczne wyrażenie

```
(*((*w).hus)).name
```

za pomocą którego mając wskaźnik do żony (zmienna w) wyłuskujemy imię jej męża. Zauważmy, jak notacja ze „strzałką" upraszcza tego typu konstrukcje – w powyższym wyrażeniu wszystkie nawiasy są niezbędne!

13.2 Szablony struktur

Podobnie jak dla funkcji, możemy definiować struktury w postaci *wzorców* (szablonów) zależnych od jednego lub więcej parametrów typu. Wywołując funkcję zadaną w postaci szablonu zazwyczaj nie musieliśmy specyfikować konkretnych typów jakie mają zostać podstawione za parametry typów szablonu: kompilator mógł je sam wywnioskować na podstawie typów argumentów. W przypadku szablonów struktur (klas) tak nie jest — zazwyczaj (choć nie zawsze) musimy ten typ określić. Jeśli, jak w przykładzie poniżej, **Node** jest szablonem struktury zależnym od jednego parametru typu, to tworząc obiekt tego typu musimy określić typ: **Node** jest nazwą szablonu, nazwą konkretnej struktury będzie **Node<int>**, **Node<std::string>**, itd. Ilustruje to poniższy przykład:

P104: *listy.cpp* Prosta lista jednokierunkowa jako szablon

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  template <typename T>
6  struct Node {
7      T data;
8      Node *next;
9  };
10
11 template <typename T>
12 void addFront(Node<T>*& head, T data) {    ①
```

```
13      head = new Node<T>{data, head};
14  }
15
16  template <typename T>
17  void addBack(Node<T>*& head, T data) {
18      if (head == nullptr) addFront(head, data);
19      else {
20          Node<T>* tmp = head;                ②
21          while (tmp->next != nullptr)
22              tmp = tmp->next;
23          tmp->next = new Node<T>{data, nullptr};
24      }
25  }
26
27  template<typename T>
28  void printList(const Node<T>* h) {          ③
29      std::cout << "[ ";
30      while (h != nullptr) {
31          std::cout << h->data << " ";
32          h = h->next;
33      }
34      std::cout << "]\n";
35  }
36
37  template <typename T>
38  void deleteList(Node<T>*& h) {              ④
39      while (h != nullptr) {
40          Node<T>* t = h->next;
41          delete h;
42          h = t;
43      }
44  }
45
46  int main() {
47        // "jakisnapis"s będzie literałem typu std::string
48      using namespace std::literals;          ⑤
49
50      Node<int>* headI{nullptr};
51      addBack(headI,  3);
52      addBack(headI,  4);
53      addFront(headI, 2);
54      addFront(headI, 1);
55      printList(headI);
56      deleteList(headI);
57
58      Node<std::string>* headS{nullptr};
```

```
59      addBack(headS,  "kiery"s);
60      addBack(headS,  "piki"s);
61      addFront(headS, "kara"s);
62      addFront(headS, "trefle"s);
63      printList(headS);
64      deleteList(headS);
65  }
```

Szablon struktury **Node** opisuje pojedynczy element listy jednokierunkowej. Typ danych nie jest określony — jest parametrem typu szablonu (tu oznaczonym przez **T**). Definiujemy też dwie funkcje (znów, w postaci szablonów) które dodają nowe elementy do listy. W funkcji **main** lista reprezentowana jest przez wskaźnik do pierwszego węzła listy (nazywany *głową*, ang. *head*) — wartość **nullptr** tego wskaźnika odpowiada liście pustej.

Funkcja **addFront** pobiera głowę listy i daną: tworzy obiekt zawierający tę daną a jego pole next wskazuje na dotychczasową głowę; nowy węzeł staje się nową głową a dotychczasowa głowa będzie teraz drugim węzłem listy. Zwróćmy uwagę (①), że głowa została przekazana do funkcji przez referencję, aby funkcja mogła ją zmodyfikować.

Podobna jest funkcja **addBack**, która dodaje nowy węzęl na końcu listy. Jeśli lista jest pusta, to wywoływana jest funkcja **addFront**, która radzi sobie z takim przypadkiem. W przeciwnym razie, przebiegamy przez listę w pętli tak, aby zatrzymać się gdy tmp wskazuje na ostatni węzeł – ten, którego pole next jest **nullptr**. To właśnie pole zastępujemy wskaźnikiem na nowoutworzony węzeł, który staje się ostatnim. Zwróćmy uwagę, że aby przebiegać przez listę, utworzyliśmy kopię wskaźnika head (②). Musieliśmy tak zrobić, bo head został przekazany przez referencję (oryginał), więc nie możemy go dowolnie zmieniać!

Inaczej jest w funkcji **printList** (③), gdzie wskaźnik head jest przekazywany przez wartość, a więc jego zmiana wewnątrz funkcji nie wpłynie na oryginał.

Nie możemy też zapomnieć o funkcji **deleteList** (④), która usuwa w pętli wszystkie węzły listy (utworzone wcześniej na stercie za pomocą operatora **new**).

W programie tworzymy listę z danymi typu **int**, a potem **string**. Zauważmy, że napisy, które są danymi w drugiej z tych list, musieliśmy zapisać w postaci literałów typu **string** — bez sufiksu 's' typem ich byłoby **const char***, co nie byłoby zgodne z typem elementów listy; aby skorzystać z tego sufiksu, trzeba „otworzyć" przestrzeń nazw ***std::literals*** (⑤). Wydruk programu to

```
[ 1 2 3 4 ]
[ trefle kara kiery piki ]
```

Inny, podobny, przykład ilustruje jak za pomocą listy można łatwo i efektywnie zaimplementować stos

P105: ***StackTmplt.cpp*** Implementacja stosu poprzez szablony

```
1 #include <iostream>
2 #include <string>
```

```
3
4  template <typename T>
5  struct Node {
6      T     data;
7      Node* next;
8  };
9
10 template <typename T>
11 void push(Node<T>*& head, T d) {
12     head = new Node<T>{d, head};
13 }
14
15 template <typename T>
16 T pop(Node<T>*& head) {
17     T d{head->data};
18     Node<T>* n{head->next};
19     delete head;
20     head = n;
21     return d;
22 }
23
24 template <typename T>
25 bool empty(Node<T>* head) {
26     return head == nullptr;
27 }
28
29 int main() {
30       // "something"s is now a literal of type std::string
31     using namespace std::literals;
32
33     Node<int>* headI{nullptr};
34     Node<std::string>* headS{nullptr};
35     push(headI, 3); push(headS, "3"s);
36     push(headI, 2); push(headS, "2"s);
37     push(headI, 1); push(headS, "1"s);
38                     push(headS, "0"s);
39     while(!empty(headI))
40         std::cout << pop(headI) << " ";
41     std::cout << std::endl;
42
43     while(!empty(headS))
44         std::cout << pop(headS) << " ";
45     std::cout << std::endl;
46 }
```

Dzięki zastosowaniu szablonu możemy tworzyć stosy dla różnych typów (w powyższym

przykładzie są to **int** i **string**). Wydruk programu to

```
1 2 3
0 1 2 3
```

13.3 Unie

Innym, rzadziej stosowanym typem złożonym jest unia. Unie są nieco podobne do struktur — różnica między unią a strukturą polega na tym, że

> w unii wszystkie składowe obiektu umieszczane są pod tym samym adresem. Zatem w każdej chwili dostępna jest tylko jedna składowa.

Zauważmy bowiem, że wpisanie którejś ze składowych zamazuje poprzednią, bo była ona umieszczona w dokładnie tym samym miejscu w pamięci komputera. Wynika z tego, że rozmiar obiektu unii musi być taki, aby mieściła się w nim składowa o największym rozmiarze, ale nie musi być większy, choć może – zależy to od typów i rozmiarów poszczególnych składowych i, niestety, od architektury komputera, w szczególności od stosowanego tzw. wyrównywania (ang. *alignment*).

W poniższym przykładzie

P106: *un.cpp* Unie

```
1  #include <iostream>
2  using namespace std;
3
4  union Bag;
5
6  void wstaw(Bag*,float);
7  void wstaw(Bag*,long double);
8  void infor(const Bag*);
9
10 union Bag {
11     float       liczbaF;
12     long double liczbaLD;
13 } bag ;
14
15 int main() {
16     cout << "      sizeof(float)=" << sizeof(float) << endl;
17     cout << "sizeof(long double)="
18          << sizeof(long double) << endl;
19     cout << "        sizeof(Bag)=" << sizeof(Bag) << endl;
20
21     wstaw(&bag, 3.14F);                    ①
22     infor(&bag);
```

```
23
24      wstaw(&bag, 3.14L);                    ②
25      infor(&bag);
26  }
27
28  void wstaw(Bag *w, float f) {
29      w->liczbaF = f;
30  }
31
32  void wstaw(Bag *w, long double ld) {
33      w->liczbaLD = ld;
34  }
35
36  void infor(const Bag *w) {
37      cout << "\nliczbaF : " << w->liczbaF  << endl;
38      cout <<   "liczbaLD: " << w->liczbaLD << endl;
39  }
```

unia **Bag** przechowuje liczbę typu **float** (4 bajty) *lub* liczbę typu **long double** (12 lub 16 bajtów), ale nie obie jednocześnie. Każde przypisanie do składowej liczbaF obiektu zamaże poprzednią wartość tej składowej, ale również poprzednią wartość składowej liczbaLD, ponieważ obie te składowe zapisywane są w tym samym obszarze pamięci.

W linii ① wpisujemy wartość 3,14 do składowej liczbaF obiektu bag. Funkcja **infor** drukuje obie składowe: liczbaF jest rzeczywiście równa 3,14, a wartość składowej liczbaLD wygląda na przypadkową:

```
        sizeof(float)=4
  sizeof(long double)=16
          sizeof(Bag)=16

liczbaF : 3.14
liczbaLD: 3.93143e-4942

liczbaF : 1.90232e+17
liczbaLD: 3.14
```

Następnie w linii ② wpisujemy wartość 3,14 do składowej liczbaLD tego samego obiektu bag. Po wydrukowaniu liczbaLD jest 3,14, ale składowa liczbaF uległa zamazaniu i teraz ona ma wartość wyglądającą na przypadkową (jest to wartość odpowiadająca układowi bitów w tym obszarze pamięci zinterpretowanemu jako zapis liczby typu **float**).

Zwróćmy uwagą na fakt, że funkcja **wstaw** jest przeciążona i występuje w dwóch wersjach. Właściwa jest wybierana przez kompilator na podstawie typu argumentu. Dlatego wywołując tę funkcję musieliśmy jawnie ten typ zaznaczyć przez dodanie przyrostka 'F' i 'L' do literałów liczbowych (patrz rozdz. 4.3 na stronie 35).

W obu wersjach tej funkcji pierwszym parametrem jest wskaźnik do obiektu typu

Bag, tak aby funkcja mogła zmienić wartość tego obiektu, a nie tylko jego lokalnej kopii, jak byłoby, gdybyśmy przesłali ten obiekt przez wartość. Oczywiście inną możliwością było przesłanie tego argumentu przez referencję.

Z kolei funkcja **infor** korzysta z parametru wskaźnikowego raczej ze względu na efektywność niż z konieczności. Ta funkcja i tak nie zmienia obiektu, więc mogłaby równie dobrze pracować na przekazywanej przez wartość kopii. Ponieważ jednak wybraliśmy jako typ parametru typ wskaźnikowy, co daje funkcji dostęp do oryginału, zaopatrzyliśmy ten parametr w modyfikator **const**, aby zapewnić, że nawet przypadkowo oryginału w tej funkcji nie zmienimy.

Na początku programu wypisujemy rozmiar obiektu typu **Bag**: wynosi on dokładnie 16, tyle ile wynosi rozmiar dłuższej ze składowych (ale oczywiście mniej niż suma rozmiarów składowych).

Bardziej realistyczny przykład znajdujemy w programie następującym:

P107: *unie.cpp* Unia anonimowa, kontrola typów

```
#include <iostream>
#include <cassert>
using namespace std;

struct Bag;
enum Rodzaj {LICZBA, WSKAZNIK, ZNAK};          ①

void wstaw(Bag*,double);
void wstaw(Bag*,int*);
void wstaw(Bag*,char);

void daj(const Bag*,double&);
void daj(const Bag*,int*&);
void daj(const Bag*,char&);

void info(const Bag&);

struct Bag {
    Rodzaj rodzaj;
    union {                                    ②
        double dbl;
        int    *wsk;
        char   znk;
    };
};

int main() {
    Bag bag;
    double x = 3.14, y;
    int    i = 10, *pi = &i;
```

```
31      char   c ='a', b;
32      cout << "sizeof(bag) = " << sizeof(bag)
33           << " bajtow\nAdresy skladowych:\n dbl: "
34           << &bag.dbl << "\n wsk: " <<  &bag.wsk
35           << "\n znk: " << (void*)&bag.znk << endl;
36
37      wstaw(&bag,x);
38      info(bag);
39      daj(&bag,y);
40      cout << "Z funkcji main - y  = " <<    y << endl;
41
42      wstaw(&bag,&i);
43      info(bag);
44      daj(&bag,pi);
45      cout << "Z funkcji main - *pi = " << *pi << endl;
46
47      wstaw(&bag,c);
48      info(bag);
49      daj(&bag,b);
50      cout << "Z funkcji main - b   = " <<   b << endl;
51  }
52
53  void wstaw(Bag *w, double x) {
54      w->rodzaj   = LICZBA;                    ③
55      w->dbl = x;
56  }
57
58  void wstaw(Bag *w, int *pi) {
59      w->rodzaj   = WSKAZNIK;
60      w->wsk = pi;
61  }
62
63  void wstaw(Bag *w, char c) {
64      w->rodzaj   = ZNAK;
65      w->znk = c;
66  }
67
68  void daj(const Bag *w, double& x) {
69      assert(w->rodzaj == LICZBA);
70      x  = w->dbl;
71  }
72
73  void daj(const Bag *w, int*& pi) {
74      assert(w->rodzaj == WSKAZNIK);
75      pi = w->wsk;
76  }
```

```
77
78 void daj(const Bag *w, char& c) {
79     assert(w->rodzaj == ZNAK);                    ④
80     c  = w->znk;
81 }
82
83 void info(const Bag &w) {
84     cout << "\nZ funkcji info - ";
85     switch (w.rodzaj) {
86     case LICZBA:
87         cout << "Liczba:    " << w.dbl     << endl;
88         break;
89     case WSKAZNIK:
90         cout << "Wskaznik: " << *(w.wsk) << endl;
91         break;
92     case ZNAK:
93         cout << "Znak:      " << w.znk     << endl;
94         break;
95     }
96 }
```

Struktura **Bag** ma dwa pola: jedno o nazwie rodzaj typu **Rodzaj**, który to typ jest globalnie zdefiniowanym wyliczeniem (①), a drugie, bez nazwy (*sic!*), które jest typu unii anonimowej zdefiniowanej lokalnie wewnątrz struktury (②). Typ ten jest anonimowy; pomiędzy słowem kluczowym **union** a nawiasem klamrowym rozpoczynającym definicję unii nie podaliśmy bowiem żadnej nazwy. Po jego zdefiniowaniu, zaraz za zamykającym nawiasem klamrowym, *nie* zdefiniowaliśmy żadnego obiektu tego typu. Otóż w takiej sytuacji ujawnia się mało znana cecha unii anonimowych: jeśli zdefiniowana jest unia anonimowa, a *nie* została zdefiniowana żadna zmienna tego typu (zaraz za definicją a przed kończącym średnikiem), to kompilator sam tworzy pojedynczy egzemplarz unii i nazwy jego składowych przenosi do zakresu otaczającego. A zatem nazwy dbl, wsk i znk będą widoczne bezpośrednio w zakresie struktury **Bag**!

Wyliczenie **Rodzaj** składa się z trzech wartości: LICZBA, WSKAZNIK i ZNAK. Będziemy ich używać do określenia typu danej aktualnie przechowywanej w składowej unii. Funkcje **wstaw** są przeciążone. Każda z nich wywoływana jest dla innego typu danej przeznaczonej do wstawienia do składowej unii obiektu typu **Bag**. Kiedy dana jest wstawiana, każda z funkcji dba o to, aby w składowej rodzaj danego obiektu zapisać informację o typie wstawianej danej (jak w linii ③).

Funkcje **daj** do pobierania danej z obiektu typu **Bag** też są przeciążone ze względu na typ danej. Dane pobieramy poprzez drugi argument funkcji przez referencję. W ten sposób zabezpieczamy się przed niewskazanymi w tym przypadku konwersjami. Za każdym razem, gdy pobieramy dane, typ drugiego argumentu decyduje o wyborze funkcji. Każda z funkcji sprawdza za pomocą makra **assert** (np. linia ④), czy typ żądanej danej jest zgodny z typem danej aktualnie przechowywanej w składowej unii. W ten sposób zabezpieczamy się przed błędami podczas sięgania do danych zapisanych w obiektach typu **Bag**.

```
sizeof(bag) = 16 bajtow
Adresy skladowych:
 dbl: 0x7fff0476ee48
 wsk: 0x7fff0476ee48
 znk: 0x7fff0476ee48

Z funkcji info - Liczba:   3.14
Z funkcji main - y  = 3.14

Z funkcji info - Wskaznik: 10
Z funkcji main - *pi = 10

Z funkcji info - Znak:     a
Z funkcji main - b   = a
```

Drukując adresy składowych unii zawartych w obiekcie bag na początku programu, przekonujemy się, że rzeczywiście wszystkie one są zapisane pod tym samym adresem.

— 14 —

Klasy (I)

W tym rozdziale rozpoczniemy dyskusję klas: podstawowego elementu programowania obiektowego.

PODROZDZIAŁY:

14.1 Wstęp

C++ jest językiem obiektowym, a więc oczywiście umożliwia definiowanie nowych typów danych bogatszych niż proste C-struktury, które zawierały wyłącznie dane, bez opisu operacji, jakie na tych danych można wykonywać — operacje takie trzeba w C definiować za pomocą globalnych funkcji.

Klasa jest uogólnieniem typu: w zasadzie dobrze zdefiniowana (tzw. pierwszorzędowa) klasa zachowuje się dokładnie jak typ wbudowany (*nie* jest tak w Javie). Aby ta analogia była pełna, projektanci C++ zadbali nawet o to, żeby „prawdziwe" typy wbudowane mogły być używane zgodnie ze składnią właściwą dla klas definiowanych przez użytkownika; tak na przykład, dopuszczalne jest definiowanie wraz z inicjalizacją zmiennych typów wbudowanych, jak **int** albo **double**, poprzez wywołanie „konstruktora":

```
int k(5);
```

albo

```
double *p = new double(6.5);
```

Tak naprawdę żadnego konstruktora nie ma, bo **int** i **double** są typami wbudowanymi, ale składnia jest taka, jakby tworzone zmienne były obiektami klasy.

Składnikami klasy są w zasadzie pola i funkcje, zwane razem składowymi klasy. Ale klasa stanowi też osobną przestrzeń nazw: wewnątrz klasy można definiować nowe typy, jak i nowe aliasy dla nazw typów za pomocą instrukcji **typedef**. Nazwy tak zdefiniowanych typów czy aliasów należeć będą do zasięgu klasy: można się do nich odwołać z zewnątrz poprzez nazwę kwalifikowaną z zastosowaniem operatora zasięgu '`::`'.

Definicję klasy piszemy zwykle na zewnątrz innych klas i funkcji (nie jest to jednak wymaganie języka: można, choć rzadko bywa to przydatne, definiować klasę wewnątrz funkcji).

Sama definicja ma postać

```
class Klasa {
    // ... pola, metody...
};
```

lub

```
struct Klasa {
    // ... pola, metody...
};
```

Nie należy zapominać o średniku kończącym definicję klasy! Nowy typ jest już zdefiniowany po napotkaniu zamykającego nawiasu klamrowego, zatem za nawiasem, a przed średnikiem można umieścić definicje obiektów właśnie zdefiniowanej klasy; na przykład po

```
struct Klasa {
    // ...
} x, y, z;
```

zdefiniowana byłaby klasa **Klasa** i utworzone trzy obiekty tej klasy o nazwach `x`, `y` i `z` (pod warunkiem, że istnieje w klasie/strukturze publiczny konstruktor domyślny, co za chwilę omówimy bardziej szczegółowo).

14.2 Dostępność składowych

Wszystkie składowe klasy są widoczne wewnątrz klasy. Oznacza to, że mają do nich dostęp funkcje (metody) zadeklarowane jako funkcje składowe danej klasy. Mogą jednak mieć różny poziom dostępności z zewnątrz, a więc z funkcji które nie sa składowymi danej klasy: poziom dostępności jest określany jednym ze słów kluczowych: **public**, **private**, lub **protected**.. W odróżnieniu od Javy, *nie* stawia się go jednak przed każdą ze składowych osobno, a definicję klasy dzieli się na tzw. **sekcje**: każda sekcja rozpoczyna się od jednego z tych słów kluczowych z następującym po nim dwukropkiem. Sekcja rozciąga się do końca definicji klasy lub do rozpoczęcia innej sekcji, na przykład:

```
1     class Klasa {
2         int s1;
3     public:
4         int s2;
5         double d2;
6     private:
7         double s3;
8         void fun3(int,double);
9     public:
10        int s4;
11        char c4;
12    };
```

Zauważmy, że pole s1 zostało zdefiniowane przed pojawieniem się jakiegokolwiek specyfikatora poziomu dostępności. Przyjmuje się wtedy dostępność domyślną, według następującej zasady:

> Klasy można definiować w C++ za pomocą słowa kluczowego **class** lub **struct**. Jeśli użyliśmy słowa **class**, to domyślnie składowe są prywatne (**private**), jeśli użyliśmy słowa **struct**, to domyślnie składowe są publiczne (**public**). Innych różnic między klasami i strukturami w C++ nie ma!

Tak więc w powyższym przykładzie występują cztery sekcje: pierwsz i trzecia **private** a druga i czwarta **public**. Tak więc prywatne są pola s1, s3 i funkcja (metoda) **fun3**, natomiast publiczne s2, d2, s4 i c4. Jak widzimy, w definicji klasy (struktury) może być wiele sekcji publicznych czy prywatnych.

Pod względem dostępności składowe dzielą się zatem na:

- publiczne (**public**), których nazwy mogą być używane we wszystkich miejscach programu, gdzie widoczna jest definicja klasy;
- prywatne (**private**), których nazwy mogą być używane tylko przez funkcje, które same są składowymi tej samej klasy, lub funkcje z daną klasą zaprzyjaźnione (o czym dalej);
- chronione (**protected**), których nazwy mogą być używane tylko przez funkcje, które same są składowymi tej samej klasy, funkcje z daną klasą zaprzyjaźnione, a także funkcje składowe i zaprzyjaźnione klas pochodnych (dziedziczących z) danej klasy.

Te same zasady dotyczą nie tylko składowych, ale i innych nazw (jak na przykład nazw typów czy aliasów definiowanych za pomocą **typedef**) wprowadzanych w zakresie klasy. Wszystkie one należą do przestrzeni nazw klasy, tak więc spoza klasy trzeba się do nich odwoływać albo poprzez obiekty tej klasy (składowe niestatyczne), albo poprzez kwalifikację nazw nazwą klasy przy użyciu operatora zakresu '::' (składowe statyczne, nazwy typów zdefiniowanych w zakresie klasy, aliasy zdefiniowane za pomocą **typedef**).

Zauważmy, że ograniczenie dostępności dotyczy *nazw*, a nie na przykład obszarów pamięci zajmowanych przez składowe prywatne.

Rozpatrzmy króciutki program

P108: ***pozdro.cpp*** Dostępność składowych

```
#include <iostream>
using namespace std;

class Pozdro {
    int k1;                                  ①
public:
    enum Kraj { PL, DE, FR };
    int k2;                                  ②
    void fun(Kraj kraj) {
        switch (kraj) {
            case PL:
                cout << "Dzien dobry\n"; k1 = 1; break;
            case DE:
                cout << "Guten Tag\n";    k1 = 2; break;
            case FR:
                cout << "Bonjour\n";      k1 = 3; break;
        }
    }
};

int main() {
    Pozdro dd;                               ③

    dd.fun(Pozdro::DE);                      ④

    int *pk1 = &dd.k2 - 1;                   ⑤

    cout << "sizeof(dd) = " << sizeof(dd) << endl;    ⑥
    cout << "dd.k1      = " << *pk1       << endl;    ⑦
}
```

Definiujemy tu klasę **Pozdro**. Pole k1 jest prywatne (①), pole k2 (②), definicja wyliczenia **Kraj** i metoda (funkcja) **fun** są publiczne. W programie głównym tworzymy obiekt dd tej klasy (③). Zauważmy, że składnia jest taka jak przy tworzeniu zwykłej zmiennej typu **int** — nazwa typu i nazwa wprowadzanej zmiennej.

W linii ④ wywołujemy na jego rzecz metodę **fun** kwalifikując nazwę funkcji nazwą obiektu — bardziej szczegółowo omówimy tę kwestię poniżej. Argument jest typu **Kraj**, ale ten typ (wyliczeniowy) został zdefiniowany w klasie **Pozdro**; jego nazwa nie jest widoczna poza zasięgiem klasy. Nie możemy więc użyć po prostu nazwy DE oznaczającej jeden z elementów wyliczenia; musimy poprzedzić ją nazwą klasy

Pozdro i operatorem zasięgu '`::`' (czyli **kwalifikować**). Piszemy zatem Pozdro::DE, aby poinformować kompilator, że chodzi nam o nazwę DE z przestrzeni nazw (w tym przypadku klasy) **Pozdro**.

Metoda **fun** wywołana z argumentem Pozdro::DE wpisała do prywatnej składowej k1 obiektu na rzecz którego została wywołana wartość 2. Oczywiście mogła to zrobić, gdyż wszystkie metody (funkcje) klasy „widzą" wszystkie nazwy zdefiniowane w zasięgu klasy, niezależnie od tego, czy są one publiczne czy nie (a samą funkcję mogliśmy wywołać, bo jest ona publiczna). Ale jak wewnątrz funkcji **main** przekonać się, że rzeczywiście wartość składowej dd.k1 wynosi 2, skoro ta składowa jest prywatna i użycie nazwy k1 poza klasą spowoduje błąd? W linii ⑥ drukujemy rozmiar obiektu dd:

```
Guten Tag
sizeof(dd) = 8
dd.k1 = 2
```

Wynosi on 8 bajtów, co przekonuje nas, że rzeczywiście obiekt ten zawiera dwie składowe typu **int** i nic więcej (zakładamy, że uszeregowane są w kolejności odpowiadającej kolejności ich deklaracji w definicji klasy). W linii ⑤ pobieramy zatem adres składowej k2, do której mamy dostęp, bo jest publiczna (i wystarczy odnieść się do niej poprzez nazwę obiektu). Od niego odejmujemy 1, co zgodnie z arytmetyką wskaźników powinno dać nam adres poprzedniej składowej, czyli prywatnej składowej k1 (zmienna wskaźnikowa pk1). Wydrukowanie wartości zmiennej spod tego adresu (⑦) daje 2, co przekonuje nas, że jest to istotnie ta prywatna składowa, o którą nam chodziło.

Jak więc widać, to nie sama składowa jest chroniona za pomocą kwalifikatora **private**; chroniona jest *nazwa* składowej. Jeśli potrafimy „dobrać się" do tej składowej bez użycia jej nazwy, to język na to pozwala. Wprowadzenie podziału na składowe publiczne i prywatne jest raczej elementem wspierającym tworzenie przejrzystszego i mniej podatnego na błędy kodu, niż sposobem na crackerów. Sztuczne omijanie zabezpieczeń, jakie ten podział daje prowadzi do kodu niezrozumiałego, niebezpiecznego i niemożliwego do pielęgnacji.

Dostępność wszystkich składowych klasy przez jej składowe funkcje (statyczne, niestatyczne, konstruktory, destruktor) dotyczy klasy, a nie konkretnych obiektów. Na przykład w poniższym programie

P109: *acc.cpp* Dostępność składowych innych obiektów klasy

```
#include <iostream>
using namespace std;

class Vector {
    double x, y, z;
public:
    void set(double xx, double yy, double zz) {
        x = xx;
        y = yy;
        z = zz;
```

```
11      }
12      double dot_product(const Vector& w) {
13          return x*w.x + y*w.y + z*w.z;
14      }
15  };
16
17  int main() {
18      Vector w1, w2;
19      w1.set(1, 1, 2);
20      w2.set(1,-1, 2);
21      cout << "w1*w2 = " << w1.dot_product(w2) << endl;   ①
22  }
```

funkcja **dot_product** (iloczyn skalarny), jako składowa klasy, ma dostęp do prywatnych składowych x, y, z zarówno obiektu w1, na rzecz którego została wywołana (i do których odnosi się poprzez ich niekwalifikowane nazwy), jak i do składowych obiektu w2, przesłanego przez referencję jako argument wywołania (①). Świadczy o tym wydruk programu:

```
w1*w2 = 4
```

14.3 Pola klasy

Pola klasy definiują dane, z jakich składać się będzie każdy obiekt klasy. Danymi tymi mogą być wielkości dowolnego typu wbudowanego lub zdefiniowanego w programie przez samego programistę lub autora biblioteki, z której program korzysta. Niestatyczne pole klasy *nie* może być typu, który dana klasa właśnie definiuje (z powodów o których mówiliśmy przy omawianiu struktur), natomiast może mieć typ wskaźnika do obiektu tejże klasy.

Deklaracje pól mają postać definicji zmiennych, ale często *nie* zawierają inicjalizacji:

```
class Klasa {
    int    k1, k2;
    double x,  y;
    // ...
};
```

było i jest prawidłowe, ale

```
class Klasa {
    int    k1, k2 = 1;
    double x = 1.5 , y = 3.5; // teraz OK
    // ...
};
```

przed wersją C++11 powodowało błąd kompilacji, gdyż inicjowanie w definicji pola było niedozwolone (z pewnymi wyjątkami). Obecnie taka inicjalizacja *jest* już jednak dozwolona. Zadeklarowanie pola (niestatycznego) oznacza, że w każdym obiekcie definiowanej klasy będzie utworzona zmienna o nazwie i typie określonym w deklaracji. Sama definicja klasy nie powoduje utworzenia żadnych obiektów.

Pola można deklarować wewnątrz klasy w dowolnej kolejności i miejscu — przed lub po metodach; ich zakresem jest cała klasa, a nie tylko fragment następujący leksykalnie po definicji. Kolejność definicji pól ma jednak znaczenie podczas inicjowania i niszczenia (destrukcji) — pola są inicjowane w kolejności takiej, w jakiej były zadeklarowane, a usuwane w kolejności odwrotnej.

Szczególny rodzaj pól to pola statyczne. Deklaruje się je ze specyfikatorem **static**. Oznacza on, że będzie utworzona tylko jedna zmienna o tej nazwie i że nazwa ta będzie leżeć w zasięgu klasy. Zmienna ta jest niezależna od obiektów (zmiennych) klasy; można z niej korzystać nawet, jeśli żaden obiekt tej klasy nie został utworzony.

Sama deklaracja pola statycznego w klasie nie tworzy jednak tej zmiennej (bo właśnie jest tylko deklaracją, a nie definicją)! Zmienne odpowiadające polom niestatycznym będą tworzone podczas tworzenia obiektów klasy. Kiedy w takim razie będzie utworzona zmienna statyczna klasy? Utworzyć ją, czyli *zdefiniować*, trzeba *na zewnątrz* klasy. Ponieważ jej nazwa należy do zasięgu klasy, poza klasą trzeba się do tej zmiennej odnieść poprzez jej nazwę kwalifikowaną, za pomocą operatora zasięgu klasy, '`::`'. W definicji poza klasą *nie* powtarzamy już specyfikatora **static**. Definiując zmienną statyczną klasy można (choć nie trzeba) ją zainicjować. Na przykład:

```
1     class D {
2         static int k1;  // deklaracje
3         static int k2;
4     };
5     int D::k1;       // definicja; inicjowanie zerem
6     int D::k2 = 7;   // definicja z inicjowaniem
```

W liniach 2 i 3, w definicji klasy, deklarujemy `k1` i `k2` jako statyczne składowe klasy **D**. Natomiast poza definicją klasy, w liniach 5 i 6, zmienne te *definiujemy*, czyli tworzymy, przydzielając im pamięć. Jeśli opuściliśmy inicjator, składowe statyczne *są* inicjowane wartością 0 (zero) odpowiedniego typu.

Wyjątkowo, składowe statyczne mogą być definiowane i inicjalizowane bezpośrednio w punkcie deklaracji, wewnątrz klasy, ale tylko jeśli są ustalone (**const**) i typu całkowitego

```
class D {
    static const int k1, k2 = 2;
    // ...
};
const int D::k1 = 1;
```

Tu `k2` jest tworzone i inicjalizowane wewnątrz definicji klasy **D**, natomiast `k1` tylko deklarowane — definicja z ostatniej linii jest zatem konieczna, i to wraz z inicjalizacją, bo stałe, jak pamiętamy, muszą być inicjalizowane w momencie ich definiowania.

Zmienne statyczne są tworzone (i inicjowane) po załadowaniu programu do pamięci, ale *przed* rozpoczęciem wykonywania funkcji **main** (jak zmienne globalne).

Do zmiennych statycznych klasy można się odwoływać w programie poprzez ich pełną nazwę kwalifikowaną (D::k1) lub tak jak do normalnej składowej, czyli poprzez nazwę dowolnego obiektu tej klasy i operator wyboru składowej (kropka lub „strzałka", w zależności od tego, czy nazwa była nazwą obiektu czy wskaźnika do niego). Oczywiście, ponieważ istnieje tylko jeden egzemplarz zmiennej statycznej, nie ma wtedy znaczenia, którego obiektu użyjemy.

Ponieważ składowa statyczna istnieje w jednym egzemplarzu i nie wchodzi fizycznie w skład obiektów klasy, więc składową taką *może* być obiekt tejże klasy. Na przykład klasa (struktura) **Punkt** z poniższego programu

P110: ***statskl.cpp*** Pola statyczne

```
#include <iostream>
#include <cmath>    // sqrt
using namespace std;

struct Punkt {
    double x, y;
    static Punkt srodek;   ①
};
Punkt Punkt::srodek;       ②

void ustal_srodek(double,double);
double odl_od_srodka(const Punkt&);

int main() {
    Punkt P = {3, 4};      ③
    cout << "Punkt P = (" << P.x << "," << P.y << ")\n";

    ustal_srodek(0,0);
    cout << "Odl. P-srodek: " << odl_od_srodka(P) << endl;

    ustal_srodek(9,-4);
    cout << "Odl. P-srodek: " << odl_od_srodka(P) << endl;
}

void ustal_srodek(double xx,double yy) {
    Punkt::srodek.x = xx; ④
    Punkt::srodek.y = yy; ⑤
    cout << "Srodek w p-cie (" << xx << "," << yy << ")\n";
}

double odl_od_srodka(const Punkt& p) {
    return
```

```
33          sqrt((p.x-Punkt::srodek.x)*(p.x-Punkt::srodek.x) +
34               (p.y-Punkt::srodek.y)*(p.y-Punkt::srodek.y));
35 }
```

zawiera niestatyczne pola x i y (współrzędne punktu, zależne od obiektu) i jako pole statyczne obiekt srodek tejże klasy **Punkt** — może on na przykład reprezentować aktualny punkt odniesienia, względem którego mierzymy odległości innych punktów. W linii ② definiujemy tę składową statyczną, zadeklarowaną w definicji klasy (①). Zauważmy, że nie powtarzamy tu specyfikatora **static**.

W programie głównym tworzymy obiekt p klasy **Punkt** (③) i nadajemy wartości jego składowym x i y. Ta składnia inicjalizacji jest tu prawidłowa, bo struktura co prawda nie jest czystą C-strukturą, ale jest *agregatem*, o którym będzie mowa dalej.

Za pomocą funkcji **ustal_srodek** ustalamy punkt odniesienia na $(0, 0)$. Zauważmy, że w treści tej funkcji, niebędącej składową klasy, odwołujemy się do statycznej składowej srodek klasy **Punkt**, a zatem używamy jej pełnej, kwalifikowanej nazwy z „czterokropkiem" — linie ④ i ⑤. Składowa ta sama jest obiektem klasy **Punkt**, więc ma (niestatyczne) składowe x i y.

Funkcja **odl_od_srodka** oblicza odległość między dowolnym punktem przekazanym jej przez referencję a aktualnym punktem odniesienia, który jest opisywany statyczną składową klasy

```
Punkt P = (3,4)
Srodek w p-cie (0,0)
Odl. P-srodek: 5
Srodek w p-cie (9,-4)
Odl. P-srodek: 10
```

Pola statyczne stosuje się często, gdy potrzebne są przeróżnego rodzaju liczniki obiektów, parametry wspólne dla wszystkich obiektów klasy (na przykład cena, data wykonania programu, aktualny kurs dolara, deskryptor pliku), flagi, za pomocą których obiekty „porozumiewają się" między sobą itd.

14.4 Metody

Tak jak pola klasy opisują dane, które zawarte będą w każdym obiekcie klasy, tak **metody** definiują zbiór operacji, jakie na tych danych będzie można wykonywać. Metody są wyrażone w języku jako niestatyczne funkcje o pewnych szczególnych własnościach (co to znaczy *niestatyczne*, wyjaśnimy za chwilę).

Deklaracja metody ma postać deklaracji funkcji, tyle że zawarta jest wewnątrz definicji klasy. Tak jak dla zwykłych funkcji, deklaracja może być połączona z definicją. Można też, z podobnym skutkiem, wewnątrz klasy tylko metodę zadeklarować, a zdefiniować ją już poza klasą, w definicji odwołując się do niej poprzez nazwę kwalifikowaną operatorem zakresu klasy (np. **Klasa::**), bo przecież mogłoby istnieć wiele niezwiązanych ze sobą metod o tej samej nazwie w różnych klasach. Istnieje pewna różnica między tymi sposobami:

Jeśli funkcja (metoda, funkcja statyczna, konstruktor, destruktor) jest *definiowana* wewnątrz klasy, to domyślnie przyjmuje się dla niej modyfikator **inline**.

Tak więc, jeśli metodę definiujemy wewnątrz klasy, to kompilator będzie próbował ją rozwijać. Jak pamiętamy (patrz rozdz. 11.10 na stronie 179), *nie* oznacza to, że funkcje te będą na pewno rzeczywiście rozwinięte — kompilator może uznać to zadanie za zbyt trudne. Jeśli metoda (ogólnie funkcja) jest w klasie tylko zadeklarowana, natomiast zdefiniowana jest poza klasą, to domyślnie nie będzie rozwijana, chyba że jawnie tego zażądamy w definicji (ale nie w deklaracji — rozwijanie nie jest cechą należącą do *kontraktu* między programistą a przyszłym użytkownikiem, który nie musi wiedzieć, czy dana funkcja jest czy nie jest rozwijana).

Z kolei argumenty domyślne, jeśli istnieją, określamy wtedy w deklaracji, ale, stosownie do ogólnych reguł, *nie* powtarzamy ich w definicji (argumenty domyślne, oczywiście, *należą* do kontraktu). Na przykład program ***acc.cpp*** (str. 267) moglibyśmy przekształcić do formy następującej, przenosząc definicje funkcji poza klasę:

P111: ***accout.cpp*** Definiowanie metod poza klasą

```
#include <iostream>
using namespace std;

class Vector {
    double x, y, z;

public:
    void set(double xx=0, double yy=0, double zz=0);
    double dot_product(const Vector& w);
};

void Vector::set(double xx, double yy, double zz) {
    x = xx;
    y = yy;
    z = zz;
}
double Vector::dot_product(const Vector& w) {
    return x*w.x + y*w.y + z*w.z;
}

int main() {
    Vector w1, w2;
    w1.set(1, 1, 2);
    w2.set(1,-1, 2);
    cout << "w1*w2 = " << w1.dot_product(w2) << endl;
}
```

Zwróćmy uwagę, że poza klasą, w definicjach jej metod, posługujemy się ich nazwami *kwalifikowanymi* **Vector::set** i **Vector::dot_product**. Dla funkcji **Vector::set** zadeklarowaliśmy argumenty domyślne; jak wiemy, w definicji już ich powtórzyć *nie wolno*.

Metody, a więc funkcje składowe niestatyczne i nie będące konstruktorem, wywoływane są zawsze „na rzecz" konkretnego, istniejącego wcześniej obiektu klasy, której są składowymi. Wywołanie metody **fun** spoza klasy, a więc z funkcji która sama nie jest funkcją składową tej samej klasy, przybiera zatem jedną z postaci

```
a.fun()
pa->fun()
```

gdzie a jest zmienną będącą obiektem tej klasy lub referencją do obiektu tej klasy, a pa jest wskaźnikiem wskazującym obiekt tej klasy. Dopóki nie rozpatrujemy dziedziczenia i polimorfizmu, wszystkie te formy wywołania („przez obiekt", „przez referencję" i „przez wskaźnik") są równoważne.

Można sobie wyobrażać (i tak jest najczęściej w rzeczywistości), że metody mają ukryty parametr, będący typu *ustalony wskaźnik do obiektu klasy* i że podczas wywołania argumentem związanym z tym parametrem jest wskaźnik do (adres) obiektu, na rzecz którego następuje wywołanie (w niektórych językach, jak na przykład w Pythonie, parametr taki wcale nie jest ukryty i musi być jawnie uwzględniony na liście parametrów metod). Jakby nie było to zaimplementowane, wewnątrz metody wskaźnik do tego obiektu, na rzecz którego metoda została wywołana, jest znany i nazywa się **this**.

W C++ **this** jest nazwą ustalonego *wskaźnika* do obiektu na rzecz którego wykonywana jest metoda. Nazwą tego obiektu jest zatem *this. Słowo kluczowe **this** może być użyte wyłącznie w metodach (niestatycznych funkcjach składowych), konstruktorach i destruktorze klasy.

Wskaźnik **this** jest *niemodyfikowalny*. Zatem przypisanie na this nie byłoby legalne. Natomiast obiekt, na który ten wskaźnik wskazuje, „ten obiekt", jest modyfikowalny: przypisanie na *this jest dopuszczalne.

Jeśli odwołujemy się do składowej tego obiektu, na rzecz którego została wywołana funkcja, która jest metodą, konstruktorem albo destruktorem klasy, to w jej ciele można to zrobić bez specyfikowania obiektu: przyjmuje się wtedy, że jest to „ten" obiekt, a więc *this. Do składowej sklad obiektu metoda (konstruktor, destruktor) może też oczywiście odwoływać się poprzez pełną nazwę tego obiektu, czyli używając notacji `this->sklad` lub `(*this).sklad`. Taka forma jest konieczna, jeśli wewnątrz funkcji zadeklarowaliśmy zmienną o tej samej nazwie, co któraś ze składowych klasy; nazwa niekwalifikowana odnosi się wtedy do tej zmiennej lokalnej, a nie do składowej. Pamiętać trzeba, że zmiennymi lokalnymi są też zmienne skojarzone z parametrami funkcji.

Przyjrzyjmy się programowi:

P112: ***met.cpp*** Wskaźnik *this* i metody klasy

```
#include <iostream>
using namespace std;

class Number {
    double x;
public:
    void    set(double);
    Number& add(double);
    Number& subtract(double);
    Number* multiply(double);
    Number* divide(double);
    void    info(const char*);
};
void Number::set(double x) {
    this->x = x;                                                    ①
}
inline Number& Number::add(double x) {
    this->x += x;
    return *this;
}
inline Number& Number::subtract(double x) {
    this->x -= x;
    return *this;
}
inline Number* Number::multiply(double x) {
    this->x *= x;
    return  this;
}
inline Number* Number::divide(double x) {
    this->x /= x;
    return  this;
}
void Number::info(const char* s) {
    cout << s << " " << x << endl;
}

int main() {
    Number L;

    L.set(10);
    L.info("set                  :");
    L.add(5).subtract(7).info("add + subtract    :");  ②
    L.multiply(2)->divide(4)->                          ③
                   info("multiply + divide :");
```

```
45 }
```

Wszystkie metody klasy **Number** zostały zadeklarowane w klasie, ale zdefiniowane poza klasą (z modyfikatorem **inline**). Zauważmy, że w funkcji **set** nazwa parametru funkcji koliduje z nazwą składowej klasy: aby się odnieść do składowej `x`, musimy zatem użyć wskaźnika **this** (①): `this->x` po lewej stronie odnosi się do składowej `x` *tego* obiektu, podczas gdy samo `x` po prawej do lokalnej zmiennej odpowiadającej parametrowi funkcji.

Funkcje **add** i **subtract** zwracają przez referencję ten obiekt, na rzecz którego zostały wywołane. Typem zwracanym jest więc **Number&**, a w instrukcji **return** zwracany jest obiekt *this. Tak więc na przykład wyrażenie `L.add(5)` (②) jest po prostu nazwą obiektu `L` *po* wykonaniu na jego rzecz metody **add** z argumentem 5. Ponieważ jest to nazwa obiektu klasy **Number**, więc na jego rzecz można z kolei wywołać metodę **subtract** i, na tej samej zasadzie, **info**.

Podobne kaskadowe wywołania można stosować dla funkcji **multiply** i **divide**. Zwracają one **this**, czyli *wskaźnik* do obiektu, na rzecz którego zostały wywołane (a zatem są typu **Number***); teraz zatem trzeba używać notacji „ze strzałką” (linia ③). Wynik

```
set                : 10
add + subtract     : 8
multiply + divide : 4
```

jest oczywiście zgodny z oczekiwaniami. W całym programie tworzony jest tylko jeden obiekt klasy **Number**, do którego odnosimy się poprzez jego nazwę, referencję do niego lub wskaźniki.

14.5 Statyczne funkcje składowe

Funkcje składowe klasy mogą być zadeklarowane jako statyczne. Takie funkcje można wywołać nawet wtedy, gdy nie istnieje jeszcze żaden obiekt klasy. Do ich nazwy z zewnątrz klasy odwołujemy się poprzez nazwę klasy za pomocą operatora zasięgu, czyli „czterokropka” (`'::'`), albo za pomocą operatora wyboru składowej (kropka) — nie ma wtedy znaczenia, jakiego obiektu tej klasy użyjemy. Ponieważ funkcja statyczna *nie* jest wywoływana na rzecz obiektu, ale jak funkcja globalna, nie można w niej odwoływać się do **this** ani do żadnych składowych niestatycznych — te bowiem istnieją tylko wewnątrz konkretnych obiektów i w każdym z nich mogą być różne. Można natomiast w funkcjach statycznych klasy odwoływać się do składowych statycznych tej klasy: innych funkcji statycznych i zmiennych klasowych (określanych przez statyczne pola klasy).

Funkcje statyczne klasy od funkcji zadeklarowanych w zasięgu globalnym różni to, że należą do zakresu (przestrzeni nazw) klasy. Mają zatem bezpośredni dostęp do nazw z zakresu tej klasy (również prywatnych).

Na przykład program ***acc.cpp*** (str. 267) może być przepisany w następujący sposób, tym razem z funkcją obliczającą iloczyn skalarny jako funkcją statyczną:

P113: ***memstat.cpp*** Składowe funkcje statyczne

```
#include <iostream>
using namespace std;

class Vector {
    double x, y, z;
public:
    void set(double xx = 0, double yy = 0, double zz =0) {
        x = xx;
        y = yy;
        z = zz;
    }
    static double dot_product(const Vector& w1,
                              const Vector& w2) {
        return w1.x * w2.x + w1.y * w2.y + w1.z * w2.z;
    }
};

int main() {
    Vector w1, w2, ww;
    w1.set(1, 1, 2);
    w2.set(1,-1, 2);

    cout << "w1*w2 = "
         << Vector::dot_product(w1, w2) << endl;   ①

    cout << "w1*w2 = "
         << ww.dot_product(w1, w2)      << endl;   ②
}
```

Zauważmy wywołanie z linii ① poprzez kwalifikację nazwą klasy i wywołanie z linii ②, gdzie funkcja jest wywoływana formalnie na rzecz obiektu ww, choć w rzeczywistości żadna informacja o tym obiekcie nie zostanie do funkcji przekazana (nie był on nawet sensownie zainicjowany!). Obiekt ww został użyty wyłącznie do tego, aby określić klasę w której zasięgu określona jest nazwa funkcji; równie dobrze mogliśmy tu użyć dowolnego innego obiektu klasy **Vector**. Cała informacja o wektorach które mają być pomnożone jest przekazywana przez argumenty.

14.6 Konstruktory

W zasadzie każda klasa musi mieć **konstruktor**: funkcję wywoływaną podczas kreowania nowego obiektu danej klasy. Jeśli sami żadnego konstruktora nie zdefiniowaliśmy, to możemy zakładać, że system dostarczy własny konstuktor **domyślny**, który jest publiczny, nie ma żadnych parametrów i nie wykonuje żadnych czynności.

Jeśli *jakikolwiek* konstruktor został w klasie zdefiniowany, to konstruktor domyślny *nie* będzie kreowany automatycznie. Konstruktor jest konstruktorem domyślnym wtedy i tylko wtedy gdy może być wywołany bez żadnych argumentów.

Nie znaczy to, że konstruktor domyślny koniecznie musi być zdefiniowany jako funkcja bezparametrowa; może mieć dowolną liczbę parametrów, ale jeśli je ma, to wszystkie muszą mieć zdefiniowane wartości domyślne (patrz rozdz. 11.4 na str. 163).

W klasie może oczywiście istnieć tylko jeden konstruktor domyślny. Jeśli został wygenerowany automatycznie, to, jak już mówiliśmy, jego ciało jest puste (czyli „nic nie robi"), a jego dostępność jest typu **public**.

Konstruktor (niekoniecznie domyślny) jest wywoływany automatycznie wyłącznie podczas tworzenia obiektu (ściśle mówiąc, na zakończenie tego procesu); nie można go wywołać „ręcznie", jak normalnej metody, na rzecz obiektu już istniejącego.

Nazwa konstruktora musi być identyczna z nazwą klasy; wielkość liter, oczywiście, *jest* przy tym ważna.

Konstruktor nie może być funkcją rezultatową; mimo to deklarując/definiując go *nie wolno* podać jako typu zwracanego **void** — typu zwracanego nie podajemy w ogóle.

Konstruktory, jak inne funkcje, mogą mieć parametry domyślne. Można nawet powiedzieć, że parametry domyślne są właśnie w przypadku konstruktorów szczególnie przydatne i często stosowane.

Konstruktory mogą też być przeładowywane (przeciążane), jak normalne funkcje. Wszystkie mają w danej klasie taką samą nazwę: taką jak nazwa tej klasy. Jeśli jest ich kilka, to podczas tworzenia obiektu wybierany jest ten „najbardziej pasujący", zgodnie z zasadami obowiązującymi dla wywoływania funkcji przeciążonych (patrz rozdz. 11.11 na str. 181).

Zauważmy, że jeśli zmienne obiektowe definiujemy w zakresie globalnym, a więc poza definicjami funkcji, albo jeśli takie zmienne są polami statycznymi klasy, to konstruktory dla tych obiektów zostaną wywołane w czasie tworzenia tych obiektów, a więc jeszcze *przed* wywołaniem funkcji **main**! Obiekty globalne są tworzone w kolejności, w jakiej są w programie definiowane, a zatem w takiej też kolejności będą wywoływane konstruktory.

W czasie wykonywania konstruktora obiekt *jest już* skonstruowany, to znaczy został fizycznie utworzony w pamięci, jak również, co bardzo ważne, zostały już utworzone wszystkie jego składowe opisane polami klasy. Składowe te tworzone są w kolejności takiej, w jakiej zostały zadeklarowane w definicji klasy/struktury.

Ponieważ obiekt w zasadzie już istnieje, gdy konstruktor rozpoczyna działanie, ma sens posługiwanie się wewnątrz konstruktora wskaźnikiem **this** wskazującym na tworzony obiekt. Tym niemniej, do obiektu można się na razie odwoływać tylko z wnętrza konstruktora. Dopóki konstruktor nie zakończy swej pracy, tworzony obiekt jest niedostępny z zewnątrz; ma to znaczenie na przykład w programach wielowątkowych.

14.7 Destruktory

Prócz konstruktorów można również zdefiniować w klasie **destruktor**. Tak jak konstruktor jest wywoływany automatycznie podczas tworzenia obiektu, tak destruktor wywoływany jest na rzecz obiektu automatycznie podczas destrukcji tego obiektu przez system (sam destruktor obiektu nie niszczy!). Niszczenie obiektów lokalnych, utworzonych na stosie, zachodzi gdy sterowanie wychodzi z bloku, w którym były zdefiniowane. Jak wiemy, stos jest wtedy zwijany do stanu, w jakim był przy wejściu do danego bloku (funkcji); usunięte zatem zostaną wszystkie zmienne utworzone na stosie w tym bloku, wśród nich również zmienne obiektowe.

Usuwanie obiektów na stosie odbywa się w kolejności odwrotnej do ich kreowania — jako pierwsze zostaną zatem zniszczone te, które utworzone były jako ostatnie. Tak więc, na przykład, zmienne obiektowe zdefiniowane w funkcji **main** są niszczone po wyjściu z tej funkcji — wtedy dopiero wywoływane są dla tych obiektów destruktory, w kolejności odwrotnej do tej, w jakiej były tworzone. Ogólnie, obiekty lokalne (na stosie) są niszczone zaraz po wyjściu sterowania z bloku, w którym były utworzone. Niszczenie obiektów zdefiniowanych globalnie zachodzi po zakończeniu wykonywania funkcji **main** i po zniszczeniu wszystkich obiektów lokalnych z tej funkcji, tuż przed zakończeniem wykonywania całego programu — przykład podamy w następnym podrozdziale.

Inne reguły obowiązują dla obiektów wykreowanych za pomocą operatora **new** na stercie, a więc *nie* jako obiekty lokalne. Nie są one w ogóle niszczone automatycznie; programista musi sam pamiętać, aby wywołać operator **delete** podając jako argument wskaźnik do obiektu na stercie; wartością tego wskaźnika musi być dokładnie ten sam adres, który został zwrócony przez operator **new** podczas tworzenia obiektu (patrz rozdz. 12.3 na str. 217).

Destruktora można czasem w ogóle nie definiować. Jeśli go jednak definiujemy, to jego nazwą jest nazwa klasy poprzedzona znakiem tyldy: **~Nazwa**. Tak jak dla konstruktorów, definiując destruktor nie podaje się żadnego typu zwracanego, nawet **void**. Destruktor *musi* być bezparametrowy. Wynika z tego, że *nie może* być przeciążany. Jest to zrozumiałe: destruktor jest wywoływany automatycznie przez system podczas usuwania obiektu — system nie może wiedzieć, jakie argumenty chcielibyśmy przekazać...

Dla niektórych klas definiowanie destruktorów nie jest potrzebne. Zazwyczaj jednak napisanie destruktora jest wskazane lub nawet konieczne — kiedy takie sytuacje występują, omówimy w następnym rozdziale.

Destruktor zachowuje się jak zwykła metoda, tyle tylko, że jest wywoływany automatycznie podczas usuwania obiektu z pamięci. Samo wywołanie destruktora obiektu nie niszczy: wykonuje on to i tylko to, co zapisane jest w jego kodzie. W szczególności, jeśli chcemy, to możemy jawnie wywołać destruktor na rzecz istniejącego obiektu, na przykład

```
Klasa* obiekt = new Klasa();
// ...
obiekt -> ~Klasa();
```

Takie wywołanie *nie* spowoduje zniszczenia obiektu; treść destruktora zostanie po prostu wykonana, tak jakby było to wywołanie zwykłej metody klasy **Klasa**. Zauważmy, że jawne wywołanie *konstruktora* na rzecz już istniejącego obiektu byłoby błędem.

14.8 Tworzenie obiektów

Definiując zmienne obiektowe klasy należy określić, jaki konstruktor ma być wywołany dla kreowanego obiektu. Robi się to zwykle podając w nawiasach argumenty dla konstruktora. Jeśli ma to być konstruktor domyślny, czyli bezargumentowy, to nawiasy czasem można umieścić, czasem trzeba umieścić, a czasem nie wolno umieszczać w definicji!

Rozpatrzmy na przykładzie sposoby tworzenia obiektów i kolejność wywoływania konstruktorów i destruktorów.

W poniższym programie definiujemy klasę **Klasa**. Ma ona konstruktor z jednym parametrem całkowitym (②). W takim razie żaden konstruktor bezparametrowy (domyślny) nie jest generowany automatycznie. Zatem, jeśli chcemy, żeby taki konstruktor istniał, to musimy sami go zdefiniować (①).

P114: ***tworzob.cpp*** Tworzenie obiektów

```
#include <iostream>
using namespace std;

class Klasa {
    static char ID;
    int          a;
    char        id;
public:
    Klasa() {                                          ①
        id = ID++;
        a  = 0;
        cout << "Ctor()     " << id << a << endl;
    }

    Klasa(int aa) {                                    ②
        id = ID++;
        a  = aa;
        cout << "Ctor(int) " << id << a << endl;
    }

    ~Klasa() {                                         ③
        cout << "Dtor      " << id << a << endl;
    }
};
```

```
25 char Klasa::ID = 'A';                                    ④
26
27 Klasa k1;                              // <- A
28 //Klasa ka();  // NIE!
29 //Klasa ka{};  // OK!
30
31 int main() {
32     cout << "Wchodzimy do funkcji \'main\'" << endl;
33
34     // Klasa kb = Klasa; // NIE!
35     {
36         Klasa k3 = Klasa();          // <- C
37         Klasa k4 = Klasa(4);         // <- D
38     }
39
40     Klasa* pk5 = new Klasa;          // <- E
41     Klasa* pk6 = new Klasa();        // <- F
42     Klasa* pk7 = new Klasa(7);       // <- G
43
44     delete pk6;
45     delete pk7;
46
47     cout << "Wychodzimy z funkcji \'main\'" << endl;
48 }
49
50 Klasa k2(2);                              // <- B
```

Pola klasy są typu **int** (składowa a) i typu **char** (składowa id). Są to pola niestatyczne, a więc odpowiednie składowe będą istnieć w każdym obiekcie klasy. Prócz tego deklarujemy w klasie pole statyczne, też typu **char** o nazwie ID. Zgodnie z tym, co mówiliśmy na temat składowych statycznych klasy, deklaracja pola statycznego nie wystarczy: trzeba tę zmienną statyczną jeszcze zdefiniować poza klasą (czyli spowodować przydzielenie dla niej pamięci). Robimy to w linii ④.

Oba konstruktory zawierają instrukcję '`id = ID++;`', która w składowej id tworzonego obiektu zapamiętuje aktualną wartość zmiennej statycznej ID (istniejącej w jednym egzemplarzu), po czym zwiększa tę zmienną o jeden, zmieniając znak będący wartością tej zmiennej na następny według kolejności kodów ASCII (zmienna była zainicjowana kodem ASCII litery 'A' w linii ④). W ten sposób każdy tworzony obiekt będzie miał unikalną składową znakową id identyfikującą ten obiekt. Oba konstruktory i destruktor (③) drukują komunikat, pozwalający nam śledzić kolejność, w jakiej obiekty są tworzone i usuwane.

Zobaczmy najpierw, jak obiekty można tworzyć.

W linii A, za pomocą instrukcji '`Klasa k1;`' tworzymy zmienną globalną k1. Zauważmy, że taka definicja nie różni się składniowo niczym od definicji zmiennej typu wbudowanego, na przykład '`int k;`' — najpierw podajemy nazwę typu, w tym przypadku jest to **Klasa**, a następnie nazwę deklarowanej/definiowanej zmiennej. Przy

tworzeniu obiektu użyty zostanie konstruktor domyślny (bezargumentowy), bo żaden argument nie został podany. Przy tej formie definiowania zmiennej obiektowej z wykorzystaniem konstruktora domyślnego *nie wolno* umieszczać nawiasów: wykomentowana następna linia (’`Klasa ka();`’) byłaby nielegalna. Jest tak dlatego, że byłaby ona niejednoznaczna, nie dałoby się bowiem odróżnić tej *definicji* zmiennej od *deklaracji* funkcji bezparametrowej o nazwie **ka** i typie zwracanym **Klasa** (standard mówi, że jeśli coś może być zinterpretowane jako deklaracja, to jest deklaracją). Natomiast forma z pustymi nawiasami klamrowymi byłaby legalna (taka postać nie może być zinterpretowana jako deklaracja).

W ostatniej lini w podobny sposób tworzymy obiekt k2 za pomocą instrukcji ’`Klasa k2(2);`’. Teraz użyty ma być konstruktor jednoparametrowy, zatem argument dla tego konstruktora podajemy w nawiasie, tak jakby k2 było nazwą wywoływanej funkcji. Obecność nazwy typu po lewej stronie powoduje jednak, że niejednoznaczności nie ma — składnia tej instrukcji nie odpowiada żadnej możliwej formie wywołania funkcji. Zauważmy, że k2 nie jest tu wcale nazwą konstruktora, który przecież nazywa się **Klasa**, a jest nazwą tworzonego obiektu!

W linii C widzimy inną formę definicji obiektu: ’`Klasa k3 = Klasa();`’. Teraz to nie nazwa tworzonego obiektu, a nazwa typu (klasy) występuje tak, jakby była nazwą wywoływanej funkcji. Zauważmy, że k3 jest tu nazwą *obiektu*, a nie referencji czy wskaźnika do obiektu, jak w Javie (w Javie obiekty w ogóle nie mają nazw — nazwy mają tylko odnośniki do nich). Ponieważ w nawiasie nie podaliśmy żadnych argumentów, użyty będzie konstruktor domyślny. Przy tej formie tworzenia obiektów (poprzez nazwę klasy a nie obiektu) z wykorzystaniem konstruktora domyślnego *trzeba* użyć nawiasów — zatem wykomentowana linia ’`Klasa kb = Klasa;`’ byłaby nielegalna. W linii D podobnie tworzymy obiekt k4, tym razem argument w nawiasie podajemy, więc wywołany zostanie konstruktor jednoparametrowy.

W liniach E, F i Gtworzymy trzy obiekty klasy **Klasa**, tym razem na stercie, za pomocą operatora **new**. Ponieważ operator ten zwraca adres utworzonego obiektu, więc wpisujemy go do zmiennej typu wskaźnikowego **Klasa***. Nazwy pk5, pk6 i pk7 są więc nazwami wskaźników, a nie nazwami utworzonych obiektów; same obiekty nazw nie mają. Zauważmy, że teraz, tworząc obiekt z wykorzystaniem konstruktora domyślnego możemy (linia F), ale nie musimy (linia 39), użyć nawiasów.

To jeszcze nie wszystkie formy definiowania nowych obiektów! Jeśli k1 byłoby nazwą już istniejącego obiektu klasy **Klasa**, to możliwe byłyby też definicje

```
Klasa k8 = k1;
Klasa k9(k1);
```

Takie formy tworzenia obiektów poznamy przy okazji omawiania konstruktorów kopiujących (rozdz. 15.3.1 na stronie 296).

Podsumowując, obiekty klas można definiować na następujące sposoby (zakładamy, że jest zdefiniowany publiczny konstruktor domyślny i konstruktor akceptujący jeden argument typu **int**; w ostatnich dwóch przypadkach musi istnieć konstruktor kopiujący):

```
Klasa a;
```

```
Klasa a(5);

Klasa a = Klasa();
Klasa a = Klasa(5);

Klasa* pa = new Klasa;
Klasa* pa = new Klasa();
Klasa* pa = new Klasa(5);

Klasa b = a;
Klasa b(a);
```

Zwróćmy teraz uwagę na kolejność tworzenia i usuwania obiektów. Pomocny będzie tu wydruk programu, w którym każde wywołanie konstruktora lub destruktora zostawia „ślad" (tradycyjnie skrót *ctor* oznacza konstruktor, a *dtor* — destruktor):

```
Ctor()    A0
Ctor(int) B2
Wchodzimy do funkcji 'main'
Ctor()    C0
Ctor(int) D4
Dtor      D4
Dtor      C0
Ctor()    E0
Ctor()    F0
Ctor(int) G7
Dtor      F0
Dtor      G7
Wychodzimy z funkcji 'main'
Dtor      B2
Dtor      A0
```

Najpierw tworzone są obiekty globalne, w kolejności ich definicji, a zatem obiekt k1 o identyfikatorze A i obiekt k2 o identyfikatorze B i składowej a wynoszącej 2. Definicja tego drugiego obiektu występuje na samym końcu programu. Leży jednak w zasięgu globalnym (poza wszystkimi funkcjami i klasami), a zatem obiekt zostanie utworzony jeszcze *przed* wejściem do funkcji **main**. Na wydruku widzimy, że rzeczywiście konstruktory obiektów o identyfikatorach A i B wywołane zostały przed wejściem do **main**.

Następnie, już wewnątrz funkcji **main**, tworzone są zmienne k3 i k4 o identyfikatorach C i D. Zdefiniowane są one lokalnie wewnątrz bloku ograniczonego nawiasami klamrowymi. Zatem natychmiast po wyjściu sterowania z tego bloku są niszczone w kolejności odwrotnej do tej, w jakiej zostały utworzone w tym bloku: najpierw więc usuwany jest obiekt o identyfikatorze D, a potem obiekt o identyfikatorze C.

Trzy instrukcje następujące za blokiem powodują powstanie obiektów o identyfikatorach E, F i G. Dwa z nich natychmiast usuwamy „ręcznie" za pomocą **delete**. Następnie funkcja **main** kończy swoje działanie. Teraz dopiero usuwane są obiekty

zdefiniowane w zasięgu globalnym B i A. Jak widzimy z wydruku, usuwane są w kolejności odwrotnej niż ta, w jakiej zostały utworzone.

Obiekt wskazywany przez wskaźnik pk5 (o identyfikatorze E) został utworzony na stercie za pomocą operatora **new**. A zatem usunąć go trzeba samemu; ponieważ tego nie zrobiliśmy, nie został w ogóle usunięty. Widzimy z wydruku, że destruktor dla obiektu o identyfikatorze E nie został wywołany.

Dla pewnych klas istnieje też możliwość zainicjowania obiektów za pomocą listy wartości poszczególnych składowych podanej w nawiasach klamrowych, podobnie jak w rozdz. 5.1 na stronie 51 robiliśmy to dla tablic. Z taką możliwością spotkaliśmy się już przy omawianiu C-struktur (patrz rozdz. 13.1 na stronie 235). Metoda ta, zastępująca wywołanie konstruktora, może być jednak stosowana nie tylko dla „czystych" C-struktur. Można jej użyć dla ogólniejszego typu klas, mianowicie dla klas będących **agregatami** danych. Są to klasy spełniające następujące warunki:

- nie ma w nich konstruktorów ani destruktora definiowanych przez użytkownika;
- wszystkie składowe są publiczne;
- jeśli dziedziczą z innej klasy, to klasa bazowa też jest agregatem;
- nie są polimorficzne, czyli nie mają metod wirtualnych (co to znaczy, powiemy w jednym z następnych rozdziałów);
- ich ewentualne składowe obiektowe są obiektami klas, które również są agregatami, a więc spełniają powyższe warunki.

Na przykład w programie

P115: ***aggreg.cpp*** Agregaty

```
#include <iostream>
#include <string>
using namespace std;

class A  {
public:
    int  ia;
    char ca;
    void print() {
        cout << "A: ia = " << ia << " ca = " << ca << endl;
    }
};

struct B {
    A    obA;                              ①
    double x;
    void print() {
        cout << "B: x = " << x << " ";
```

```
19          obA.print();
20      }
21 };
22
23 int main() {
24      B b{ {4,'a'}, 7.5 };                    ②
25      b.print();
26 }
```

obie klasy, **A** i **B**, są takimi agregatami. W szczególności jest agregatem klasa **B**, choć zawiera pole obiektowe obA (①). Pole to jest jednak typu **A**, a klasa **A** jest agregatem. W linii ② widzimy definicję obiektu klasy **B** i inicjowanie go wartościami podanymi na liście w nawiasach klamrowych. Na pierwszej pozycji tej listy, zgodnie z kolejnością pól w klasie **B**, występuje podlista wartości inicjujących obiekt-agregat klasy **A** będący składową tworzonego obiektu klasy **B**. Nawiasy wokół tej podlisty mogą być w pewnych sytuacjach opuszczone, ale lepiej je zawsze jawnie pisać. Wydruk z programu to

```
B: x = 7.5 A: ia = 4 ca = a
```

Jak widzimy z tego przykładu, agregat może nie być czystą C-strukturą; w szczególności może zawierać metody (byle nie polimorficzne).

Poniższy przykład pokazuje, że agregaty można też tworzyć bez jawnej inicjalizacji składowych

P116: *iniagg.cpp* Inicjowanie agregatów

```
1 #include <iostream>
2
3 class A {
4 public:
5     int i;
6     double x;
7 };
8
9 void pr(const A* p) {
10     std::cout << p->i << ", " << p->x << '\n';
11 }
12
13 int main() {
14     A a1{1, 2.5};
15     A a2;
16     A a3{};
17     A* pa4 = new A{3, 4.5};
18     A* pa5 = new A{};
19     pr(&a1); pr(&a2); pr(&a3); pr(pa4); pr(pa5);
20 }
```

Z wydruku tego programu

```
1, 2.5
0, 2.07351e-317
0, 0
3, 4.5
0, 0
```

widzimy, że składowe a2 nie zostały zainicjowane (x ma przypadkową wartość), ale tworząc obiekt a3 użyliśmy składni z (pustym) inicjatorem klamrowym — teraz składowe typów prostych *są* domyślnie zainicjowane (0/**false**/**nullptr**).
Jak widzimy z tego przykładu, podobnie można inicjować obiekty tworzone na stercie (przez **new**).

14.9 Tablice obiektów

Obiekty klasy można grupować w tablice. Zauważmy, że nie jest to możliwe w Javie, w której istnieją wyłącznie tablice typów prostych, w szczególności odnośników (wskaźników).

Nieco skomplikowana jest inicjalizacja takich takich tablic obiektów.

Jeśli klasa jest agregatem (patrz poprzedni podrozdział), to tworzona na stosie tablica obiektów tej klasy sama jest agregatem i może być inicjowana poprzez listę wartości w nawiasach klamrowych w odpowiedniej kolejności. Jeśli podamy za mało wartości, to reszta zostanie zainicjowana zerami (pustymi napisami, zerowymi wskaźnikami). Taką tablicę można też utworzyć nie podając w ogóle inicjatorów — w takim przypadku, tak jak to było dla tablic, wartości składowych będą miały wartość domyślną (która dla typów prostych wynosi „nieokreślona").

W programie poniżej klasa **Klasa** jest agregatem, więc tworzona w lini ① tablica też jest agregatem. Inicjujemy ją przez podanie wartości składowych dla kolejnych obiektów, ale tylko dla pierwszych czterech elementów; piąty będzie zatem wypełniony zerami:

P117: ***agrtab.cpp*** Tablice klas-agregatów

```
1  #include <iostream>
2  using namespace std;
3
4  class Klasa {
5  public:
6      char imie[4];
7      int  wiek;
8  };
9
10 int main() {
11     Klasa ktab[5] = {{"Ala",17},{"Ola",32},     ①
12                      {"Ula",26},{"Iza",29}};
```

```
13      ktab[4].wiek = 22;                                  ②
14
15      for (int i = 0; i < 5; i++)
16          cout << ktab[i].imie << " lat "
17               << ktab[i].wiek << endl;
18  }
```

W linii ② inicjujemy składową wiek dla ostatniego, piątego elementu. Składowa imie tego elementu pozostała wypełniona zerami; odpowiada to pustemu, tym niemniej dobrze zdefiniowanemu, C-napisowi:

```
Ala lat 17
Ola lat 32
Ula lat 26
Iza lat 29
 lat 22
```

Zajmijmy się teraz przypadkiem, gdy klasa/struktura *nie* jest agregatem.

Tablicę obiektów takiej klasy można utorzyć bez jawnej inicjalizacji. Każdy element zostanie utworzony za pomocą konstruktora domyślnego. Zatem konstruktor domyślny musi dla takiej klasy istnieć!

Druga możliwość to na liście inicjalizacyjnej tablicy — w nawiasach klamrowych — wywoływać jawnie konstruktory kreujące obiekty anonimowe, jak w lini ① poniższego programu:

P118: ***classtab.cpp*** Tablice obiektów

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class Klasa {
6      string imie;
7      int    wiek;
8  public:
9      Klasa(const string& imie = "No Name", int wiek = 100) {
10         this->imie = imie;
11         this->wiek = wiek;
12         cout << "konstrukcja " << this->imie << endl;
13     }
14
15     int     getAge() { return wiek; }
16
17     string getImie() { return imie; }
18 };
19
20 int main() {
```

```
21      Klasa ob("Celestyna");
22
23      Klasa ktab[5] = { Klasa("Honoratka", 17),   ①
24                        Klasa("Albertyna"),
25                        Klasa("Hortensja", 26),
26                        ob                        ②
27                      };
28
29      for (int i = 0; i < 5; i++)
30          cout << ktab[i].getImie() << " lat "
31               << ktab[i].getAge()  << endl;
32  }
```

Zauważmy, że tablica ma wymiar 5, ale jawnie skonstruowaliśmy tylko trzy jej elementy. Czwarty element będzie *kopią* wcześniej utworzonego obiektu ob (②), a piąty będzie utworzony przez konstruktor domyślny. Musi on zatem w tej klasie istnieć — i istnieje, dzięki temu, że w jedynym konstruktorze zastosowaliśmy argumenty domyślne. Gdybyśmy jawnie zainicjowali wszystkie pięć elementów, to konstruktora domyślnego mogłoby nie być. Wydruk tego programu:

```
konstrukcja Celestyna
konstrukcja Honoratka
konstrukcja Albertyna
konstrukcja Hortensja
konstrukcja No Name
Honoratka lat 17
Albertyna lat 100
Hortensja lat 26
Celestyna lat 100
No Name lat 100
```

Zauważmy jeszcze konstrukcję z linii ②: element czwarty tablicy ma być tu *kopią* obiektu ob. Aby to było możliwe, musi istnieć publiczny konstruktor kopiujący; w naszej klasie on istnieje, choć go nie widać — więcej o konstruktorach kopiujących powiemy później.

Jeśli tablicę obiektów tworzymy na stercie (poprzez użycie operatora **new**), to nie ma możliwości indywidualnego inicjalizowania elementów tablicy: wszystkie zostaną utworzone za pomocą konstruktora domyślnego, który wobec tego musi istnieć.

14.10 Pola bitowe

Składowymi klas mogą być też **pola bitowe**. Są to składowe klasy o długości wyrażonej w bitach. Ich wartościami są liczby całkowite zapisane (w układzie dwójkowym) w zadeklarowanej ilości bitów. Oczywiście wynika z tego, że zakres tych wartości zależy od zadeklarowanej ilości bitów. Na przykład, deklarując pole bitowe bez znaku

o długości 3, możemy tam zapisywać dane całkowite z zakresu $[0, 7]$, bo istnieje osiem zero-jedynkowych kombinacji trzech bitów:

```
  7    6     5     4     3    2     1     0
111  110   101   100   011  010   001   000
```

Można deklarować pola bitowe tylko typu **int**: ze znakiem (**signed**) lub bez znaku (**unsigned**) — podobnie jak dla normalnych typów całkowitych. Liczbę bitów przeznaczonych dla danego pola deklarujemy po dwukropku; na przykład po

```
struct A {
    unsigned kolor : 4;
    // ...
};
```

w klasie **A** istnieje składowa kolor będąca polem bitowym bez znaku o długości czterech bitów. Można zatem tej składowej przypisywać wartości z zakresu $[0, 15]$. Dlaczego nie użyliśmy zwykłego typu **int** bez określania ilości bitów? Otóż chodzi o to, że jeśli w klasie występuje kilka pól bitowych, kompilator zadba o to, aby je wszystkie upakować jak najgęściej. Jeśli zatem w klasie mamy pola bitowe o długościach 4, 7 i 3, to, ponieważ jedna liczba typu **int** zajmuje 32 bity, wszystkie zmieszczą sie w jednym **int**'cie! Uzyskujemy zatem oszczędność pamięci i mamy możliwość kodowania kilku różnych informacji w pojedynczej fizycznie danej.

W poniższym przykładzie definiujemy klasę opisującą rodzaj czcionki: w polach bitowych przechowujemy informację o kroju, wadze i kolorze czcionki:

P119: *polbit.cpp* Pola bitowe

```
#include <iostream>
using namespace std;

class Czcionka {
    unsigned  kroj : 3;
    unsigned  waga : 1;
    unsigned kolor : 2;
public:
    enum  Kroj { HELVETICA, TIMES, ARIAL,       ①
                 COURIER, BOOKMAN, SYMBOL};
    enum  Waga { NORMAL, BOLD };
    enum Kolor { BLACK, RED, GREEN, BLUE };

    Czcionka(Kroj kroj, Waga waga, Kolor kolor) {
        this->kroj  = kroj;
        this->waga  = waga;
        this->kolor = kolor;
    }

```

```
20      void opis() {
21          cout << "Kroj nr " << kroj << "; Waga nr "
22               << waga << "; Kolor nr " << kolor << endl;
23      }
24  };
25
26  int main() {
27      Czcionka tytul(Czcionka::ARIAL,  Czcionka::BOLD,
28                                       Czcionka::RED);
29      Czcionka tekst(Czcionka::TIMES,  Czcionka::NORMAL,
30                                       Czcionka::BLACK);
31      Czcionka greka(Czcionka::SYMBOL, Czcionka::BOLD,
32                                       Czcionka::BLUE);
33      tytul.opis();
34      tekst.opis();
35      greka.opis();
36      cout << "Rozmiar obiektu: " << sizeof(Czcionka) << endl;
37  }
```

Pole przewidziane dla wyboru kroju ma długość trzech bitów, a więc może przyjmować wartości od 0 do 7. Tym wartościom nadaliśmy dla wygody nazwy za pomocą wyliczenia **Kroj** (①). W ten sposób użytkownik klasy nie musi pamiętać, jakie liczby zostały przypisane poszczególnym krojom — może się do nich odnosić poprzez ich czytelne nazwy. Podobnie postąpiliśmy dla wagi (jeden bit, a więc dwie możliwości odpowiadające liczbom 0 i 1) i koloru (dwa bity, a więc cztery możliwości: 0, 1, 2 i 3). Wydruk z tego programu

```
Kroj nr 2; Waga nr 1; Kolor nr 1
Kroj nr 1; Waga nr 0; Kolor nr 0
Kroj nr 5; Waga nr 1; Kolor nr 3
Rozmiar obiektu: 4
```

przekonuje nas, że rozmiar obiektu wynosi cztery bajty, a więc kompilator upakował wszystkie trzy pola w jednym **int**'cie. Ponieważ kilka pól bitowych może być upakowanych przez kompilator w obszarze jednej liczby całkowitej typu**int** w sposób trudny do przewidzenia, nie ma sensu stosowanie do składowych bitowych operatora wyłuskania adresu ('&'), a zatem i wskaźników.

Pola takie nie mogą być składowymi statycznymi.

Biblioteka standardowa C++ dostarcza klasę (a właściwie szablon klasy) **bitset**, która zapewnia funkcjonalność pól bitowych, a zawiera wygodny interfejs dla użytkownika i efektywną implementację. Tam, gdzie to możliwe, należy zatem korzystać z tej klasy, a nie z pól bitowych.

— 15 —

Klasy (II)

W poprzednim rozdziale zapoznaliśmy się z podstawowymi pojęciami związanymi z definiowaniem klas i tworzeniem obiektów. Teraz zajmiemy się szeregiem dalszych szczegółów, które są niezbędne, aby operować klasami w sposób efektywny i bezpieczny. Omówimy zatem metody stałe, konstruktory kopiujące, listy inicjalizacyjne, klasy z polami wskaźnikowymi, funkcje zaprzyjaźnione.

PODROZDZIAŁY:

15.1 Metody stałe

Metody klasy mogą być zdeklarowane jako metody stałe. Oznacza to „obietnicę", że dana metoda nie zmienia stanu obiektu, na rzecz którego została wywołana, czyli nie zmienia żadnej jego składowej. Kompilator sprawdzi oczywiście, czy rzeczywiście obiekt nie jest zmieniany. Deklarujemy metodę jako stałą umieszczając słowo kluczowe **const** tuż za nawiasem zamykającym listę parametrów, a *przed*

- średnikiem w deklaracji;
- nawiasem klamrowym otwierającym ciało funkcji w definicji.

Oczywiście, deklarujemy tę stałość tylko raz, jeśli metodę definiujemy bezpośrednio wewnątrz klasy. Jeśli natomiast, jak to zwykle czynimy, w klasie tylko metodę deklarujemy, a definicję podajemy poza klasą, to **const** trzeba podać w obu miejscach. Pamiętać bowiem trzeba, że dwie metody — nawet o tej samej sygnaturze — z których jedna jest stała, a druga nie są *różnych* typów! Zatem

deklaracja stałości musi wystąpić zarówno w definicji, jak i w deklaracji metody.

Zauważmy też, że metoda stała nie może zmienić składowych obiektu, na rzecz którego została wywołana, ale może zmienić składowe innych obiektów tej samej klasy, do których ma dostęp.

Rozpatrzmy przykład:

P120: *constmet.cpp* Metody stałe

```
#include <iostream>
using namespace std;

class Point {
    double x, y;
public:
    Point(double x, double y) {
        this->x = x;
        this->y = y;
    }
    Point translate(double dx, double dy) const;
    void  translate(double dx, double dy);
};

Point Point::translate(double dx, double dy) const {
    cout << "const translate\n";
    return Point(x+dx,y+dy);
}

void Point::translate(double dx, double dy) {
    cout << "nonconst translate\n";
    x += dx;
    y += dy;
}

int main() {
    const Point p1(1,1);
    Point p2(2,2);

    p1.translate(3,3);
    p2.translate(4,4);
}
```

W klasie **Point** zadeklarowane są dwie metody **translate** (o tej samej nazwie i tej samej liczbie i typach parametrów!), ale pierwsza jest stała, a druga nie, więc ich typ różni się wystarczająco, aby takie przeciążenie było możliwe. W programie definiujemy dwa punkty: p1, który jest ustalony (**const**) i p2, który ustalony nie jest. Dla obu tych punktów wywołujemy metodę **translate** ignorując ewentualną wartość zwracaną. Jak widzimy z wydruku

```
const translate
nonconst translate
```

w pierwszym przypadku, kiedy metoda jest wywołana na rzecz obiektu stałego, kompilator wybierze metodę, która „obiecuje", że obiektu nie zmieni. Gdyby jej nie było, powstałby błąd kompilacji, bo

> na rzecz obiektu stałego można wywoływać wyłącznie metody zadeklarowane jako stałe.

W drugim natomiast przypadku, kiedy obiekt ustalony nie jest, kompilator wybiera nieustaloną wersję metody jako lepiej pasującą. Zauważmy, że gdyby tej nieustalonej wersji nie było, to wywołana zostałaby w tym przypadku wersja ustalona. Tak więc pisząc metodę, która nie zmienia stanu obiektu, lepiej jest napisać ją jako ustaloną, bo wtedy można ją będzie wywoływac zarówno na rzecz obiektów zmiennych jak i stałych. Natomiast metodę nieustaloną można wywoływać *tylko* na rzecz obiektów zmiennych, nawet jeśli w rzeczywistości wcale obiektu nie zmienia.

Z oczywistych względów konstruktor nie może być deklarowany z modyfikatorem **const** — jego głównym zadaniem jest bowiem właśnie modyfikowanie stanu tworzonego obiektu.

15.1.1 Pola *mutable*

Z metodami stałymi występuje pewien problem: czasem chcemy, aby pewna metoda była stała, bo wynika to z logiki klasy do której należy, ale jednocześnie powinna jednak mieć możliwość dokonania pewnych zmian stanu obiektu (czyli jego składowych), dzięki czemu może na przykład być zaimplementowana w sposób bardziej efektywny.

W takich sytuacjach można niektóre pola klasy zadeklarować ze specyfikatorem **mutable**. Oznacza to właśnie, że odpowiednie składowe obiektów tej klasy *mogą* być modyfikowane nawet przez metody stałe, zadeklarowane jako **const**.

Aby zobaczyć, że taka pozorna niekonsekwencja może jednak mieć sens, rozpatrzmy klasę opisującą, w bardzo uproszczony sposób, osobę, którą może na przykład być klient banku.

P121: ***mutab.cpp*** Pola *mutable*

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 struct FullInfo {
6     string address;
7
8     FullInfo(string name) {
9         cout << "Szukam adresu w bazie danych" << endl;
```

```
10          address = "Adres pana " + name;
11      }
12 };
13
14 class Klient {
15      string  name;
16      mutable FullInfo *fullInfo;
17 public:
18      Klient(string n) {
19          name     = n;
20          fullInfo = nullptr;
21      }
22
23      string getInfo() const {
24          return name;
25      }
26
27      string getFullInfo() const {
28          if (fullInfo == nullptr)                    ①
29              fullInfo = new FullInfo(name);
30          return name + ", " + fullInfo->address;
31      }
32
33      ~Klient() {
34          delete fullInfo;
35          cout << "usuwam " + name << endl;
36      }
37 };
38
39 int main() {
40      Klient klient("Nowak");
41      cout << klient.getInfo()     << endl;          ②
42      cout << klient.getFullInfo() << endl;          ③
43      cout << "Koniec \'main\'\n";
44 }
```

Obiekt klasy **Klient** zawiera pewną informację o kliencie, w naszym uproszczonym przypadku jest to po prostu nazwisko (składowa name). Wyobraźmy sobie, że w większości zastosowań ta informacja o kliencie wystarcza. Czasem jednak potrzebna jest informacja bardziej szczegółowa, opisana klasą **FullInfo** (w naszym przypadku jest tam adres klienta). Ta informacja jest trudno dostępna, bo na przykład wymaga połączenia z odległą bazą danych albo skomplikowanej procedury autoryzacyjnej. Użytkownik klasy **Klient** nie musi jednak o tym wiedzieć — dla niego ważne jest, że mając obiekt tej klasy powinien być w stanie w każdej chwili zapytać zarówno o nazwisko jak i o adres. Jeśli interesuje go tylko informacja skrócona, używa metody **getInfo** (②) — ta metoda oczywiście jest **const**, bo tylko zwraca żądaną informację, nie modyfikując

obiektu.

Co jednak będzie, jeśli użytkownik potrzebuje informacji pełnej? Wywołuje metodę **getFullInfo** (③), która też powinna być stała, bo tylko zwraca informację. Ale implementacja jest inna — zastosowaliśmy tu strategię leniwego wyliczania (*lazy evaluation*): pełną informację pobieramy dopiero, gdy użytkownik rzeczywiście jej zażąda, gdyż dla większości obiektów nie będzie potrzebna, a zatem pobieranie jej dla wszystkich tworzonych obiektów klasy **Klient** nie byłoby rozsądne. Widzimy, że metoda **getFullInfo** sprawdza, czy jest to pierwsze wywołanie (składowa fullInfo jest wtedy **nullptr**, linia ①) i jeśli tak, tworzy dopiero teraz obiekt klasy **FullInfo**, co wymaga połączenia z bazą danych. Oczywiście, gdy ta sama metoda zostanie wywołana drugi raz dla tego samego obiektu, pełna informacja będzie już dostępna i ponowne łączenie z bazą danych nie będzie potrzebne. A zatem metoda **getFullInfo** z punktu widzenia „kontraktu" z użytkownikiem powinna być stała; z drugiej strony, przy pierwszym wywołaniu musi zmienić stan obiektu zmieniając składową fullInfo. Dlatego pole fullInfo *musiało* być zadeklarowane jako **mutable**. Gdybyśmy ten specyfikator opuścili, program w ogóle by się nie skompilował:

```
cpp> g++ -pedantic-errors -Wextra mutab.cpp
mutab.cpp: In member function
    `std::string Klient::getFullInfo() const':
mutab.cpp:29: error: assignment of data-member
    `Klient::fullInfo' in read-only structure
```

Wydruk tego programu (w postaci poprawnej, ze specyfikatorem **mutable**)

```
Nowak
Szukam adresu w bazie danych
Nowak, Adres pana Nowak
Nowak, Adres pana Nowak
Koniec 'main'
usuwam Nowak
```

dodatkowo ilustruje fakt, że obiekty zdefiniowane w funkcji są usuwane już po jej opuszczeniu i wywoływany jest dla nich destruktor.

15.2 Metody ulotne

Metody klasy mogą też być zdeklarowane jako metody ulotne (ang. *volatile*). Tak jak metody stałe „obiecują", że nie zmienią stanu obiektu, na rzecz którego zostały wywołane, tak funkcje ulotne obiecują, że przy dostępie do składowych nie będą stosowane żadne optymalizacje, polegające na przykład na pamiętaniu wartości składowej w pamięci podręcznej (ang. *cache*) lub rejestrze procesora przed ponownym użyciem. Każde odwołanie się do składowej obiektu połączone będzie z odczytaniem na nowo jej aktualnej wartości, a modyfikacja wartości składowej wykonywana będzie natychmiast, bez ewentualnego buforowania. Z takich metod korzysta się rzadko, najczęściej wtedy, gdy stan obiektu może być zmieniony przez czynniki zewnętrzne, nieznane programowi. Taka *asynchroniczna* zmiana danych może mieć miejsce, na przykład, na

skutek dołączenia do komputera jakichś czujników czy mierników pracujących w trybie *on line* lub, jeszcze częściej, na skutek uruchomienia funkcji obsługi sygnałów czy też na skutek wielowątkowości programu.

Składnia definiowania metod ulotnych jest dokładnie taka sama jak składnia definiowania i deklarowania metod stałych; należy tylko słowo kluczowe **const** zastąpić słowem kluczowym **volatile**. Metoda może być jednocześnie stała i ulotna.

Podobnie jak w przypadku modyfikatora **const**, modyfikatora **volatile** nie można użyć w odniesieniu do konstruktora.

15.3 Konstruktory – dalsze szczegóły

O konstruktorach mówiliśmy już w rozdz. 14.6. Nie wyczerpaliśmy jednak tego tematu. Bardzo ważną rolę w C++ pełnią konstruktory kopiujące. Omówimy je w następnym podrozdziale. Innym ważnym mechanizmem jest mechanizm inicjowania tworzonych obiektów za pomocą list inicjalizacyjnych. Często jest on tylko wygodnym skrótem, czasem jednak jest konieczny do prawidłowego skonstruowania obiektu. Definiowanie konstruktorów kopiujących jest zazwyczaj konieczne w klasach zawierających pola wskaźnikowe; wtedy też niezbędne jest zdefiniowanie samemu *destruktora* i przedefiniowanie operatora przypisania.

15.3.1 Konstruktory kopiujące

Zadaniem **konstruktora kopiującego** jest utworzenie obiektu identycznego jak inny, istniejący wcześniej, obiekt tej samej klasy, pełniący wobec tego rolę wzorca. Obiekt ma być identyczny, ale z oryginałem całkowicie rozłączny; późniejsza modyfikacja jednego z nich nie powinna mieć wpływu na drugi.

Dla klasy **A** jest to konstruktor o sygnaturze '`A(const A&)`'. Modyfikator **const** nie jest tu obowiązkowy, co za chwilę wyjaśnimy. Oczywiście rolę konstruktora kopiującego może też pełnić konstruktor, w którym pierwszy parametr jest typu **const A&** (lub **A&**), a pozostałe mają zdefiniowane wartości domyślne. W każdym razie pierwszym (i najczęściej jedynym) argumentem wywołania takiego konstruktora będzie zawsze istniejący wcześniej obiekt klasy **A**, który zostanie do konstruktora kopiującego przesłany przez referencję.

Mogłoby się wydawać, że tego rodzaju konstruktor często w ogóle nie będzie potrzebny. Tak jednak nie jest: jest on potrzebny bardzo często, choć nie zawsze sami musimy go definiować. Zauważmy bowiem, że kopiowanie obiektów zachodzi zawsze, gdy obiekt pełni rolę argumentu wywołania funkcji, jeśli tylko przekazywany jest do funkcji przez wartość, a nie przez wskaźnik lub referencję; na stosie musi zostać położona *kopia* obiektu. Podobnie rzecz się ma przy zwracaniu przez wartość obiektu jako rezultatu funkcji: tu również jest wykonywana kopia obiektu. Za każdym razem w takim przypadku używany jest konstruktor kopiujący. Widzimy zatem, że trudno byłoby napisać program, w którym konstruktory kopiujące nie byłyby używane. Rozpatrzmy przykład programu, który nic nie robi prócz pisania informacji o wywołaniach konstruktorów.

P122: ***kopiow.cpp*** Konstruktor kopiujący

```
#include <iostream>
using namespace std;

class A {
    double x;
public:
    A(double x = 1) {
        this->x = x;
        cout << "W konstruktorze domyslnym"  << endl;
    }

    A(const A& a) {
        x = a.x;
        cout << "W konstruktorze kopiujacym" << endl;
    }
};

A fun(A a) {
    cout << "W funkcji fun" << endl;
    return a;
}

int main() {
    cout << "**1**" << endl;
    A a;

    cout << "**2**" << endl;
    A b = a;                    ①

    cout << "**3**" << endl;
    A c(b);                     ②

    cout << "**4**" << endl;    ③
    c = fun(a);                 ④
}
```

Program ten demonstruje przy okazji dwa sposoby tworzenia obiektów, których jeszcze nie omawialiśmy w rozdz. 14.8. W linii ① instrukcja '`A b = a;`' tworzy na stosie (a więc jako zmienną lokalną) obiekt `b` będący *kopią* wcześniej istniejącego obiektu `a`. A zatem wywołany będzie konstruktor kopiujący! Zauważmy, że taki zapis nie ma nic wspólnego z przypisaniem: przypisywać można tylko do istniejących obiektów. Jest to prawie to samo co jawnie robimy w linii ② za pomocą nieco różnej składni ('`A c(b);`'). Tu jawnie wywołujemy konstruktor klasy **A** posyłając istniejący już teraz obiekt `b` jako wzór. Jest drobna różnica między tymi przypadkami — za pierwszym

razem wywołanie konstruktora kopiującego jest traktowane jako *niejawne* (ang. *implicit*) a w drugim jako *jawne* (ang. *explicit*). Jak się przekonamy, konstruktory mogą być zadeklarowane jako tylko jawne i w takim przypadku pierwsza forma nie zadziałałaby.

Przyglądając się wydrukowi z tego programu

```
**1**
W konstruktorze domyslnym
**2**
W konstruktorze kopiujacym
**3**
W konstruktorze kopiujacym
**4**
W konstruktorze kopiujacym
W funkcji fun
W konstruktorze kopiujacym
```

zauważamy, że gdy wydrukowany już został napis '`**4**`', a więc po wykonaniu instrukcji z linii ③ programu, konstruktor kopiujący został jeszcze wywołany dwukrotnie, mimo że, jak się wydaje, żadnych nowych obiektów już nie kreujemy. Dlaczego zatem zadziałał? Konstruktor ten został wywołany dwa razy podczas obsługi wywołania funkcji **fun** w linii ④:

- aby skopiować argument, czyli obiekt `a`, w celu położenia tej kopii na stosie, gdzie będzie utożsamiona z lokalną dla funkcji zmienną `x`,
- po wykonaniu treści funkcji do skopiowania zmiennej lokalnej `x` przed zwróceniem jej wartości jako rezultatu funkcji.

Co by było, gdybyśmy w naszej klasie **A** konstruktora kopiującego nie napisali? Akurat w tym przypadku nic złego by się *nie* stało. Jeśli programista takiego konstruktora nie napisał, to zostanie on wygenerowany przez kompilator niezależnie od tego, czy w ogóle jakieś inne konstruktory dostarczyliśmy, czy nie (a zatem *inaczej* niż dla konstruktorów domyślnych). Wygenerowany automatycznie konstruktor kopiujący jest zawsze konstruktorem o dostępności typu **public**, tak jak automatycznie generowany konstruktor domyślny. Kopiuje on po prostu składowe obiektu-wzorca do odpowiednich składowych kreowanego obiektu. W przypadku tak prostej klasy jak klasa **A** z naszego przykładu, będzie to akurat to, o co nam chodzi: obiekt tej klasy zawiera tylko jedną składową (typu **double**). A zatem konstruktor kopiujący wygenerowany automatycznie wykonałby dokładnie to samo, co ten napisany przez nas (z wyjątkiem drukowania informacji o swoim działaniu).

Z tego, co powiedzieliśmy wynika, że w naszej klasie **A** konstruktora kopiującego w ogóle nie musieliśmy pisać; służył w niej jedynie celom dydaktycznym. Jak się jednak przekonamy, nie zawsze tak jest.

Wyjaśnijmy jeszcze, dlaczego argument musi być referencją, a nie obiektem przekazywanym przez wartość. Gdyby był obiektem, to ponieważ argumenty przekazywane przez wartość są podczas wywołania kopiowane i kładzione na stosie, musiałaby najpierw zostać wykonana kopia tego obiektu. Ale do tego potrzebne byłoby

. . . wywołanie konstruktora kopiującego i przesłanie do niego przez wartość argumentu, a do tego znowu trzeba by wykonać kopię, a więc wywołać konstruktor kopiujący, i tak dalej, *ad infinitum*. Konstruktor kopiujący byłby zatem wywoływany rekursywnie w nieskończoność.

Z drugiej strony, przekazując do konstruktora obiekt-wzorzec przez referencję, dajemy mu możliwość zmiany tego obiektu-wzorca, dostaje on bowiem wtedy oryginał obiektu, a nie jego kopię. Najczęściej taka zmiana byłaby niepożądana. Dlatego właśnie, aby się przed możliwością takiej zmiany zabezpieczyć, parametr konstruktora kopiującego deklarujemy z modyfikatorem **const**. Sam kompilator zadba wtedy o to, abyśmy nawet nieświadomie czy przypadkowo nie zmodyfikowali obiektu-wzorca, a prócz tego będzie możliwe przekazywanie jako argumentu referencji do obiektu stałego (**const**).

Wewnątrz konstruktora kopiującego, jak i każdego innego, można odwoływać się do składowych tworzonego obiektu (które już istnieją). Można również wywoływać inne metody.

Jak wspomnieliśmy, automatycznie wygenerowany konstruktor kopiujący kopiuje obiekt-wzorzec na nowo tworzony obiekt „pole po polu". Znaczy to, że kopiowanie takie jest płytkie: jeśli istnieją pola wskaźnikowe — np. dynamicznie tworzone tablice — elementem obiektu jest wtedy tylko wskaźnik, a nie cała tablica, a więc kopiowane są te wskaźniki, a nie obiekty przez nie wskazywane. W takich przypadkach programista musi dostarczyć właściwy konstruktor kopiujący zapewniający kopiowanie głębokie.

Zobaczmy o co tu chodzi na przykładzie. Rozważmy klasę **Osoba**, której składowymi ma być np. wiek i imię danej osoby:

```
class Osoba {
    int   wiek;
    char* imie;
    // ...
};
```

Pole wiek jest typu całkowitego i nie sprawia kłopotu. Gorzej jest z polem imie, które ma być napisem, czyli tablicą znaków. Zadeklarowaliśmy je jako typu **char***. Dlaczego? Zastanówmy się, jaki wymiar ma mieć ta tablica? Tego nie wiemy: jedne imiona są krótkie, inne długie. Moglibyśmy ustalić maksymalną długość raz na zawsze pisząc

```
class Osoba {
    int  wiek;
    char imie[20];
    // ...
};
```

W każdym obiekcie klasy zawarta wtedy będzie tablica dwudziestu znaków. Zazwyczaj imiona są znacznie krótsze, więc większość pamięci przydzielonej w ten sposób na imiona będzie się marnować. Z drugiej strony, nie możemy za bardzo zmniejszyć tego wymiaru, bo jednak od czasu do czasu zdarzyć się mogą imiona rzeczywiście długie. Zatem decydujemy się na inne rozwiązanie: w obiekcie będziemy przechowywać

tylko wskaźnik do tablicy znaków, a zatem wielkość typu **char***, a samą tablicę będziemy alokować dynamicznie w konstruktorze dokładnie takiej wielkości, jaka będzie potrzebna: małą tablicę dla imion krótkich, większą dla długich. Do konstruktora będziemy zatem przesyłać tablicę znaków (czyli wskaźnik do istniejącej tablicy znaków zakończonej znakiem pustym '\0').

Dlaczego w ogóle alokować tablicę w konstruktorze? Dlaczego nie napisać:

```
class Osoba {
    int   wiek;
    char* imie;
public:
    Osoba(char* im) {
        imie = im;
    }
    // ...
};
```

Tak zrobić zazwyczaj nie można. Błędu formalnie tu nie ma, ale zauważmy, że w ten sposób w obiekcie zapamiętaliśmy adres pewnego napisu utworzonego gdzie indziej. Nie wiemy, czy jest on modyfikowalny czy nie, nie wiemy co się z tym napisem będzie dalej działo: może zaraz zostanie zmieniony albo usunięty, a obiekt zostanie z adresem czegoś, czego już w ogóle nie ma!

Zatem lepiej zrobimy, jeśli w konstruktorze zaalokujemy odpowiednią ilość miejsca w pamięci wolnej (na stercie) i tam ten napis przekopiujemy:

P123: *osoba1.cpp* Alokowanie pamięci w konstruktorze

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  class Osoba1 { // niezupełnie dobra klasa...
6  public:
7      int   wiek;
8      char* imie;
9      Osoba1(int w, const char* im) {
10         wiek = w;
11         imie = new char[strlen(im)+1]; ①
12         strcpy(imie,im);               ②
13     }
14 };
15
16 int main() {
17     char imie[] = "Basia";
18
19     Osoba1 basia(29, imie);
20
```

```
21      imie[0] = 'K';                          ③
22
23      cout << "Oryginal:  " << imie       << endl;
24      cout << "Z obiektu: " << basia.imie << endl;
25  }
```

Użyliśmy tu funkcji z nagłówka ***cstring***

- **strlen** — zwraca długość napisu wskazywanego przez argument typu **char***, nie licząc kończącego znaku '\0';
- **strcpy** — kopiuje napis wskazywany przez drugi argument do napisu wskazywanego przez pierwszy argument, łącznie z kończącym znakiem '\0'. Zwraca wskaźnik będący pierwszym argumentem.

Zauważmy, że w konstruktorze przydzieliliśmy sobie (linia ①) o jeden bajt więcej niż wynosi długość napisu przesłanego do tego konstruktora; jest to konieczne, gdyż przekopiować trzeba cały napis, łącznie ze znakiem '\0' (co robimy w linii ②). Program drukuje

```
Oryginal:  Kasia
Z obiektu: Basia
```

co przekonuje nas, że obiekt zawiera adres swojej „prywatnej" kopii napisu przesłanego do konstruktora. Oryginał tego napisu zmieniliśmy w linii ③ z `Basia` na `Kasia`, ale składowa imie obiektu wskazuje na tę „prywatną" kopię, która zmianie nie uległa.

Mamy tu zatem do czynienia z sytuacją, gdy obiekt fizycznie nie zawiera pełnej informacji o reprezentowanej przez siebie osobie (w tym przypadku nie zawiera imienia tej osoby), a tylko adres obszaru pamięci na stercie, gdzie ta informacja została zapisana. W ten sposób sam obiekt jest niewielki: zawiera tylko liczbę całkowitą (składowa wiek) i wskaźnik typu **char*** (składowa imie). Związana z nim informacja (tablica znaków zawierająca imię) jest zaalokowana poza obiektem, na stercie i zajmuje tylko tyle miejsca, ile jest to niezbędne.

Co teraz będzie, jeśli zechcemy użyć konstruktora kopiującego wygenerowanego przez system?

P124: ***osoba2.cpp*** Pola wskaźnikowe i automatyczny konstruktor kopiujący

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  class Osoba2 { // niezupełnie dobra klasa...
6  public:
7      int   wiek;
8      char* imie;
9      Osoba2(int w, const char* im) {
```

```
10          wiek = w;
11          imie = new char[strlen(im)+1];
12          strcpy(imie,im);
13      }
14 };
15
16 int main() {
17      char imie[] = "Basia";
18
19      Osoba2 basia(29, imie);
20      Osoba2 kasia(basia);       // użycie konstr. kopiującego
21
22      cout << "Po utworzeniu:   basia " << basia.imie << endl;
23      cout << "                 kasia " << kasia.imie << endl;
24
25      kasia.imie[0] = 'K';
26
27      cout << "Po zmianie Kasi: basia " << basia.imie << endl;
28      cout << "                 kasia " << kasia.imie << endl;
29 }
```

Na początku tworzymy obiekt basia o imieniu 'Basia'. W linii następnej tworzymy, poprzez konstruktor kopiujący, obiekt kasia. W obu obiektach imię jest 'Basia', o czym świadczą dwie pierwsze linie wydruku:

```
Po utworzeniu:   basia Basia
                 kasia Basia
Po zmianie Kasi: basia Kasia
                 kasia Kasia
```

Następnie (linia 25) zmieniamy imię w obiekcie kasia na Kasia. Jak widać z wydruku, spowodowało to również zmianę imienia Basi! Oczywiście, wiemy dlaczego tak się stało. Konstruktor kopiujący, tworząc obiekt kasia, przekopiował składową wskaźnikową imie, zawierającą adres, a nie sam napis, tak więc po utworzeniu obiektu kasia w obu obiektach składowa ta ma tę samą wartość: adres napisu zaalokowanego przez zwykły konstruktor podczas tworzenia obiektu basia. Obiekty są dwa, ale napis z imieniem jest tylko jeden, wskazywany przez składowe wskaźnikowe imie obu obiektów.

Tak oczywiście być nie powinno i to właśnie jest sytuacja, gdy automatyczny konstruktor kopiujący nie wykonuje tego, o co nam chodzi. Zatem sami musimy zadbać o to, by podczas tworzenia obiektu za pomocą konstruktora kopiującego każdy tworzony obiekt zaopatrzyć we własną kopię imienia:

P125: *osoba3.cpp* Własny konstruktor kopiujący

```
1 #include <iostream>
2 #include <cstring>
3 using namespace std;
```

```
4
5  class Osoba {    // trochę lepsza klasa
6  public:
7      int   wiek;
8      char* imie;
9      Osoba(int w, const char* im) { // zwykły konstruktor
10         wiek = w;
11         imie = new char[strlen(im)+1];
12         strcpy(imie,im);
13     }
14     Osoba(const Osoba& os) {       // konstruktor kopiujący
15         wiek = os.wiek;
16         imie = new char[strlen(os.imie)+1];
17         strcpy(imie,os.imie);
18     }
19     ~Osoba() {                     // destruktor
20         delete [] imie;
21     }
22 };
23
24 int main() {
25     char imie[] = "Basia";
26
27     Osoba basia(29, imie);
28     Osoba kasia(basia);
29
30     cout << "Po utworzeniu:   basia " << basia.imie << endl;
31     cout << "                 kasia " << kasia.imie << endl;
32
33     kasia.imie[0] = 'K';
34
35     cout << "Po zmianie Kasi: basia " << basia.imie << endl;
36     cout << "                 kasia " << kasia.imie << endl;
37 }
```

W tym programie definiujemy „normalny” konstruktor, ale również konstruktor kopiujący, który kopiuje pole wiek, mierzy długość imienia w obiekcie-wzorcu i przydziela odpowiednią ilość miejsca na stercie, po czym kopiuje napis z imieniem wskazywany przez składową imie obiektu-wzorca do zaalokowanej pamięci. Wydruk

```
Po utworzeniu:   basia Basia
                 kasia Basia
Po zmianie Kasi: basia Basia
                 kasia Kasia
```

wskazuje, że tym razem napisy wskazywane przez składowe imie obu obiektów są inne: zmiana jednego z nich nie wpłynęła na drugi.

Dodaliśmy też do naszej klasy destruktor. W tej klasie jest on potrzebny. Tworząc obiekt, niezależnie od tego, który konstruktor został użyty, zaalokowaliśmy pewien obszar pamięci na stercie za pomocą operatora **new**. Nawet gdy obiekt jest lokalny (na stosie) i zostanie usunięty po wyjściu sterowania z bloku, w którym był zdefiniowany, pamięć, która do niego „należała" na stercie *nie* zostanie zwolniona. Ponieważ jednak wiemy, że gdy obiekt będzie usuwany, wywołany będzie automatycznie na jego rzecz destruktor, właśnie w nim umieszczamy kod zwalniający tę pamięć.

Tak skonstruowana klasa wciąż nie jest jeszcze prawidłowa. Jak się przekonamy, brakuje tu przeciążenia operatora przypisania (patrz rozdz. 19 na stronie 385).

15.3.2 Listy inicjalizacyjne

Jak podkreślaliśmy, w momencie, gdy zaczyna działać konstruktor, obiekt już istnieje. W szczególności istnieją wszystkie jego składowe — konstruktor może tylko zmienić ich wartość poprzez przypisania.

Pojawia się tu problem. Jak na przykład zainicjować składową odpowiadającą polu stałemu (z modyfikatorem **const**) albo referencyjnemu? W obu przypadkach składnia wymaga zainicjowania już w trakcie tworzenia, zatem nie można tego zrobić w konstruktorze — wtedy składowa już powinna istnieć! Co zrobić, gdy składową jest obiekt klasy, która nie ma konstruktora domyślnego?

We wszystkich tych przypadkach kompilacja nie uda się, jeśli sprawę inicjowania takich składowych pozostawimy dla konstruktora: wtedy jest już za późno.

Jednym ze sposobów zdefiniowania stałej jako składowej klasy jest użycie wyliczenia, które może być wtedy anonimowe:

```
class A {
    enum { dim = 10 };
    int tab[dim];
public:
    ...
    void fun() {
        ...
        for (int i = 0; i < dim; ++i) { ... }
        ...
    }
};
```

Stała taka jednak musi mieć nadaną wartość bezpośrednio w kodzie programu i będzie taka sama dla wszystkich obiektów. W innych przypadkach inicjalizacja tego rodzaju składowych może odbywać się tylko na podstawie **listy inicjalizacyjnej** konstruktora (ang. *initialization list*).

Lista inicjalizacyjna to lista oddzielonych przecinkami identyfikatorów pól (składowych) z podanymi w nawiasach okrągłych argumentami dla konstruktorów obiektów będących składowymi tworzonego obiektu. Zwykle są to jednocześnie argumenty formalne definiowanego konstruktora, choć nie musi tak być. Jeśli argumentem przesyłanym do konstruktora obiektu składowego na liście inicjalizacyjnej jest obiekt tego

samego typu, co ta składowa, to traktowane to będzie jako wywołanie konstruktora kopiującego. Taka składnia działa również dla składowych, które są typu wbudowanego — jak wielokrotnie podkreślaliśmy, twórcy języka starali się, aby reguły składniowe dla typów obiektowych i wbudowanych były do siebie tak podobne, jak to tylko możliwe.

Listę inicjalizacyjną, poprzedzoną dwukropkiem, umieszcza się bezpośrednio po nawiasie zamykającym listę parametrów konstruktora, a przed nawiasem klamrowym otwierającym *definicję* tego konstruktora.

Jeśli w klasie tylko deklarujemy konstruktor, a jego definicję podajemy poza klasą, to w deklaracji listy inicjalizacyjnej *nie* umieszczamy.

To jest logiczne: lista inicjalizacyjna należy logicznie do implementacji, a nie do interfejsu (kontraktu). Jak pamiętamy, odwrotnie było z argumentami domniemanymi (domyślnymi) — nie tylko konstruktorów, ale w ogóle funkcji; jeśli deklaracja występuje, to argumenty domniemane muszą być zdefiniowane właśnie w deklaracji, ale nie w definicji, bowiem ich wartość, i sama ich obecność, należy jak najbardziej do kontraktu z użytkownikiem (interfejsu).

Niezależnie od kolejności na liście inicjalizacyjnej,

składowe obiektu są inicjowane *zawsze* w kolejności ich deklaracji w ciele klasy.

Niektóre kompilatory wysyłają ostrzeżenia, jeśli kolejność deklaracji w definicji klasy i kolejność na liście inicjalizacyjnej nie są zgodne.

Na liście inicjalizacyjnej *nie* musimy wymieniać wszystkich składowych klasy. Te składowe, które nie zostały wymienione na liście, zainicjowane będą (przed rozpoczęciem wykonania ciała konstruktora!) przez:

- użycie konstruktorów domyślnych dla składowych obiektowych. Takie konstruktory muszą zatem istnieć;
- standardową inicjalizację dla składowych typów wbudowanych. Polega ona zwykle na tym, że żadna szczególna wartość w ogóle nie jest do zmiennej wpisywana, zatem zmienna ta, z naszego punktu widzenia, ma przypadkowy układ bitów.

Rozpatrzmy przykład: załóżmy, że mamy klasę **A** ze zdefiniowanym jednym konstruktorem, który jednak nie może pełnić roli konstruktora domyślnego, bo zawsze wymaga argumentów. Drugi konstruktor dostarczy wtedy kompilator: będzie to konstruktor kopiujący.

```
class A {
    //...
    A(int x, int y) { ... }
    //...
};
```

Załóżmy dalej, że obiekty klasy **A** są składowymi klasy **B**. Ponieważ obiekty te muszą istnieć przed rozpoczęciem konstruktora, a do ich utworzenia potrzebne są argumenty, ich inicjalizacja *musi* nastąpić poprzez listę inicjalizacyjną:

```
1     class B {
2         A pole1, pole2;
3         //...
4         B(A a, int x, int y)
5             : pole1(a), pole2(x,y)
6         {  }
7         //  ...
8     };
```

Konstruktor **B** jest zdefiniowany w liniach 4-6 — jego ciało jest puste, bo wszystko co jest do zrobienia jest robione poprzez listę inicjalizacyjną. Składowa pole1 będzie zainicjowana za pomocą konstruktora kopiującego klasy **A**, bo argumentem jest obiekt tej klasy, natomiast składowa pole2 za pomocą konstruktora dwuargumentowego, zdefiniowanego w klasie **A**. Konstruktora kopiującego, co prawda, nie zdefiniowaliśmy, ale dostarczy go system.

Podobna sytuacja zajdzie, gdy pole klasy jest zadeklarowane jako stałe. Stałe muszą być inicjowane już w momencie tworzenia, a zatem nadawanie im wartości dopiero w konstruktorze nie wchodzi w rachubę. Trzeba to zrobić już na liście inicjalizacyjnej. To samo dotyczy pól odnośnikowych (referencyjnych); jak pamiętamy, patrz rozdz. 4.7 na stronie 47, referencje, jak stałe, muszą być inicjowane już w momencie tworzenia.

Pola stałe stosuje się rzadko. Zazwyczaj wystarczy mechanizm ochrony danych poprzez umieszczenie składowej w sekcji prywatnej klasy. Również pola referencyjne nie występują często: składowa referencyjna jest inną nazwą czegoś spoza klasy, co stwarza niepotrzebną zwykle więź między obiektem a danymi spoza obiektu. Natomiast pola obiektowe, jak w powyższym przykładzie, występują często i w takich przypadkach stosowanie listy inicjalizacyjnej może być konieczne.

W poniższym przykładzie klasa **Punkt** nie ma konstruktora domyślnego. Istnieje tylko konstruktor zdefiniowany przez nas, pobierający dwie liczby typu **double**, oraz dostarczony przez system konstruktor kopiujący, który w tym przypadku jest odpowiedni i nie wymaga przedefiniowywania, gdyż klasa nie ma pól wskaźnikowych. Również destruktor nie jest potrzebny, gdyż z obiektem nie są związane żadne dane alokowane na stercie.

P126: *trian.cpp* Lista inicjalizacyjna

```
1 #include <iostream>
2 using namespace std;
3
4 struct Punkt {
5     double x, y;
6
7     Punkt(double x, double y)
```

```
8           : x(x), y(y)
9       { }
10
11      void show() const {
12          cout << "(" <<  x << "," << y << ")";
13      }
14  };
15
16  struct Trojkat {
17      Punkt a, b, c;
18
19      Trojkat(const Punkt&,const Punkt&,const Punkt&);    ①
20      Trojkat(double,double,double,double,double,double);②
21
22      void show() const {
23          cout << "Trojkat ";
24          a.show(); cout << "-";
25          b.show(); cout << "-";
26          c.show(); cout << endl;
27      }
28  };
29
30  Trojkat::Trojkat(const Punkt &a, const Punkt &b,
31                                   const Punkt &c)
32      : a(a), b(b), c(c)                                  ③
33  { }
34
35  Trojkat::Trojkat(double x1, double y1, double x2,
36                   double y2, double x3, double y3)
37      : a(x1,y1), b(x2,y2), c(x3,y3)                      ④
38  { }
39
40  int main() {
41      Punkt a1(1,1), b1(2,2), c1(3,3);
42
43      Trojkat T1(a1,b1,c1);
44
45      Trojkat T2(11,22,22,33,33,44);
46
47      T1.show();
48      T2.show();
49  }
```

Konstruktor klasy **Punkt** ma, jak widać, puste ciało; cała jego praca zostaje wykonana poprzez użycie listy inicjalizacyjnej. Zapis `x(x)` znaczy „zainicjuj składową `x` (to jest `x` zewnętrzne, poza nawiasem) posyłając do konstruktora wartość (lokalnej)

zmiennej x — czyli tej związanej z parametrem konstruktora". Oczywiście w naszym przypadku składowa jest typu wbudowanego; klasy **double** tak naprawdę nie ma, ale składnia wywołania konstruktora kopiującego, jak wspominaliśmy, może być użyta i w odniesieniu do typów wbudowanych.

Zauważmy, że w klasie **Punkt** lista inicjalizacyjna nie jest konieczna: składowe typów wbudowanych i tak byłyby utworzone bez kłopotów, a nadaniem im wartości moglibyśmy zająć się w ciele konstruktora.

Inaczej jest z klasą/strukturą **Trojkat**. Jej trzy pola są typu **Punkt**. A zatem podczas tworzenia obiektów klasy **Trojkat** muszą być utworzone, jeszcze przed wywołaniem konstruktora klasy **Trojkat**, trzy obiekty klasy **Punkt**, które są składowymi tego obiektu. Klasa **Punkt** nie ma jednak konstruktora domyślnego, zatem jedyną możliwością jest tu użycie listy inicjalizacyjnej. Odpowiednie konstruktory są tu zadeklarowane w liniach ① i ②, a zdefiniowane poza klasą. Zauważmy, że lista inicjalizacyjna pojawia się wyłącznie w definicjach konstruktorów, ale nie w ich deklaracjach. Pierwszy z konstruktorów pobiera poprzez argumenty trzy punkty i przekazuje je poprzez listę inicjalizacyjną (linia ③) do automatycznie wygenerowanego konstruktora kopiującego klasy **Punkt**. Drugi pobiera sześć liczb typu **double** i przekazuje je, parami, do konstruktora klasy **Punkt** pobierającego dwie liczby (linia ④). Wynik tego programu

```
Trojkat (1,1)-(2,2)-(3,3)
Trojkat (11,22)-(22,33)-(33,44)
```

został uzyskany za pomocą metod **show** w obu klasach. Zauważmy, że metoda ta w klasie **Trojkat** korzysta jawnie z tak samo nazwanej metody z klasy **Punkt**. Obie te metody zostały zdefiniowane jako stałe, gdyż ich rolą jest wydrukowanie informacji o obiektach, a nie jakakolwiek modyfikacja tych obiektów.

Przykład klasy z polami ustalonymi i referencyjnymi podamy w następnym podrozdziale.

Poczynając od wersji C++11, z listy inicjalizacyjnej jednego konstruktora można też wywołać inny konstruktor tej samej klasy, czyli jeden konstruktor „deleguje" pracę do drugiego (dlatego nazywamy go konstruktorem delegującym, *delegating constructor*). W takim przypadku na liście inicjalizacyjnej jednego konstruktora umieszczamy wyłącznie jeden element: nazwę danej klasy wraz z argumentami dla innego konstruktora. Ten inny konstruktor zostanie wtedy wykonany najpierw (zarówno to, co jest umieszczone na jego liście inicjalizacyjnej, jak i jego ciało), po czym sterowanie wraca do pierwotnie wywołanego konstruktora. Na przykład w programie

P127: ***delegconstr.cpp*** Konstruktory delegujące

```
#include <iostream>

class Point {
    double x, y;
public:
    Point(double x, double y) : x(x), y(y) {
        std::cerr << "CTOR 1: (double,double)\n";
```

```
8       }
9
10      Point(double x) : Point(x,0) {
11          std::cerr << "CTOR 2: (double)\n";
12      }
13      Point() : Point(0) {
14          std::cerr << "CTOR 3: ()\n";
15      }
16  };
17
18  int main() {
19      std::cerr <<   "Point p1(1,1)\n";
20      Point p1(1,1);
21      std::cerr << "\nPoint p2(2)\n";
22      Point p2(2);
23      std::cerr << "\nPoint p3\n";
24      Point p3;
25  }
```

jak przekonuje jego wydruk

```
Point p1(1,1)
CTOR 1: (double,double)

Point p2(2)
CTOR 1: (double,double)
CTOR 2: (double)

Point p3
CTOR 1: (double,double)
CTOR 2: (double)
CTOR 3: ()
```

podczas tworzenia obiektu `p3` konstruktor domyślny deleguje do konstruktora drugiego, a ten z kolei do konstruktora pierwszego: wykonanie wraca potem do konstruktora drugiego i następnie znów do trzeciego.

15.4 Funkcje zaprzyjaźnione

W poprzednim przykładzie wszystkie składowe klas **Punkt** i **Trojkat** były publiczne, gdyż w definicji tych klas użyliśmy słowa kluczowego **struct**, a nie **class**.

Najczęściej jednak tak nie jest — pola odpowiadające danym są zwykle prywatne. Często jednak chcielibyśmy, aby pewne funkcje *nie* będące metodami klasy miały dostęp do składowych niepublicznych tej klasy. Oczywiście efekt ten możemy osiągnąć definiując metody publiczne klasy udostępniające dane prywatne zawarte w obiekcie. Wtedy jednak *każda* funkcja może ich użyć i uzyskać dostęp do tych danych, podczas

gdy naszym celem było ich udostępnienie tylko wybranym funkcjom „zaprzyjaźnionym" z tą klasą. Takie pojęcie funkcji zaprzyjaźnionej rzeczywiście istnieje.

Funkcja taka musi być zadeklarowana wewnątrz klasy, z którą jest zaprzyjaźniona. Robi się to dodając na początku jej deklaracji modyfikator **friend**, na przykład

```
class Klasa {
    // ...
    friend int fun(double, const Klasa&);
    // ...
};
```

Deklarację przyjaźni można umieścić w dowolnej sekcji definicji klasy, publicznej, prywatnej lub chronionej — nie ma to żadnego znaczenia.

Funkcja zaprzyjaźniona z klasą *nie* jest metodą tej klasy.

Ma ona dostęp do wszystkich składowych klasy, ale nie będąc jej metodą, nie ma określonego wskaźnika **this** i nie jest wywoływana na rzecz obiektu. W szczególności, ta sama funkcja może być zaprzyjaźniona z wieloma klasami: deklaracja przyjaźni musi być wtedy zawarta w definicji każdej z tych klas.

Przyjaźń może być tylko zaofiarowana funkcji przez klasę. Funkcja natomiast nie może „żądać" przyjaźni od klasy. Nie ma sposobu, aby zaprzyjaźnić funkcję z klasą, która tej przyjaźni jawnie nie deklaruje; w szczególności nie da się zaprzyjaźnić funkcji z klasą pochodzącą z biblioteki której nie możemy czy nie chcemy zmodyfikować i zrekompilować.

Deklaracja przyjaźni zawarta jest wewnątrz definicji klasy, a więc jej zakresem jest zakres klasy. Jeśli w zakresie otaczającym potrzebna jest deklaracja tej samej funkcji (bo, na przykład, definicja jest w innym module programu, albo jej użycie następuje leksykalnie przed definicją), to należy taką deklarację powtórzyć poza klasą, oczywiście wtedy już bez modyfikatora **friend**. Wyjątkiem są funkcje zaprzyjaźnione z klasą, których *parametry* są typu tejże klasy (jak to zwykle ma miejsce); kompilator przegląda wtedy przestrzeń nazw klasy argumentu w poszukiwaniu deklaracji funkcji, co pozwoli mu ją odnaleźć bez deklaracji na zewnątrz klasy (jest to tzw. wyszukiwanie Koeniga, ang. *Koenig lookup*, lub ADL – *argument-dependent name lookup*).

Można zadeklarować w klasie przyjaźń ze wszystkimi metodami innej klasy: tak np., jeśli w klasie A zadeklarujemy:

```
class B;

class A {
    // ...
    friend class B;
    //...
};
```

to wszystkie metody klasy **B** będą miały dostęp do wszystkich składowych klasy **A**, ale nie odwrotnie. Przyjaźń bowiem nie jest relacją zwrotną: jeśli klasa **B** jest zaprzyjaźniona z klasą **A**, to nie oznacza to wcale, że klasa **A** jest zaprzyjaźniona z klasą **B**. Zauważmy, że w powyższym przykładzie potrzebna była deklaracja zapowiadająca klasy **B**, jeśli definicja tej klasy następuje po definicji klasy **A** (nie byłaby potrzebna, gdyby definicja **B** występowała przed definicją **A**).

Aby było również odwrotnie, to znaczy, aby również funkcje klasy **A** miały dostęp do składowych klasy **B**, można zadeklarować przyjaźń wzajemną klas:

```
class A;

class B {
    // ...
    friend class A;
    // ...
};

class A {
    // ...
    friend class B;
    // ...
};
```

Przyjaźń nie jest też relacją przechodnią. Jeśli klasa **A** jest zaprzyjaźniona z klasą **B**, a klasa **B** jest zaprzyjaźniona z klasą **C**, to *nie* znaczy, że klasa **A** jest zaprzyjaźniona z klasą **C**.

W końcu, przyjaźń nie jest też dziedziczna — klasy potomne *nie* dziedziczą przyjaźni od swoich klas bazowych.

W poniższym przykładzie definiujemy dwie klasy: jedną opisującą punkty na prostej rzeczywistej (o współrzędnej liczba) i drugą opisującą odcinki, a więc zakresy współrzędnych w przedziale [lewy, prawy]. Pola obu klas są prywatne.

P128: *iswew.cpp* Funkcje zaprzyjaźnione z dwoma klasami

```
1  #include <iostream>
2  using namespace std;
3
4  class Zakres;                                                    ①
5
6  class Punkt {
7
8      int liczba;
9      friend void isInside(const Punkt*, const Zakres*); ②
10
11 public:
12     Punkt(int liczba = 0)
```

```
13          : liczba(liczba)
14      { }
15 };
16
17 class Zakres {
18
19      int lewy, prawy;
20      friend void isInside(const Punkt*, const Zakres*);
21
22 public:
23      Zakres(int lewy = 0, int prawy = 0)
24          : lewy(lewy), prawy(prawy)
25      { }
26 };
27
28 void isInside(const Punkt *p, const Zakres *z) {
29      if ((p->liczba >= z->lewy) && (p->liczba <= z->prawy))
30          cout << "Punkt " << p->liczba   << " lezy w "
31                  "zakresie [" << z->lewy << ","
32               << z->prawy  << "]\n";
33      else
34          cout << "Punkt " << p->liczba   << " lezy poza "
35                  "zakresem [" << z->lewy << ","
36               << z->prawy  << "]\n";
37 }
38
39 int main() {
40      Punkt p(7);
41      Zakres z1(0,10), z2(8,20);
42
43      isInside(&p,&z1);
44      isInside(&p,&z2);
45 }
```

Z oboma klasami jest zaprzyjaźniona funkcja **isInside**, której zadaniem jest wypisanie informacji, czy dany punkt należy do zadanego zakresu czy nie. Aby sprawdzić, czy punkt przesłany do funkcji przez referencję leży w zakresie zadanym drugim argumentem funkcji, potrzebuje ona dostępu do danych zawartych w obu obiektach. Dzięki zaprzyjaźnieniu, taki dostęp posiada:

```
Punkt 7 lezy w zakresie [0,10]
Punkt 7 lezy poza zakresem [8,20]
```

Zauważmy, że deklaracja zapowiadająca z linii ① była konieczna, bo nazwa **Zakres** została użyta wewnątrz definicji klasy **Punkt** (linia ②).

Funkcje zaprzyjaźnione stosuje się szczególnie często przy przeładowywaniu operatorów (o czym więcej w rozdz. 19, str. 385).

Inne zastosowanie funkcji zaprzyjaźnionych to funkcje zwane fabrykami obiektów — zastępują one czasem, z punktu widzenia użytkownika klasy, konstruktory. Konstruktory, ze swej natury, są zwykle publiczne. Możliwa, i czasem wskazana, jest jednak sytuacja, gdy wszystkie lub niektóre konstruktory są prywatne. Tworzenie obiektów jest wtedy możliwe, jeśli istnieje funkcja zaprzyjaźniona tej klasy — wewnątrz takiej funkcji wszystkie składowe klasy, włączając konstruktory, są dostępne, a więc może być użyty również prywatny konstruktor (inna możliwość to zdefiniowanie w klasie publicznej funkcji statycznej zwracającej utworzony za pomocą prywatnego konstruktora obiekt).

Na zakończenie tego rozdziału rozpatrzmy przykład programu ilustrującego zarówno funkcje zaprzyjaźnione, jak i listy inicjalizacyjne dla pól stałych, odnośnikowych i obiektowych.

P129: *confiel.cpp* Pola stałe, referencyjne i obiektowe

```
#include <iostream>
#include <cstring>
using namespace std;

// UWAGA: klasy niekompletne; brak tu
// przeciążenia operatora przypisania

class Pracownik;                                    ①

enum stanowisko {zwykly, kierownik, prezes};        ②

class Osoba {
    char* nazwisko;
    int   rok_urodzenia;

      // deklaracja przyjazni
    friend void pracinfo(const Pracownik*);
public:
    Osoba(char* n, int r)
        : nazwisko(strcpy(new char[strlen(n)+1],n)),
          rok_urodzenia(r)
    { }

      // konstruktor kopiujacy
    Osoba(const Osoba& os)
        : nazwisko(strcpy(new
              char[strlen(os.nazwisko)+1],os.nazwisko)),
          rok_urodzenia(os.rok_urodzenia)
    { }

      // destruktor
```

```
32      ~Osoba() {
33          cout << "Usuwamy osobe " << nazwisko << endl;
34          delete [] nazwisko;
35      }
36  };
37
38  class Pracownik {
39      static int       ID;
40      Osoba          dane;
41      const int &zarobki;
42      const int        id;
43
44        // deklaracja przyjazni
45      friend void pracinfo(const Pracownik*);
46  public:
47      Pracownik(char* nazw, int rok, int& zar)
48          : dane(nazw,rok), zarobki(zar), id(++ID)
49      { }
50
51        // konstruktor kopiujacy
52      Pracownik(const Pracownik& prac)
53          : dane(prac.dane), zarobki(prac.zarobki), id(++ID)
54      { }
55  };
56  int Pracownik::ID;
57
58  void pracinfo(const Pracownik* prac) {
59      cout << prac->dane.nazwisko       << " (r.ur. "
60           << prac->dane.rok_urodzenia  << ") id="
61           << prac->id << "; zarobki: " << prac->zarobki
62                                        << endl;
63  }
64
65  int main() {
66      int placa[] = { 1600, 2100, 8900 };
67
68      Pracownik  jasio("Jasio  ", 1978, placa[zwykly]);
69      Pracownik  henio("Henio  ", 1980, placa[zwykly]);
70      Pracownik    jan("Jan    ", 1965, placa[kierownik]);
71      Pracownik panJan("Pan Jan", 1955, placa[prezes]);
72
73      pracinfo(&jasio);
74      pracinfo(&henio);
75      pracinfo(&jan);
76      pracinfo(&panJan);
77
```

```
78      cout << "\nZmieniamy place\n\n";
79
80      placa[zwykly] -=  300;                          ③
81      placa[prezes] += 1000;                          ④
82
83      pracinfo(&jasio);
84      pracinfo(&henio);
85      pracinfo(&jan);
86      pracinfo(&panJan);
87
88      cout << "\nKoniec programu\n\n";
89  }
```

Definiujemy tu w linii ② wyliczenie **stanowisko**. Następnie definiujemy dwie klasy: **Osoba** i **Pracownik**. Klasa **Osoba** jest standardowa. Użyliśmy tu list inicjalizacyjnych w konstruktorach tylko ze względu na wydajność; pól stałych, obiektowych ani referencyjnych nie ma, więc równie dobrze mogłyby to być zwykłe konstruktory.

Wewnątrz klasy deklarujemy przyjaźń z funkcją **pracinfo**. Ponieważ parametrem tej funkcji jest wskaźnik typu **const Pracownik***, a klasa **Pracownik** nie była jeszcze zdefiniowana, musieliśm w linii ① dostarczyć deklarację zapowiadającą.

Klasa **Pracownik** jest bardziej wyszukana. Ma jedno pole statyczne (ID), jedno obiektowe, jedno ustalone referencyjne i jedno ustalone. Zatem z wyjątkiem składowej statycznej, która i tak fizycznie nie wchodzi w skład obiektu, żadnej innej składowej nie da się utworzyć w konstruktorze — trzeba to zrobić posługując się listą inicjalizacyjną i to w każdym konstruktorze, również w konstruktorze kopiującym.

Klasa **Pracownik** deklaruje również przyjaźń z funkcją **pracinfo**. Funkcja ta bowiem drukuje informacje o obiekcie klasy **Pracownik**, a więc musi mieć dostęp do jego składowych. Bez przyjaźni byłoby to niemożliwe, gdyż wszystkie składowe są prywatne. Ponieważ funkcja nie modyfikuje obiektów klasy **Pracownik**, jej parametr wskaźnikowy został zadeklarowany z modyfikatorem **const**. Jedną ze składowych obiektu klasy **Pracownik** jest obiekt klasy **Osoba** — w niej też pola są prywatne. Aby móc wydrukować również informacje o osobie wchodzącej w skład pracownika, funkcja **pracinfo** musiała zatem być zaprzyjaźniona z klasą **Osoba**.

W funkcji **main** tworzymy cztery obiekty klasy **Pracownik**. Tworząc je musimy przesłać też do konstruktorów dane dotyczące obiektu klasy **Osoba**, który będzie utworzony jako podobiekt obiektu klasy **Pracownik**.

Zauważmy, że składową referencyjną zarobki inicjujemy referencją do jednego z elementów tablicy placa. A zatem w zakresie obiektu nazwa zarobki jest inną nazwą któregoś z elementów tej tablicy. Elementy tablicy indeksujemy tu wartościami wyliczenia: oczywiście są one konwertowane do wartości całkowitych 0, 1 i 2, ale użycie wyliczenia pozwala indeksować tablicę za pomocą nazw coś użytkownikowi mówiących.

Po utworzeniu obiektów drukujemy informacje o nich:

```
Jasio   (r.ur. 1978) id=1; zarobki: 1600
Henio   (r.ur. 1980) id=2; zarobki: 1600
Jan     (r.ur. 1965) id=3; zarobki: 2100
```

```
Pan Jan (r.ur. 1955) id=4; zarobki: 8900

Zmieniamy place

Jasio   (r.ur. 1978) id=1; zarobki: 1300
Henio   (r.ur. 1980) id=2; zarobki: 1300
Jan     (r.ur. 1965) id=3; zarobki: 2100
Pan Jan (r.ur. 1955) id=4; zarobki: 9900

Koniec programu

Usuwamy osobe Pan Jan
Usuwamy osobe Jan
Usuwamy osobe Henio
Usuwamy osobe Jasio
```

Następnie w liniach ③ i ④ zmieniamy wartości tablicy place i ponownie drukujemy informacje o wszystkich obiektach. Widzimy, że wydrukowane składowe zarobki zostały zmienione! Tak oczywiście musiało być, gdyż są one tylko innymi nazwami elementów tablicy place. Ale zauważmy, że składowe te są, po pierwsze, prywatne, po drugie stałe! Zatem „podwójnie" nie powinna być możliwa zmiana ich wartości w funkcji **main**. A jednak jest możliwa! Pokazuje to ponownie, że tak naprawdę chronione są nie zmienne (obszary pamięci), ale nazwy: jeśli do składowej prywatnej czy stałej możemy odnieść się poprzez inną nazwę, to żadnej ochrony nie ma.

Wydruk demonstruje też działanie destruktorów klasy **Osoba**. Widzimy, że zadziałały one już po wyjściu programu z funkcji **main**, i że wywoływane są w kolejności odwrotnej do tej, w jakiej obiekty były utworzone. Obiekty klasy **Osoba** usuwane są tu podczas usuwania obiektów klasy **Pracownik** jako ich składowe.

15.5 Klasy zagnieżdżone

Możliwe jest definiowanie klas wewnątrz definicji klasy. Takie klasy, których definicja jest zanurzona w definicji innej klasy, nazywamy klasami **zagnieżdżonymi** lub klasami **wewnętrznymi**. Klasę, we wnętrzu której podana jest definicja klasy zagnieżdżonej, nazywamy **klasą otaczającą**. Zagnieżdżenie definicji oznacza, że nazwa tej klasy leży w zakresie klasy otaczającej, a nie w zakresie globalnym. W zasadzie nie znaczy nic więcej. Klasy otaczająca i zagnieżdżona nie są specjalnie ze sobą związane. W szczególności, w odróżnieniu od Javy, obowiązują dla nich normalne zasady dostępności, czyli na przykład pola prywatne lub chronione jednej z nich nie są widoczne w drugiej. Z tego, że zakresem definicji klasy zagnieżdżonej jest klasa otaczająca, wynika, że widoczne są w niej, bez żadnych kwalifikacji, nazwy aliasów zdefiniowanych za pomocą **typedef** wewnątrz klasy otaczającej, czy na przykład nazwy zdefiniowanych w niej typów wyliczeniowych.

Rozpatrzmy jako przykład następujący program:

P130: ***klawew.cpp*** Klasy zagnieżdżone

```
#include <iostream>
#include <cstring>
using namespace std;

class Klient {
    static int ID;
    char*  nazwisko;
    const  int id;
    int    ma,winien;

public:
    class Saldo {
        int id;
        int saldo;
    public:
        Saldo(int id, int saldo)
            : id(id), saldo(saldo)
        {}

        void printinfo();
    };

    Klient(const char* n)
        : nazwisko(strcpy(new char[strlen(n)+1],n)),
          id(++ID), ma(0), winien(0)
    {}

    Klient& wplata(int w)  { ma     += w; return *this; }
    Klient& wyplata(int w) { winien += w; return *this; }

    Saldo* dajSaldo();

    ~Klient() { delete [] nazwisko; }
};
int Klient::ID = 0;

Klient::Saldo* Klient::dajSaldo() {
    return new Saldo(id, ma - winien);
}

void Klient::Saldo::printinfo() {
    cout << "id: " << id << " Saldo: " << saldo << endl;
}

```

```
45 int main() {
46     Klient  kow("Kowalski");
47     Klient* pmal = new Klient("Malinowski");
48 
49     kow.wplata(100).wplata(50).wyplata(75);
50     pmal->wplata(200).wyplata(50).wyplata(25);
51 
52     Klient::Saldo* pskow =   kow.dajSaldo();
53     Klient::Saldo* psmal = pmal->dajSaldo();
54 
55     pskow->printinfo();
56     psmal->printinfo();
57 
58     Klient::Saldo anonim(9,500);
59     anonim.printinfo();
60 
61     delete pskow;
62     delete psmal;
63     delete pmal;
64 }
```

Wewnątrz klasy **Klient** zdefiniowna jest klasa **Saldo**. Jej metoda **printinfo** jest w klasie zadeklarowana, ale zdefiniowana poza klasą. Zauważmy, że w lini 41 musieliśmy do jej zdefiniowania użyć podwójnej kwalifikacji: jest to funkcja **printinfo** z zakresu klasy **Saldo**, który z kolei zawarty jest w zakresie klasy **Klient**.

W liniach 52 i 53 tworzymy zmienne wskaźnikowe typu **Saldo***. Zauważmy, że i tutaj musieliśmy użyć kwalifikacji nazwą klasy otaczającej, ponieważ w zakresie globalnym nazwa klasy **Saldo** jest niewidoczna.

Nie jest jednak tak, że nie da się tworzyć obiektów klasy **Saldo** poza klasą otaczającą (czyli klasą **Klient**). Jak widzimy w linii 58, i o czym przekonuje nas wydruk programu

```
id: 1 Saldo: 75
id: 2 Saldo: 125
id: 9 Saldo: 500
```

można utworzyć obiekt klasy **Saldo** w funkcji **main** w ogóle nie odwołując się do żadnego obiektu ani metody klasy otaczającej (inaczej jest w Javie).

Klasy wewnętrzne występują w programach C++ rzadko, bo w zasadzie nie są nigdy absolutnie konieczne, choć mogą być wygodne.

Na marginesie zauważmy w tym miejscu, że możliwe jest też definiowanie **klas lokalnych**, definiowanych wewnątrz *funkcji*. Wtedy rzeczywiście nazwa tej klasy nie jest dostępna nigdzie poza tą funkcją, tak jak nie są dostępne nazwy żadnych zmiennych lokalnych funkcji. Klasy lokalne stosuje się jeszcze rzadziej niż klasy zagnieżdżone.

15.6 Wskaźniki do składowych

Obiekty tej samej klasy zawsze mają tę samą długość w pamięci. Poszczególne składowe w każdym obiekcie są wewnątrz tego obiektu usytuowane jednakowo, czyli ich początek jest zawsze tak samo przesunięty względem początku obiektu. Pozwala to na istnienie specjalnego rodzaju wskaźników, **wskaźników do składowych**, których wartością nie jest bezwzględny adres składowej obiektu, ale jej przesunięcie względem początku obiektu. Gdy znany jest adres obiektu (jego początku) oraz ten właśnie wskaźnik do składowej, system może wyliczyć z tych informacji adres bezwzględny tej składowej.

Wskaźnik do składowej typu **Typ** w klasie **Klasa** definiuje się tak:

```
Typ Klasa::*wskaz;
```

i oznacza on, że wartością zmiennej wskaz będzie przesunięcie pewnej *publicznej* i *niestatycznej* składowej typu **Typ** w obiektach klasy **Klasa**. Widać, że definicja ta różni się od definicji zwykłego wskaźnika obecnością specyfikacji zakresu klasy (w naszym przypadku 'Klasa::'). Tak zdefiniowany wskaźnik do składowej nie ma na razie żadnej sensownej wartości.

Przypuśćmy, że klasa **Klasa** ma dwa publiczne, niestatyczne pola, pole1 i pole2, oba typu **Typ**. Można wtedy przypisać

```
wskaz = &Klasa::pole1;
```

albo

```
wskaz = &Klasa::pole2;
```

Symbol '&' nie oznacza tu pobrania bezwzględnego adresu jakiegoś obiektu — przypisanie to oznacza, że wartością wskaźnika do składowej wskaz będzie przesunięcie składowej pole1 (albo, w drugim przypadku, pole2) względem początku *dowolnego* obiektu klasy **Klasa**. Zauważmy, że to przesunięcie jest bezużyteczne, jeśli nie wiemy, o jaki obiekt chodzi, bo nie wiadomo względem czego przesuwać. Po takiej definicji, dla konkretnego obiektu obiekt klasy **Klasa** można teraz odwołać się do jego składowej pole1 poprzez operator '.*':

```
obiekt.*wskaz
```

Tutaj obiekt wskazuje, o który obiekt chodzi, a wskaz mówi gdzie, w obrębie obiektu, szukać odpowiedniej składowej. Gdybyśmy dysponowali nie nazwą obiektu, ale wskaźnika do niego,

```
Klasa* wsk_do_obiektu = &obiekt;
```

to do tej samej składowej tego obiektu moglibyśmy się odnieść za pomocą operatora '->*'

```
wskaznik_do_obiektu->*wskaz
```

W podobny sposób można definiować wskaźniki do składowych metod. Teraz trudno to sobie wyobrazić jako przesunięcie względem początku obiektu, bo metody nie są zawarte fizycznie w obiektach. Jest problemem implementatorów kompilatorów C++, jak to zostanie rozwiązane: z punktu widzenia programisty zasada jest taka sama jak dla pól klasy.

Jeśli na przykład w klasie **Klasa** są dwie metody, obie typu **double** → **double**,

```
double fun1(double);
double fun2(double);
```

to wskaźnik do składowej wf mogący wskazywać na jedną z tych metod można zdefiniować tak (uwaga na nawiasy!):

```
double (Klasa::*wf)(double);
```

Czytamy: wf jest wskaźnikiem do składowej w klasie **Klasa** wskazującym na metodę (publiczną i niestatyczną) pobierającą **double** i zwracającą **double** (patrz rozdz. 11.12 o wskaźnikach funkcyjnych, str. 184). I znowu, po takiej definicji wskaźnik wf nie wskazuje na nic sensownego. Możemy teraz dokonać przypisania

```
wf = Klasa::fun1;
```

i teraz wskaźnik wf będzie wskazywał na metodę **fun1**. Zauważmy, że w definicji podajemy tylko nazwę metody — nie ma nawiasów, które oznaczałyby jej wywołanie. Ponieważ jest to metoda, aby użyć tego wskaźnika i poprzez niego tę metodę wywołać, musimy określić, o który obiekt nam chodzi, a więc na rzecz którego obiektu ma nastąpić wywołanie. Jak dla pól, możemy to zrobić za pomocą operatora '.*' lub '->*', czyli:

```
(obiekt.*wf)(5.5)
```

albo, jeśli dysponujemy wskaźnikiem do obiektu,

```
(wskaznik_do_obiektu->*wf)(5.5)
```

Nawiasy są tu konieczne, ze względu na priorytet operatorów.

Często używa się takich wskaźników do wskazywania metod klasy przy projektowaniu różnego rodzaju menu. Definiuje się wtedy tablicę takich wskaźników, które wskazują na różne metody wywoływane po dokonaniu pewnego wyboru przez użytkownika. Na przykład:

```
double (Klasa::*wf[8])(double);
Klasa menu;
// ...
wf[0] = Klasa::fun1;
```

```
wf[1] = Klasa::fun2;
// ...
cin >> k;
(menu.*wf[k])( argument );
// ...
```

Przyjrzyjmy się następującemu programowi:

P131: *wsklas.cpp* Wskaźniki do składowych

```
#include <iostream>
#include <cmath>
using namespace std;

struct Punkt {
    double x, y;
    Punkt(double x = 0, double y = 0)
        : x(x), y(y)
    { }
    double r2() { return  x*x + y*y; }
    double dd() { return sqrt(r2()); }
};

int main() {
    double  Punkt::*wi[2];
    double (Punkt::*wf[2])();

    wi[0] = &Punkt::x;
    wi[1] = &Punkt::y;

    wf[0] = &Punkt::r2;
    wf[1] = &Punkt::dd;

    Punkt P(3,4), *p = &P;

    cout << " P.*wi[0]      = " <<  P.*wi[0]      << endl;
    cout << " P.*wi[1]      = " <<  P.*wi[1]      << endl;
    cout << "(P.*wf[0])()   = " << (P.*wf[0])()   << endl;
    cout << "(P.*wf[1])()   = " << (P.*wf[1])()   << endl;

    cout << endl;

    cout << " p->*wi[0]     = " <<  p->*wi[0]     << endl;
    cout << " p->*wi[1]     = " <<  p->*wi[1]     << endl;
    cout << "(p->*wf[0])() = " << (p->*wf[0])() << endl;
    cout << "(p->*wf[1])() = " << (p->*wf[1])() << endl;
}
```

Definiujemy tu klasę z dwoma polami typu **double** i dwoma metodami typu **double** → **double**. W funkcji **main** definiujemy dwie tablice wskaźników do składowych: tablicę wi wskaźników do pól i tablicę wf wskaźników do metod. W liniach 18-22 przypisujemy im wartości, a następnie używamy ich w liniach 26-36 ilustrując to, o czym mówiliśmy wyżej. Wydruk z programu

```
 P.*wi[0]      = 3
 P.*wi[1]      = 4
(P.*wf[0])()   = 25
(P.*wf[1])()   = 5

 p->*wi[0]     = 3
 p->*wi[1]     = 4
(p->*wf[0])()  = 25
(p->*wf[1])()  = 5
```

przekonuje nas, że wszystko działa zgodnie z przewidywaniem.

— 16 —

Operacje wejścia/wyjścia

W tym rozdziale przerwiemy na pewien czas omawianie klas, aby lepiej poznać sposoby wprowadzania i wyprowadzania danych. Do tej pory stosowaliśmy wyłącznie najbardziej elementarne mechanizmy zapewniane przez przeciążone operatory '<<' i '>>'. Wystarczało to do podstawowych zastosowań, ale istnieją znacznie bogatsze mechanizmy, zapewniające większą elastyczność operacji we/wy.

Oczywiście źródłem i ujściem danych nie musi być terminal (klawiatura i ekran). Jak się przekonamy, te same lub bardzo podobne narzędzia zapewniają możliwość pisania i czytania także wtedy, gdy źródłem lub ujściem danych jest na przykład plik, obszar pamięci czy gniazdo internetowe.

W C++ operacje we/wy realizowane są inaczej niż w tradycyjnym C. Dlatego to, o czym piszemy w tym rozdziale, *nie* stosuje się do programów napisanych w czystym C.

Podrozdziały:

16.1 Strumienie

Operacje we/wy realizuje się w C++ za pomocą **strumieni**. Strumień możemy wyobrażać sobie jako strumień informacji, w postaci bajtów, płynący od źródła do ujścia. Zachodzą przy tym dwie możliwości:

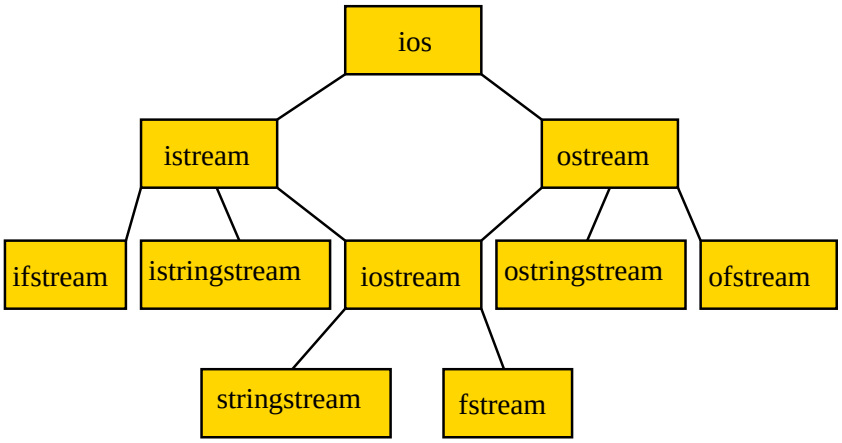
- informacja płynie *od* naszego programu do ujścia. Ujściem może być terminal, plik, gniazdo internetowe, obszar w pamięci, nazwany potok, systemowa kolejka

FIFO, modem itd. O takim strumieniu mówimy, że jest strumieniem *wyjściowym* (ang. *output stream*),

- informacja płynie *do* naszego programu ze źródła. Źródłem może być terminal, plik itd. O takim strumieniu mówimy, że jest strumieniem *wejściowym* (ang. *input stream*).

Do reprezentowania strumieni i obsługi operacji na nich zdefiniowane są specjalne klasy wyposażone w pola i metody. Struktura tych klas jest skomplikowana, ale na szczęście nie musimy jej dokładnie znać, aby z obiektów tych klas i ich składowych korzystać.

Ogólny, uproszczony schemat klas przedstawiony jest na rysunku (klasy **istrstream**, **ostrstream** i **strstream**, omówione poniżej, nie należą do tej hierarchii):



Podstawową dla nas klasą jest klasa **ios** (sama wyprowadzona z **ios_base**); z niej wyprowadzone są (dziedziczą) klasy:

- **istream** — podstawowa klasa reprezentująca strumienie wejściowe. W niej, między innymi, zdefiniowane jest przeciążenie operatora '>>' i tej właśnie klasy obiektem jest cin, reprezentujący standardowy strumień wejściowy (znany jako **stdin**). Bardziej wyspecjalizowane klasy do obsługi strumieni wejściowych to:
  - **istringstream** — źródłem jest obiekt klasy **string**, czyli napis. Dostępna po dołączeniu za pomocą dyrektywy `#include` pliku ***sstream***.
  - **istrstream** — źródłem jest C-napis (czyli tablica znaków ze znakiem '\0' jako ostatnim). Udostępniana w pliku nagłówkowym ***strstream***. UWAGA: ta klasa *nie* należy do standardu!
  - **ifstream** (z ang. *intput file stream*) — źródłem jest plik. Udostępniana po dołączeniu ***fstream***.
- **ostream** — podstawowa klasa reprezentująca strumienie wyjściowe. W niej właśnie zdefiniowane jest przeciążenie operatora '<<' i obiekty cout, reprezentujący

standardowy strumień wyjściowy (zwany **stdout**) oraz cerr i clog, reprezentujące standardowy strumień błędów (**stderr**) niebuforowany i buforowany, a także manipulatory endl i ends. Bardziej wyspecjalizowane klasy do obsługi strumieni wyjściowych to:

- **ostringstream** — ujściem jest obiekt klasy **string**, czyli napis. Udostępniana po dołączeniu pliku ***sstream***.
- **ostrstream** — ujściem jest C-napis, czyli tablica znaków ze znakiem '\0' jako ostatnim. Udostępniana po dołączeniu pliku nagłówkowego ***strstream***. UWAGA: ta klasa *nie* należy do standardu!
- **ofstream** (z ang. *output file stream*) — ujściem jest plik. Udostępniana po dołączeniu pliku nagłówkowego ***fstream***.

Klasa **iostream**, z nagłówka o tej samej nazwie, dziedziczy zarówno z **istream**, jak i z **ostream**, zatem dołączając ***iostream*** zapewniamy funkcjonalność obu. Podobnie, dołączając **fstream** zapewniamy dostęp do klas **ifstream** i **ofstream**, a dołączając **sstream** do **istringstream** i **ostringstream**.

Generalnie zatem:

- jeśli w naszym programie wykonujemy tylko konsolowe operacje we/wy (pisanie na ekran, czytanie z klawiatury), to wystarczy dołączyć plik nagłówkowy ***iostream***;
- jeśli wykonujemy operacje na plikach, to trzeba włączyć plik nagłówkowy ***fstream***;
- jeśli wykonujemy operacje we/wy, dla których źródłem lub ujściem są obiekty klasy **string**, to należy dołączyć ***sstream***.
- jeśli wykonujemy operacje we/wy, dla których źródłem lub ujściem są C-napisy, to należy dołączyć ***strstream***.

16.1.1 Strumienie predefiniowane

Po dołączeniu pliku nagłówkowego ***iostream*** mamy do dyspozycji cztery już otwarte strumienie: jeden wejściowy — cin, i trzy wyjściowe — cout, cerr i clog. Nazwy te są w rzeczywistości nazwami obiektów reprezentujących strumienie; dlatego, jak się przekonamy, można na ich rzecz wywoływać metody. Predefiniowane strumienie to:

- cin — standardowy strumień wejściowy. Źródłem jest standardowy strumień wejściowy (stdin) przydzielony procesowi wykonującemu program przez system operacyjny; zwykle jest to klawiatura, ale można to zmienić.
- cout — standardowy strumień wyjściowy. Ujściem jest standardowy strumień wyjściowy (stdout) przydzielony procesowi wykonującemu program przez system operacyjny; zwykle jest to ekran terminala, ale często jest przekierowywany do pliku. Strumień ten jest **buforowany**, a więc wpisane tam znaki mogą pojawiać się na ekranie z opóźnieniem, albo w ogóle się nie pojawić, jeśli program zakończył się przedwcześnie i bufor wyjściowy nie zdążył zostać opróżniony.

- cerr — standardowy strumień komunikatów o błędach. Ujściem jest standardowy strumień komunikatów o błędach (stderr), przydzielony procesowi wykonującemu program przez system operacyjny; zwykle jest to ekran terminala. Strumień ten *nie* jest buforowany, a więc wpisane tam znaki powinny ukazać się na ekranie natychmiast i być widoczne nawet jeśli zaraz potem program załamie się.
- clog — to również strumień związany z stderr, tyle, że buforowany.

Ze wszystkich tych strumieni można korzystać w znany nam już sposób z użyciem operatorów '<<' (wyjście, czyli pisanie) lub '>>' (wejście, czyli czytanie). Można używać też metod, które będą opisane w dalszym ciągu tego rozdziału.

16.2 Operatory << i >>

Czym są operatory '<<' i '>>'? Pamiętamy z rozdz. 9 (str. 121), że na przykład instrukcja '`a + b`' interpretowana jest tak naprawdę jako wywołanie funkcji dodającej, a wartości wyrażeń a i b, po ewentualnej konwersji, są do tej funkcji przesyłane jako argumenty. Podobnie jest dla operatorów '<<' i '>>'. Ich użycie powoduje wywołanie odpowiedniej funkcji dwuargumentowej, przy czym jako argumenty przesyłana jest do niej referencja do obiektu-strumienia występującego po lewej stronie i wartość wyrażenia po prawej stronie operatora. Funkcja ta jest przeciążona; wybór konkretnej wersji zależy od typu prawego argumentu. Tak więc dla typów wbudowanych (i wielu bibliotecznych) działanie operatora jest dobrze określone; dla typów definiowanych przez programistę można samemu dostarczyć dodatkowe wersje, co zademonstrujemy w rozdz. 19 na stronie 385.

Bardzo ważne jest to, że funkcja wywoływana przez użycie operatora '<<' lub '>>' jest funkcją rezultatową: prócz wykonania samej operacji we/wy, zwraca jako rezultat referencję do tego samego obiektu-strumienia, który był jej pierwszym (lewym) argumentem. Zatem w

```
(cout << x) << y;
```

wartość wyrażenia w nawiasie jest równoważna cout, które w ten sposób staje się ponownie lewym argumentem kolejnego operatora '<<'. Ponieważ wiązanie operatora '<<' jest lewe, więc użycie nawiasów w powyższym przykładzie nie jest konieczne; uzyskujemy dzięki temu możliwość „kaskadowego" wywołania odpowiednich funkcji:

```
cout << x << y << z << v << endl;
```

lub

```
cin >> x >> y >> z >> v;
```

gdzie, jak pamiętamy, kierunek „strzałek" można rozumieć jako kierunek przepływu informacji.

Aby wyprowadzić, na ekran czy do pliku, napis reprezentujący na przykład wartość zmiennej typu **double**, trzeba przyjąć jakąś umowę co do sposobu formatowania.

Sposób ten, jak się przekonamy, można zmieniać. Domyślnie zmienne typów wbudowanych są wyprowadzane zgodnie z następującymi domniemaniami:

Strumienie wyjściowe:

- wartości całościowe (jak **int**, **short**, itd.): w systemie dziesiątkowym;
- wartości znakowe: jako pojedyncze znaki;
- wartości zmiennopozycyjne (czyli **float**, **double** i **long double**): w systemie dziesiątkowym, z precyzją 6 cyfr. Precyzja oznacza liczbę cyfr znaczących, a *nie* liczbę cyfr po kropce dziesiętnej. Na przykład, wypisanie w formacie domyślnym liczby 200.0/3 daje 66.6667, a nie 66.666667. Końcowe zera po kropce dziesiętnej nie są pisane. Tak więc wartość 1.129996 zostanie najpierw zaokrąglona do sześciu cyfr znaczących, dając 1.13000, a następnie wypisana jako 1.13 (ale na przykład wydruk 1.129994 da 1.12999). Jeśli po kropce wystąpiłyby same zera, to opuszczane są i zera, i sama kropka dziesiętna. Jeśli liczba cyfr przed kropką wynosi lub przekracza sześć, to wypisane zostaną wszystkie, a część ułamkowa zostanie pominięta;
- wartości wskaźnikowe (prócz typu **char***): jako dodatnie liczby całkowite reprezentujące adresy w pamięci. Adresy wypisywane są w układzie szesnastkowym, a więc z prefiksem '0x';
- wartości wskaźnikowe typu **char***: jako napisy, interpretując kolejne bajty w pamięci, poczynając od bajtu wskazywanego przez wskaźnik, jako zapis kolejnych znaków napisu; wypisywanie kończy się po napotkaniu bajtu zerowego (odpowiadającego znakowi końca napisu '\0');
- wartości logiczne (typu **bool**): jako liczby 1 i 0.

Strumienie wejściowe:

- wiodące białe znaki są pomijane;
- każda następna niepusta sekwencja białych znaków jest interpretowana jako koniec danych. Znaki te pozostają w buforze i będą pominięte, jako wiodące, przy następnym czytaniu;
- liczby całkowite wczytywane są w postaci dziesiętnej; pierwszym niebiałym znakiem może być znak '+' lub '−', następnie wczytywane są cyfry aż do napotkania znaku nie będącego cyfrą — ten znak jest pozostawiany w buforze i będzie pierwszym znakiem wczytanym przez następną operację czytania. Na przykład, jeśli program czyta za pomocą instrukcji '`cin >> x >> y`', to można podać jako dane 128-25; wczytywanie 128 na `x` zakończy się na znaku minus, który pozostanie w strumieniu. Następnie będzie kontynuowane czytanie na `y` znaków -25;
- liczby zmiennopozycyjne wczytywane są w formacie liczb całkowitych bez kropki dziesiętnej, w formacie z kropką dziesiętną i w notacji „naukowej" (na przykład 1e-1 to to samo co 0.1, 1.201e+2 to 120.1);
- wartości logiczne mają postać literałów 1 i 0.

16.3 Formatowanie

Opisany wyżej sposób formatowania można zmienić. W szczególności dotyczy to operacji wyjścia, przy których często chcielibyśmy uzyskiwać wyniki w zaplanowanej formie, tak aby zapewnić czytelność lub zgodność z ustalonymi wymaganiami. Służą do tego flagi formatowania i manipulatory.

16.3.1 Flagi formatowania

Sposób działania operatorów we/wy określony jest aktualnym stanem związanej z każdym strumieniem **flagi stanu formatowania**.

Do manipulowania flagą stanu formatowania danego strumienia służą flagi zdefiniowane w klasie bazowej **ios_base** — a tym samym w klasie pochodnej **ios** — jako składowe statyczne. Odwołujemy się do zatem do nich poprzez operator zakresu klasy, np. ios::left, ios::scientific itd. Flagi te są, jak i cała flaga stanu formatowania, typu **ios_base::fmtflags** (wiele starszych kompilatorów nie definiuje takiego typu; w takiej sytuacji można zazwyczaj użyć typu **long**). Zauważmy, że nazwa typu **fmtflags** jest zadeklarowana w klasie **ios_base**, więc musimy się do niej odwoływać poprzez nazwę kwalifikowaną **ios_base::fmtflags** (lub po prostu **ios::fmtflags**).

Reprezentacja bitowa poszczególnych flag zawiera z reguły jeden ustawiony bit (czyli jedynkę na pewnej pozycji), a pozostałe bity nieustawione. Wynika z tego, że flagi można „ORować" za pomocą alternatywy bitowej, aby uzyskać flagę stanu formatowania o żądanych własnościach.

Zanim podamy przykłady, wyliczmy dostępne flagi formatowania, z których, za pomocą alternatywy bitowej, można zbudować flagę stanu. Flagi boolalpha, showbase, showpoint, showpos, skipws, unitbuf i uppercase mają też odpowiedniki o odwrotnym działaniu — ich nazwy mają prefix 'no', na przykład noboolalpha, noshowbase itd.

ios::skipws — zignoruj, przy wczytywaniu, wiodące białe znaki (domyślnie: TAK).

ios::left, ios::right,ios::internal — wyrównanie przy pisaniu do lewej, prawej albo obustronne, czyli „znak (lub inny prefix, jak na przykład 0x), do lewej, liczba do prawej". Na przykład, jeśli zapisujemy liczbę -123 w polu o szerokości 8 znaków, to stosując te trzy sposoby wyrównania otrzymalibyśmy

```
|-123    |
|    -123|
|-    123|
```

Te trzy flagi razem tworzą pole justowania (wyrównywania) ios::adjustfield. Najwyżej jedna z nich może być ustawiona (patrz dalej). Jeśli żadna nie jest ustawiona, to wyrównanie jest do prawej. Flaga internal, jako cokolwiek dziwaczna, w niektórych implementacjach jest ignorowana.

ios::dec, ios::hex, ios::oct — podstawa dla wczytywanych/pisanych liczb całkowitych: dziesiętna (domyślnie), szesnastkowa i ósemkowa. Razem tworzą pole podstawy ios::basefield. Najwyżej jedna z nich może być ustawiona; jeśli żadna nie jest

ustawiona, to odczyt/zapis jest dziesiętny.

ios::scientific, ios::fixed — format dla wypisywanych liczb zmiennopozycyjnych. Razem tworzą pole formatu zmiennopozycyjnego ios::floatfield. Najwyżej jedna z nich może być ustawiona. Jeśli żadna nie jest ustawiona, to użyty będzie format ogólny. W notacji naukowej (flaga scientific) wypisywana jest jedna cyfra przed kropką, maksymalnie tyle cyfr po kropce, ile wynosi aktualna precyzja, następnie litera 'e' i wykładnik potęgi dziesięciu, przez którą należy pomnożyć liczbę znajdującą się przed literą 'e'. Jeśli liczba jest ujemna, to przed nią wypisywany jest znak minus. Na przykład 1.123456e2 oznacza 112.3456, natomiast -1.123456e-3 oznacza -0.001123456. Format ustalony (flaga fixed) oznacza wypisywanie maksymalnie tylu cyfr po kropce dziesiętnej, ile wynosi aktualna precyzja. Format ogólny pozostawia implementacji sposób zapisu (naukowy lub normalny) w zależności od wartości liczby tak, żeby zapis z ustaloną precyzją zajmował jak najmniej miejsca.

ios::boolalpha — czy wartości logiczne wypisywać jako 0 i 1 (domyślnie), czy słowami jako **true** i **false**. W niektórych implementacjach znacznik ten nie jest zdefiniowany.

ios::showbase — przy wypisywaniu zawsze pokaż podstawę (wiodące 0 lub 0x w systemie ósemkowym i szesnastkowym).

ios::showpoint — zawsze wypisz kropkę dziesiętną i końcowe zera w części ułamkowej (domyślnie: NIE).

ios::showpos — pisz znak '+' przed liczbami dodatnimi (domyślnie: NIE).

ios::uppercase — litery 'e' w zapisie naukowym i 'x' w szesnastkowym pisz jako duże 'E' i 'X' (domyślnie: NIE).

ios::unitbuf — opróżnij bufor po każdej operacji zapisu (domyślnie: NIE).

Jak to zaznaczyliśmy w opisie, niektóre flagi tworzą grupy, zwane polami. Chodzi o to, że niektóre ustawienia wykluczają się wzajemnie: nie można na przykład zażądać jednocześnie wypisywania liczb w układzie dziesiętnym i ósemkowym. Tak więc flagi ios::dec, ios::hex i ios::oct tworzą pole ios::basefield i tylko jedna z nich może być ustawiona. Podobnie flagi ios::left, ios::right i ios::internal tworzą pole ios::adjustfield, a flagi ios::scientific, ios::fixed pole ios::floatfield — tu trzecią możliwością jest nieustawienie żadnej flagi, co daje format „ogólny", który jest formatem domyślnym. Flagi, które wchodzą w skład pól, powinny być ustawiane w specjalny sposób, aby zapewnić, że nigdy nie będą ustawione dwie flagi o wzajemnie wykluczającej się interpretacji. Opiszemy to poniżej.

Jak powiedzieliśmy, flaga stanu formatowania jest związana ze strumieniem, a każdy strumień jest reprezentowany przez obiekt. Zatem do zmiany flagi formatowania będą służyć metody wywoływane na rzecz obiektu reprezentującego strumień (na przykład na rzecz cin lub cout).

Metody te to:

ios::fmtflags flags( ) — zwraca aktualną flagę stanu formatowania strumienia;

ios::fmtflags flags(ios::fmtflags flg) — zwraca aktualną flagę stanu formatowania, i ustawia jej nową wartość na flg.

Powyższe metody dotyczą całej flagi stanu formatowania, a nie pojedynczych flag. Jeśli chcemy zdefiniować całą flagę stanu, po to na przykład, aby użyć jej w funkcji **flags**, możemy ją skonstruować za pomocą „ORowania" pojedynczych flag. Na przykład

```
1       // konstruujemy flage stanu
2     ios::fmtflags n = ios::hex | ios::showbase
3                                | ios::uppercase;
4       // ustawiamy nowa flage
5       // i zapamietujemy stara
6     ios::fmtflags o = cout.flags(n);
7     //
8     // ... korzystamy z nowych ustawien
9     //
10    cout.flags(o);  // przywracamy stara
```

Zauważmy, że w linii 6 ustawiliśmy nową flagę stanu, ale jednocześnie zapamiętaliśmy starą, aby móc ją później przywrócić w linii ostatniej.

Jeśli nie chcemy zmieniać ustawień, a tylko dodać jedną flagę, można postąpić na przykład tak:

```
  // pobieramy stara flage
ios::fmtflags stara = cout.flags();
  // tworzymy nowa
ios::fmtflags nowa  = stara | ios::showpos;
  // ustawiamy nowa
cout.flags(nowa);
// ... korzystamy z nowej
// ... i przywracamy stara
cout.flags(stara);
```

Konstruowanie flag w ten sposób jest nieco uciążliwe, istnieją zatem funkcje pozwalające bezpośrednio zmieniać istniejącą flagę stanu bez jej uprzedniego pobierania.

ios::fmtflags setf(ios::fmtflags flg) — zmienia flagę stanu formatowania „ORując" ją z flagą flg; zwraca flagę sprzed zmiany;

ios::fmtflags setf(ios::fmtflags flg, ios::fmtflags pole) zmienia flagę stanu formatowania „ORując" ją z flagą flg pochodzącą z pola pole; usuwa ustawienie innych flag pochodzących z tego samego pola; zwraca flagę stanu sprzed zmiany. W ten sposób należy ustawiać flagi dotyczące opcji które należą do pewnych pól; w ten sposób unikniemy sytuacji, w której ustawione są opcje wzajemnie się wykluczające. Na przykład

```
cout.setf(ios::scientific, ios::floatfield);
```

ios::fmtflags unsetf(ios::fmtflags flg) — zeruje flagę flg we fladze stanu formatowania „ANDując" ją z flagą będącą negacją bitową flagi flg.

Użycie tych metod można prześledzić na poniższym przykładzie

P132: *flags.cpp* Flagi formatowania

```
1  #include <iostream>
2  using namespace std;
3
4  typedef ios_base::fmtflags FFLAG;
5
6  int main() {
7      int    m =  49;
8      double x = 21.73;
9
10     cout << "1. m = " << m << ", x = " << x << endl;
11
12     FFLAG  newf = ios::hex | ios::showbase
13                           | ios::showpoint;
14     FFLAG  oldf = cout.flags(newf);
15     cout << "2. m = " << m << ", x = " << x << endl;
16
17     cout.setf(ios::scientific, ios::floatfield);
18     cout.unsetf(ios::showbase);
19     cout << "3. m = " << m << ", x = " << x << endl;
20
21     cout.setf(ios::fixed, ios::floatfield);
22     cout.setf(ios::showbase | ios::uppercase);
23     cout << "4. m = " << m << ", x = " << x << endl;
24
25     cout.flags(oldf);
26     cout << "5. m = " << m << ", x = " << x << endl;
27 }
```

którego wydruk to

```
1. m = 49, x = 21.73
2. m = 0x31, x = 21.7300
3. m = 31, x = 2.173000e+01
4. m = 0X31, x = 21.730000
5. m = 49, x = 21.73
```

Zwróćmy uwagę, że domyślna precyzja wynosi 6, ale oznacza to co innego w zależności od sposobu formatowania: przy formatowaniu naukowym i „fixed” oznacza liczbę cyfr po kropce (jak w liniach 3 i 4 wydruku), natomiast przy formatowaniu „ogólnym” oznacza liczbę wszystkich cyfr znaczących (przed kropką i po kropce, jak w linii 2). Dodatkowo, jeśli nie jest ustawiona flaga ios::showpos, to opuszczane są końcowe zera (linia 1 i 5). W linii 4 programu zdefiniowaliśmy za pomocą **typedef** alias dla nazwy typu **ios_base::fmtflags**. W ten sposób nie musimy powtarzać tej przydługiej nazwy w dalszej części programu.

W klasie **ios** są też zdefiniowane metody pozwalające na określenie szerokości pola, w jakim ma być wypisana liczba lub napis, oraz na określenie precyzji wyprowadzanych liczb zmiennopozycyjnych (są to też metody klasy, a więc muszą być wywoływane zawsze na rzecz konkretnego strumienia, na przykład na rzecz obiektu cout lub cin):

streamsize width( ) — zwraca aktualne ustawienie szerokości pola. Wartość zerowa oznacza „tyle ile trzeba, ale nie więcej". Typ **streamsize** to alias pewnego typu całościowego.

streamsize width(streamsize szer) — ustala szerokość pola wydruku na szer znaków; zwraca ustawienie szerokości pola sprzed zmiany. W ten sposób określana jest *minimalna* szerokość pola: jeśli wyprowadzana dana zajmuje więcej znaków, to *nie* zostanie „obcięta", a odpowiednie pole wydruku zostanie zwiększone (tak jakby szerokość pola była ustawiona na domyślną wartość 0).

streamsize precision( ) — zwraca aktualne ustawienie precyzji wyprowadzanych liczb zmiennopozycyjnych — patrz komentarz do programu ***flags.cpp*** (str. 331).

streamsize precision(streamsize prec) — ustala precyzję na prec; zwraca ustawienie precyzji sprzed zmiany.

char fill( ) — zwraca aktualny znak używany do wypełniania wyprowadzanego napisu, jeśli szerokość pola przeznaczona na ten napis jest większa niż ilość znaków w napisie (domyślnie jest to znak odstępu).

char fill(char znak) — ustala znak używany do wypełniania wyprowadzanego napisu; zwraca znak poprzednio używany do tego celu.

Należy jednak pamiętać, że

- określenie szerokości pola przez wywołanie metody **width(int szer)** ma wpływ tylko na *najbliższą* operację czytania/pisania. Po jej wykonaniu przywrócona zostanie wartość domyślna, czyli 0, co oznacza „tyle ile trzeba";
- określenie precyzji lub znaku wypełniania ma wpływ na wszystkie operacje na tym strumieniu, aż do czasu, gdy jawnie to ustawienie zmienimy.

Funkcja (metoda) **width(int szerokosc)** ma też zastosowanie do strumieni wejściowych. Jest, co prawda, ignorowana przy wczytywaniu liczb, ale za to jest bardzo przydatna przy wczytywaniu napisów: przy wczytywaniu do tablicy znakowej określa maksymalną liczbę wczytanych znaków, wliczając w to znak '\0', który zostanie dodany zawsze. Zatem, po

```
char napis[10];
cin.width(sizeof(napis));
cin >> napis;
```

wczytanych zostanie z klawiatury maksymalnie 9 znaków i dostawiony znak '\0' — w ten sposób mamy gwarancję, że nawet jeśli użytkownik podał napis liczący więcej niż 9 liter, to nie będzie „mazania po pamięci" i napis będzie zawierać prawidłowy, zakończony znakiem '\0', napis. Należy tylko pamiętać, że niewczytane znaki pozostają wtedy w buforze wejściowym i będą pierwszymi, które zostaną wczytane przy

następnej operacji wejścia na tym strumieniu (o tym, jak się ich pozbyć, powiemy za chwilę).

16.3.2 Manipulatory

Wygodniejsze od flag formatowania są **manipulatory**. Są to funkcje zdefiniowane w klasie **ios** i wywoływane poprzez podanie ich nazw jako elementów wstawianych do lub wyjmowanych ze strumienia. Ich działanie może, ale nie musi, polegać na modyfikacji flagi stanu formatowania strumienia.

Jak się przekonamy, nie muszą to być właściwie funkcje: mogą to też być obiekty funkcyjne, o których powiemy w rozdziale 24.2.2, str. 536.

Manipulatory mogą mieć argumenty, ale nie muszą.

Manipulatory bezargumentowe

Manipulatory bezargumentowe wstawia się do strumienia *nie* podając nawiasów. Mają one nazwy takie jak flagi dyskutowane już w sec 16.3.1, str. 328.

hex, oct, dec — ustawiają podstawę wyprowadzanych liczb całkowitych, podobnie jak robią to funkcje **setf(ios::hex,ios::basefield)** itd. Zmiana jest trwała: aby przywrócić poprzednie ustawienia, trzeba do strumienia wstawić odpowiedni manipulator lub na rzecz strumienia wywołać jedną z metod zmieniających flagę stanu formatownia.

left, right, internal — ustawiają sposób wyrównywania (justowania) wyprowadzanych danych, podobnie jak robią to poznane już przez nas wcześniej funkcje **setf(ios::left,ios::adjustfield)** itd.

fixed, scientific — ustawiają formatowanie wyprowadzanych liczb zmiennopozycyjnych, podobnie jak robią to funkcje **setf(ios::fixed,ios::floatfield)** itd.

showbase, noshowbase — podobne do wywołania **setf(ios::showbase)** lub **setf(ios::noshowbase)**.

showpoint, noshowpoint — podobne do wywołania **setf(ios::showpoint)** lub **setf(ios::noshowpoint)**.

flush — powoduje opróżnienie strumienia wyjściowego.

endl — — powoduje wysłanie do strumienia wyjściowego znaku końca linii i opróżnienie bufora związanego z tym strumieniem, czyli natychmiastowe wyprowadzenie znaków z bufora do miejsca przeznaczenia (na ekran, do pliku itd).

ends — — powoduje wysłanie do strumienia wyjściowego znaku końca napisu, czyli znaku '\0'.

Przykładowo, program

P133: ***manb.cpp*** Manipulatory bezargumentowe

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

```
3
4  int main() {
5      int a = 0xdf, b = 0771, c = 123;
6
7      cout << "dec (default): "
8           << dec << a << " " << b << " " << c << endl;
9
10     cout << "hex bez showbase: "
11          << hex << a << " " << b << " " << c << endl;
12
13     cout.setf(ios::showbase);
14
15     cout << "hex z showbase:   "
16          << a << " " << b << " " << c << endl;
17
18     cout << "oct z showbase:   "
19          << oct << a << " " << b << " " << c << endl;
20
21     cout.unsetf(ios::showbase);
22
23     cout << "oct bez showbase: "
24          << a << " " << b << " " << c << endl;
25 }
```

wypisuje na ekranie

```
dec (default): 223 505 123
hex bez showbase: df 1f9 7b
hex z showbase:   0xdf 0x1f9 0x7b
oct z showbase:   0337 0771 0173
oct bez showbase: 337 771 173
```

W linii 15 nie specyfikujemy podstawy, bo została ona ustawiona na hex w linii 11. Dopiero gdy chcemy zmienić podstawę na ósemkową, wstawiamy manipulator oct w linii 19.

Stosunkowo łatwo jest zdefiniować dodatkowe, własne manipulatory bezargumentowe. Manipulator taki tworzymy jako funkcję z dokładnie jednym parametrem, który musi być typu *referencja do strumienia*. Wartością zwracaną musi być ta sama referencja; na przykład

```
ostream& moj_manip(ostream& strum) {
    // ...
    return strum;
}
```

Aby użyć tak zdefiniowanego manipulatora, wstawiamy do strumienia samą jego nazwę, bez żadnego argumentu i bez nawiasów. Nasza funkcja zostanie wywołana;

argument zostanie dodany automatycznie — będzie nim referencja do strumienia, do którego manipulator został wstawiony. Ponieważ referencja do tego samego strumienia jest też wartością zwracaną, więc manipulatory można wstawiać do strumienia kaskadowo, tak jak inne elementy.

Rozpatrzmy przykład:

P134: *wman.cpp* Własne manipulatory

```
1  #include <iostream>
2  using namespace std;
3
4  ostream& naukowy(ostream&);
5  ostream& normalny(ostream&);
6  ostream& przec(ostream&);
7
8  int main() {
9      double x = 123.456;
10     cout << naukowy  << x << przec
11          << normalny << x << endl;
12 }
13
14 ostream& naukowy(ostream& str) {
15     str.setf(ios::showpos | ios::showpoint);
16     str.setf(ios::scientific, ios::floatfield);
17     str.precision(12);
18     return str;
19 }
20
21 ostream& normalny(ostream& str) {
22     str.flags((ios::fmtflags)0);
23     return str;
24 }
25
26 ostream& przec(ostream& str) {
27     return str << ", ";
28 }
```

który drukuje

```
+1.234560000000e+02, 123.456
```

Pierwszy manipulator, naukowy, zdefiniowany w liniach 14-19, znanymi nam już metodami zmienia flagę stanu formatowania strumienia. Drugi, normalny, przywraca fladze formatowania wartość domyślną, którą jest układ bitów złożony z samych zer (rzutowanie tego zera na typ **ios::fmtflags** w linii 22 jest konieczne). Wreszcie trzeci manipulator, przec, nie zmienia żadnych flag, tylko po prostu wpisuje do strumienia znaki przecinka i odstępu.

We wszystkich przypadkach parametr funkcji definiującej manipulator ma typ **ostream&**. Daje nam to dużą elastyczność: nigdzie tu nie jest powiedziane, że tym strumieniem będzie cout. Może to być dowolny strumień wyjściowy reprezentowany obiektem klasy **ostream** lub obiektem klasy dziedziczącej z **ostream**, na przykład strumień wyjściowy związany z plikiem (czyli obiekt klasy **ofstream**).

Manipulatory z argumentami

Istnieją też manipulatory argumentowe. Stosuje się je analogicznie jak manipulatory bezargumentowe, ale wymagają one argumentów, które, jak dla zwykłych funkcji, podaje się w nawiasach. Implementowane są jako obiekty funkcyjne (patrz rozdz. 24.2.2, str. 536).

Aby używać predefiniowanych manipulatorów argumentowych, należy dołączyć plik nagłówkowy ***iomanip***.

Predefiniowane manipulatory argumentowe pozwalają wykonać zadania realizowane przez metody już wcześniej opisywane, ale bez wywoływania tych metod bezpośrednio, a poprzez wstawianie manipulatorów do strumienia. Różnica jest taka, że zwracają, jak wszystkie manipulatory, referencję do strumienia do którego zostały wstawione:

setw(int szer) — ustawia szerokość pola dla najbliższej operacji na strumieniu. Działa jak opisana wcześniej *metoda* **width** (tyle, że nie zwraca liczby, ale referencję do strumienia, jak wszystkie manipulatory). Przypomnijmy, że w ten sposób określana jest *minimalna* szerokość pola: jeśli dana zajmuje więcej znaków, to odpowiednie pole zostanie zwiększone. Domyślną wartością szerokości pola jest zero, czyli każda wypisywana dana zajmie tyle znaków, ile jest potrzebne, ale nie więcej;

setfill(int znak) — ustawia znak, którym będą wypełniane puste miejsca jeśli szerokość pola wydruku jest większa niż liczba wyprowadzanych znaków (domyślnie jest to znak odstępu). Odpowiada metodzie **fill**.

setprecision(int prec) — ustawia precyzję jak metoda **precision**.

setiosflags(ios::fmtflags flag) — modyfikuje flagę formatowania tak jak jednoargumentowa metoda **setf**, czyli „ORuje” flag z flagą stanu formatowania.

resetiosflags(ios::fmtflags flag) — odpowiednik metody **unsetf**.

setbase(int baza) — zmienia podstawę używaną przy wyprowadzaniu liczb całkowitych.

Różne metody formatowania wyników zademonstrowane są w poniższym programie:

P135: ***primatr.cpp*** Formatowanie

```
1 #include <iostream>
2 #include <iomanip>
3 using namespace std;
4
```

```
5  void printMatrix(ostream&,double**,int,int,const char*);
6
7  int main() {
8      const int DIM = 5;
9      double t[][DIM] = { {    1,  3,    5,       23, 16.42},
10                         {4.567,  4,    6,  234.345,    98},
11                         {  585, 34,    1,       67,  31.2},
12                         {    1,  0,    1, 2345.967, 123.2},
13                         {  1.2, 10, 34.1,    5.900,   0.2}
14                       };
15     double* tab[DIM];
16     for (int i = 0; i < 5; i++) tab[i] = t[i];
17
18     char  name[5];
19     int  prec = 3;
20
21     cout << "Name: ";
22     cin  >> setw(5) >> name;
23
24     printMatrix(cout, tab, DIM, prec, name);
25 }
26
27 void printMatrix(ostream& strm, double* tab[], int size,
28                                 int prec, const char* name) {
29     ios::fmtflags old =
30         strm.setf(ios::fixed, ios::floatfield);
31
32     strm << setiosflags(strm.flags() | ios::showpoint)
33          << setprecision(prec) << "\nMatrix: " << name
34          << "\n\n";
35
36     for (int i = 0; i < size; i++) {
37         strm << "ROW " << setfill('0') << setw(2)
38              << (i+1)  <<  ":" << setfill(' ');
39         for (int j = 0; j < size; j++)
40             strm << setw(9) << tab[i][j];
41         strm << endl;
42     }
43     strm << endl << setiosflags(old);
44 }
```

W programie zdefiniowana jest funkcja wyprowadzająca macierz w czytelnej formie (macierz jest przekazywana jako tablica wskaźników do wierszy). Zauważmy, że funkcja ta pobiera poprzez argument strumień, na który mają być wyprowadzane wyniki. W ten sposób ta sama funkcja będzie mogła być użyta do zapisania macierzy na przykład do pliku.

Zauważmy również sposób pobierania od użytkownika pewnej nazwy w linii 22. Ponieważ tablica przewidziana na nazwę ma tylko pięć znaków, używamy tu funkcji **setw**: w ten sposób, nawet jeśli użytkownik poda za długą nazwę, zostanie ona obcięta (do czterech znaków + NUL), ale nie nastąpi przepełnienie tablicy:

```
Name: zanzibar

Matrix: zanz

ROW 01:    1.000    3.000    5.000   23.000   16.420
ROW 02:    4.567    4.000    6.000  234.345   98.000
ROW 03:  585.000   34.000    1.000   67.000   31.200
ROW 04:    1.000    0.000    1.000 2345.967  123.200
ROW 05:    1.200   10.000   34.100    5.900    0.200
```

Inną ważną kwestią jest pisanie funkcji tak, aby nie powodowały skutków ubocznych. Dlatego dbamy o to, aby po zakończeniu wykonywania zadania funkcja przywróciła flagę formatowania używanego strumienia do stanu sprzed wywołania (linie 29 i 43).

Podobnie jak dla manipulatorów bezargumentowych, programista ma możliwość zdefiniowania dodatkowych manipulatorów argumentowych. Jak to zrobić, powiemy przy okazji omawiania obiektów funkcyjnych (rozdz. 24.2.2, str. 536).

16.4 Zapis i odczyt nieformatowany

Do tej pory mówiliśmy o zapisie/odczycie formatowanym — informacja czytana lub pisana jest w jakiś sposób interpretowana: opuszczane są białe znaki, dokonuje się przekształceń liczb do napisów w różnych formatach itd. Istnieją też operacje we/wy nieformatowane, które czytają lub piszą „surowe” bajty — bez ich żadnej interpretacji.

16.4.1 Odczyt nieformatowany

Na rzecz obiektu strumieniowego mogą być też wywołane metody powodujące wczytanie danych ze strumienia bez ich interpretacji i formatowania. Należą do nich następujące metody:

istream& get(char& c) — czyta jeden bajt; zwraca referencję do strumienia, na rzecz którego została wywołana, więc może być używana kaskadowo. Argument typu **char** jest przesyłany przez referencję; po powrocie jego wartością będzie wczytany znak. Może to być *dowolny* znak, również znak kontrolny, biały lub zerowy (czyli '\0'). Jeśli czytanie nie powiodło się, bo napotkany został koniec pliku, znak c będzie równy EOF, czyli znak, który w danym systemie operacyjnym oznacza koniec danych (Ctrl-Z w Windows, Ctrl-D pod Uniksem/Linuksem). Sam strumień będzie wtedy w stanie błędu. Poznać to można przez wymuszenie konwersji zmiennej strumieniowej do typu **void*** — jeśli strumień jest „dobry” to otrzymamy wartość niezerową, jeśli zły, to otrzymamay **nullptr**. Taka konwersja zachodzi automatycznie

w kontekście, w którym wymagana jest wartość logiczna, a zatem można jej użyć bezpośrednio do części testującej instrukcji **if**, **for**, **while** itd. Na przykład, jeśli strin jest strumieniem wejściowym związanym z plikiem, to można zawartość tego pliku przekopiować na standardowe wyjście prostą pętlą

```
char c;
while (strin.get(c)) cout << c;
```

Kaskadowość funkcji **get** można wykorzystać w konstrukcjach typu

```
char a, b, c;
strm.get(a).get(b).get(c);
```

przy czym w ten sposób możemy wczytać dowolne znaki; bez żadnego opuszczania białych znaków, interpretacji znaku końca linii itp.

int get() — zwraca wczytany znak w formie liczby typu **int**; w razie napotkania końca pliku zwrócone zostanie EOF. Ta forma funkcji **get** *nie* zwraca referencji do strumienia, więc nie może być używana kaskadowo.

istream& get(char* buf, streamsize length, char termin = '\n') — czyta do łańcucha (napisu) buf — czyli do pamięci poczynając od adresu będącego wartością wskaźnika buf — maksymalnie length-1 bajtów. Znak NUL (zerowy) jest automatycznie dodawany na koniec łańcucha buf. Bajt numer length ze strumienia nie jest już wczytywany. Wczytywanie kończy się również, jeśli napotkany zostanie znak termin (domyślnie, jak widać z nagłówka, jest to znak końca linii '\n'). Znak NUL też zostanie wtedy dopisany do łańcucha buf, ale sam znak termin *pozostaje* w strumieniu jako „niewyczytany" i nie jest umieszczany w wyjściowym buforze buf. Zatem przed wczytaniem kolejnych danych należy go zwykle „wyczytać" — można to zrobić za pomocą funkcji **ignore** opisanej dalej. Oczywiście przekazany do funkcji bufor musi istnieć, to znaczy pamięć na ten bufor musiała zostać zaalokowana w funkcji wołającej. Drugi parametr funkcji jest typu **streamsize**, co jest aliasem pewnego typu całkowitego. Funkcja zwraca referencję do strumienia, na rzecz którego została wywołana (*this).

istream& getline(char* buf, streamsize length, char termin = '\n') — działa dokładnie tak jak poprzednia funkcja, ale znak termin *jest* wyjmowany ze strumienia, choć *nie* jest umieszczany w buforze buf. Jest to zatem metoda znacznie praktyczniejsza od poprzedniej przy wczytywania kolejnych linii tekstu.

istream& read(char* buf, streamsize length) — wczytuje length bajtów do bufora buf. Czytanie może się zakończyć przedwcześnie tylko, jeśli napotkany został koniec danych. O tym, ile rzeczywiście bajtów zostało wczytanych, można się przekonać wywołując bezpośrednio po czytaniu funkcję **gcount** (patrz dalej). Funkcja czyta surowe bajty i nie dostawia znaku końca napisu ani żadnego innego. Stosuje się ją zwykle do czytania danych nietekstowych. Funkcja zwraca referencję do strumienia na rzecz którego została wywołana (*this).

istream& ignore(streamsize length = 1, int termin = EOF) — wczytuje length znaków (domyślnie jeden znak), nigdzie ich nie wpisuje. Jeśli napotkany został

znak termin (domyślnie koniec pliku), to jest on usuwany ze strumienia i czytanie kończy się. Funkcja zwraca referencję do strumienia, na rzecz którego została wywołana.

int gcount( ) — zwraca liczbę wczytanych ze strumienia bajtów podczas wykonywania ostatniej funkcji nieformatowanego wejścia.

int peek( ) — zwraca jeden znak ze strumienia w postaci zmiennej typu **int** — może to być EOF jeśli napotkany został koniec pliku. Znak pozostaje w strumieniu i będzie pierwszym znakiem wczytanym przy następnej operacji czytania.

istream& putback(char znak) — wstawia znak znak do strumienia; będzie on pierwszym znakiem wczytanym przez następną operację czytania. Zwraca odnośnik do strumienia, na rzecz którego została wywołana. Nie do każdego strumienia i nie zawsze można wstawić znak.

istream& unget() — wstawia ostatnio przeczytany znak z powrotem do strumienia; będzie on pierwszym znakiem wczytanym przez następną operację czytania. Zwraca referencję do strumienia, na rzecz którego została wywołana.

Kilka z tych funkcji użytych zostało w poniższym programie, który czyta kolejne linie ze standardowego wejścia i wyświetla je bez zmian, ale pomijając komentarze, czyli fragmenty od dwuznaku '//' do końca linii:

P136: *czytnf.cpp* Czytanie nieformatowane

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      cout << "Wpisuj linie tekstu. Zakoncz znakiem konca "
6              "pliku\n(Ctrl-Z w Windows, Ctrl-D w Linuksie) "
7              "Komentarze\nod \'//\' do konca linii beda "
8              "pomijane.\n";
9      char c;
10     while ( cin.get(c) ) {
11         if ( c == '/' )
12             if ( cin.peek() == '/') {
13                 cin.ignore(1024,'\n');
14                 cout << endl;
15                 continue;
16             }
17         cout << c;
18     }
19 }
```

Po wczytaniu każdego znaku prgram sprawdza, czy jest to znak '/' (linia 11). Jeśli tak, to za pomocą **peek** „podgądany" jest następny znak. Jeśli to też jest '/', to linia jest „wyczytywana" do końca (za pomocą funkcji **ignore**; linia 13), wypisywany jest znak nowej linii, po czym funkcja kontunuuje czytanie kolejnych znaków. Program kończy się, gdy wczytanym znakiem będzie znak końca pliku EOF, czyli gdy

użytkownik wpisze Ctrl-Z (Windows) lub Ctrl-D (Linux). Strumień sprawdzany w linii 10 będzie wtedy przekonwertowany do **nullptr**, co zakończy pętlę. Przykładowe uruchomienie programy dało

```
Wpisuj linie tekstu. Zakoncz znakiem konca pliku
(Ctrl-Z w Windows, Ctrl-D w Linuksie) Komentarze
od '//' do konca linii beda pomijane.
int main(void) {
int main(void) {
    const int DIM = 15; // wymiar tablicy
    const int DIM = 15;
    int tab[DIM];       // tablica znakow
    int tab[DIM];
    double x=1, y=2, z=x/y;  // trzy liczby
    double x=1, y=2, z=x/y;
    // ...

}
}
^D
```

Zwróćmy też uwagę na linie 5-8 programu. Zilustrowana tu jest bardzo przydatna cecha kompilatora C/C++: literały napisowe, a więc napisy ograniczone znakami cudzysłowu, są łączone (konkatenowane) w jeden napis, jeśli oddzielone są od siebie tylko białymi znakami (którym jest, w szczególności, znak końca linii).

16.4.2 Zapis nieformatowany

Bardzo ważna jest umiejętność nieformatowanego zapisu. Oczywiście przydaje się ona głównie przy zapisie do pliku lub gniazda sieciowego, nie na ekran komputera. Pozwala zapisywać dane z pełną dokładnością w formie oszczędzającej miejsce na dysku (zauważmy, że zapis liczby -1.234567E+19 zajmuje 13 znaków, choć ta sama liczba typu **float** z pełną dokładnością zajmuje w pamięci tylko 4 bajty, a więc tylko 4 bajty muszą być wyprowadzone do binarnego pliku).

Pliki takie nie mogą być, co prawda, oglądane czy modyfikowane w zwykłych edytorach tekstu, ale mogą być czytane przez ten sam lub inne programy, również za pomocą operacji nieformatowanych. Nieformatowane operacje we/wy są też konieczne do pracy z plikami nietekstowymi, na przykład graficznymi, dźwiękowymi itd.

Opisane poniżej dwie metody są metodami z klasy **ostream**: mogą zatem być wywoływane na rzecz dowolnego strumienia wyjściowego, nie tylko strumienia cout.

ostream& put(char c) — wstawia znak (bajt) c do strumienia. Zwraca referencję do strumienia, na rzecz którego metoda została wywołana.

ostream& write(const char* buf, streamsize length) — zapisuje length znaków (bajtów) z bufora buf do strumienia. Zwraca referencję do strumienia, na rzecz którego została wywołana.

Przykłady na zastosowanie tych operacji podamy po zapoznaniu się z obsługą strumieni związanych z plikami.

16.5 Pliki

Klasy obsługujące operacje we/wy na plikach to:

- **ofstream** — pochodna klasy **ostream**;
- **ifstream** — pochodna klasy **istream**;
- **fstream** — pochodna klasy **iostream**, łącząca funkcjonalność klas wejściowych i wyjściowych.

Z dziedziczenia wynika, że można stosować te same metody (operatory '<<' i '>>', manipulatory, funkcje **get**, **put**, **getline** itd.) w odniesieniu do obiektów tych klas. Należy tylko utworzyć odpowiedni obiekt, odpowiednik predefiniowanych cin czy cout.

Aby posłużyć się strumieniem plikowym, trzeba stworzyć do niego „uchwyt", związać go z konkretnym plikiem, otworzyć strumień, a po zakończeniu na nim operacji we/wy zamknąć.

Tworzenie uchwytu do pliku odbywa się poprzez utworzenie (zdefiniowanie) obiektu strumienia i wywołanie funkcji wiążącej strumień z konkretnym plikiem i otwierającej go. Na przykład następujący fragment

```
1    #include <fstream>
2    // ...
3    ofstream plik;
4    plik.open("plik.txt");
5    plik << "To bedzie w pliku \"plik.txt\"" << endl;
6    plik.close();
```

tworzy w linii 3 obiekt reprezentujący strumień związany z plikiem do zapisu (dlatego **ofstream** — *output file stream*). W następnej linii strumień ten jest wiązany z konkretnym plikiem i otwierany. W linii 5 widzimy, że nazwy plik możemy teraz używać tak samo, jak do tej pory używaliśmy cout; różnica jest tylko w ujściu strumienia — wtedy był to ekran komputera, teraz będzie plik. Po zakończeniu pracy z plikiem strumień należy zamknąć (linia 6). Zauważmy, że strumienie predefiniowane nie musiały być jawnie ani otwierane, ani zamykane.

Zamiast wywoływania funkcji **open** można posłać nazwę pliku bezpośrednio do konstruktora obiektu:

```
#include <fstream>
// ...
ofstream plik("plik.txt");
plik << "To bedzie w pliku \"plik.txt\"" << endl;
plik.close();
```

Zarówno do funkcji **open**, jak i do konstruktora można posłać dodatkowy, prócz nazwy pliku, argument. Określa on **tryb otwarcia** (ang. *opening mode*)) pliku. Dla plików do czytania, dla których tworzymy obiekt klasy **ifstream** (*input file stream*), domyślny tryb to ios::in (jest to statyczna stała odziedziczona z klasy **ios_base**). Dla pliku z uchwytem klasy **ofstream** domyślnie przyjmowane jest **ios::out** — plik do zapisywania. Stałe określające tryb pliku można „ORować" tak samo, jak to robiliśmy dla flag formatowania. Na przykład

```
fstream strum("plik.txt", ios::in | ios::out);
```

tworzy obiekt strumienia (klasy **fstream**) w trybie *do zapisu i odczytu*.

Dostępne stałe, z których poprzez alternatywę bitową można budować wartości określające tryb otwarcia, to:

- ios::in — zezwala na odczyt ze strumienia;
- ios::out — zezwala na zapis do strumienia;
- ios::trunc — (*truncate*) po utworzeniu usuń dotychczasową zawartoś";
- ios::ate — po otwarciu ustaw na końcu pliku (*at end*);
- ios::app — (*append*) przechodź na koniec strumienia dla każdej operacji wstawiania (pisania);
- ios::binary — nie interpretuj w żaden sposób znaków końca linii i powrotu karetki (w Linuksie ta opcja jest zbędna, ale przydaje się w Windows, gdzie koniec linii oznaczany jest za pomocą *dwóch* znaków);

Ze strumieniami są związane **lokalizatory** zawierające numer, licząc od zera, bajtu w strumieniu (pliku), na który nastąpi następny zapis/odczyt. Ich typem jest **streampos** (zwykle tożsamy z **long**). Po otwarciu pliku domyślnie lokalizatory ustawiane są na samym jego początku, a więc na bajcie numer zero, chyba że plik został otwarty w trybie ios::ate lub ios::app, gdy plik pozycjonowany jest na bajcie „pierwszym za ostatnim" (numeryczna wartość lokalizatora jest wtedy równa długości pliku w bajtach). Po każdej operacji czytania/pisania lokalizator przesuwany jest tak, aby wskazywał na pierwszy bajt jeszcze nie wczytany/zapisany. Nawet jeśli plik jest otwarty zarówno do zapisu jak i odczytu, to pamiętana jest tylko jedna pozycja w strumieniu: ta sama do operacji zapisu jak i odczytu.

Do „ręcznego" manipulowania lokalizatorami służą metody:

streampos tellg( ) — zwraca aktualną pozycję lokalizatora do odczytu (litera 'g' na końcu nazwy pochodzi od *get*). Tryb otwarcia pliku musi zawierać ios::in.

streampos tellp( ) — zwraca aktualną pozycję lokalizatora do zapisu (litera 'p' na końcu nazwy pochodzi od *put*). Tryb otwarcia pliku musi zawierać ios::out.

ostream& seekg(streampos poz) — przesuwa lokalizator do odczytu na pozycję poz. Zwraca referencję do strumienia, na rzecz którego została wywołana. Tryb otwarcia pliku musi zawierać ios::in. Próba sięgnięcia przed początek bądź za koniec pliku

powoduje jego przejście do stanu bad (patrz dalej), co można sprawdzić za pomocą warunku '`if (strum) ...`'.

ostream& seekp(streampos poz) — przesuwa lokalizator do zapisu; poza tym analogiczna do metody **seekg(streampos)**.

ostream& seekg(streamoff offset, ios::seek_dir poz) — przesuwa lokalizator do odczytu na pozycję offset bajtów licząc od pozycji poz. Argument offset może być ujemny. Typy **streamoff** i **ios::seek_dir** są aliasami pewnych typów całościowych. Argument poz musi być równy jednej ze stałych statycznych z klasy **ios**:
ios::beg — licz od początku pliku;
ios::cur — licz od aktualnej pozycji w pliku;
ios::end — licz od końca pliku;
Zwraca referencję do strumienia, na rzecz którego została wywołana. Próba sięgnięcia przed początek bądź za koniec pliku powoduje przejście strumienia do stanu bad.

ostream& seekp(streamoff offset, ios::seek_dir poz) — przesuwa lokalizator do zapisu; poza tym analogiczna do metody **seekg(streamoff offset, ios::seek_dir poz)**.

Jako przykład rozpatrzmy program

P137: ***plikrw.cpp*** Zapis i odczyt z pliku

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    int tab[] = { 97, 105, 115, 255, 111 },k;

    int size = sizeof(tab)/sizeof(tab[0]);

    cout << "Tablica o  wymiarze: " << size << endl;
    for (int i = 0; i < size; ++i)
        cout << tab[i] << " ";
    cout << endl;

    ofstream file_out("file.dat",ios::out|ios::binary);
    if (! file_out ) {
        cout << "Nie mozna otworzyc file_out" << endl;
        return -1;
    }

    file_out.write((char*)tab, sizeof(tab));
    file_out.close();

    fstream file("file.dat",ios::in|ios::out|ios::binary);
    if (! file ) {
```

```
26          cout << "Nie mozna otworzyc file" << endl;
27          return -1;
28      }
29
30      file.seekg(0,ios::end);
31      streamsize len = file.tellg();
32      cout << "Plik ma dlugosc " << len << " bajtow\n";
33      file.seekg(0);
34
35      cout << "Kolejne bajty zawieraja:" << endl;
36      while ( (k = file.get()) != EOF )
37          cout << k << " ";
38      cout << endl;
39
40      file.clear(); // <-- KONIECZNE !!!
41
42      file.seekg(4);
43      file.read((char*)&k,4);
44      cout << "Integer od pozycji 4: " << k << endl;
45
46      file.seekp(12);
47      file.write((char*)&k,4);
48
49      file.seekg(0);
50      cout << "Kolejne bajty pliku teraz zawieraja:" << endl;
51      while ( (k = file.get()) != EOF )
52          cout << k << " ";
53      cout << endl;
54
55      file.close();
56  }
```

W linii 15 tworzymy i otwieramy strumień zapisujący związany z plikiem ***file.dat***. Aby uniknąć ewentualnych kłopotów w systemie Windows, plik jest otwierany w trybie binarnym.

Następnie, w linii 21, zapisujemy do tego pliku całą zawartość pięcioelementowej tablicy liczb typu **int**. Zapis jest binarny, więc zapisanych powinno zostać dokładnie 20 bajtów. Po zapisaniu tablicy plik zamykamy, a następnie otwieramy znowu, tym razem do czytania i pisania (linia 24). Pozycjonujemy strumień na końcu pliku i odczytujemy pozycję: powinna ona wskazywać na bajt „pierwszy za ostatnim", a ostatni ma numer 19. Zatem powinna to być pozycja numer 20, co jest równe ilości bajtów w pliku (linie 30-32).

Przewijamy plik do początku (linia 33) i czytamy zawartość pliku bajt po bajcie, wypisując wynik na ekranie (linie 36-38).

Po tej operacji stan strumienia jest bad, ponieważ w pętli próbowaliśmy odczytać bajt po osiągnięciu końca pliku. Każda następna operacja wejścia/wyjścia na tym

strumieniu byłaby zignorowana (choć nie zostałby zgłoszony żaden błąd — musimy to zawsze sami sprawdzać!). Dlatego przed dalszym użyciem strumienia musimy go „naprawić". Robi się to za pomocą metody **clear** (linia 40), o której jeszcze powiemy za chwilę.

Pozycjonujemy teraz plik na bajcie numer 4 (czyli piątym, a pierwszym należącym do drugiej liczby z zapisanej tablicy). Poczynając od tej pozycji czytamy 4 bajty i kopiujemy je do zmiennej k typu **int** (linie 42-43). Po tej operacji wartość k powinna być zatem równa 105. Teraz dokonujemy operacji odwrotnej: pozycjonujemy lokalizator do pisania na bajcie numer 12 (początek czwartej liczby) i zapisujemy tam 4 bajty zmiennej k (linie 46-47). Tak więc czwarta liczba zapisana w pliku powinna zmienić się na 105. Tak jest rzeczywiście, o czym przekonuje nas wydruk:

```
Tablica o  wymiarze: 5
97 105 115 255 111
Plik ma dlugosc 20 bajtow
Kolejne bajty zawieraja:
97 0 0 0 105 0 0 0 115 0 0 0 255 0 0 0 111 0 0 0
Integer od pozycji 4: 105
Kolejne bajty pliku teraz zawieraja:
97 0 0 0 105 0 0 0 115 0 0 0 105 0 0 0 111 0 0 0
```

Wydruk zależy tu od architektury komputera; w tym przypadku jest to little-endian. Na komputerze big-endian pierwsza liczba wydrukowana zostałaby jako '0 0 0 97', a nie tak jak w przykładzie '97 0 0 0'.

16.6 Obsługa błędów strumieni

Prócz flagi stanu formatowania, każdy strumień posiada **słowo stanu** zawierające informacje o stanie i o ewentualnych błędach związanych z operacjami we/wy na tym strumieniu. Tak jak flaga stanu formatowania, słowo stanu strumienia ma postać całkowitoliczbowej zmiennej (typu **iostate**).

Programista ma możliwość badania aktualnego stanu strumienia za pomocą metod wywoływanych na rzecz obiektów-strumieni.

Można też badać stan strumienia bezpośrednio w warunkach logicznych instrukcji **if**, **for**, **while** itd. W takich przypadkach wartość strumienia zostanie przekonwertowana automatycznie dzięki przeciążeniu w klasie **ios** operatora '!' oraz operatora konwersji do typu **void***.

W wyrażeniach typu

```
if ( strm ) ...
```

wartość strm zostanie przekonwertowana do wartości wskaźnika pustego **nullptr** (odpowiadającego **false**), jeśli stan strumienia jest „zły", to znaczy, jeśli metoda **fail** wywołana na rzecz tego strumienia dałaby **true** (patrz dalej). W przeciwnym przypadku, czyli gdy stan strumienia jest „dobry", zwrócona zostanie wartość niezerowa, odpowiadająca **true**. Podobnie w wyrażeniu

```
if ( !strm ) ...
```

operator '!' zastosowany do strumienia zwróci wartość logiczną **true** wtedy i tylko wtedy, gdy metoda **fail** zwróciłaby **true**.

Słowo stanu zapewnia nam jednak więcej informacji. Tak jak flagę stanu formatowania można składać (lub rozkładać na czynniki pierwsze...) za pomocą nazwanych stałych, tak i słowo stanu strumienia można przez „ORowanie" składać z predefiniowanych stałych typu **iostate**.

ios::badbit — Strumień jest zniszczony; nie wiadomo, czy ostatnia operacja na nim powiodła się. Następna na pewno nie powiedzie się.

ios::eofbit — Wykonano próbę dostępu do danych poza końcem strumienia (pliku).

ios::failbit — Następna operacja nie powiedzie się, ale strumień nie jest zniszczony, w szczególności nie zgubiono żadnych znaków.

ios::goodbit — Strumień jest „zdrowy". Wartością goodbit jest 0.

Stan strumienia można badać za pomocą funkcji składowych (metod) klasy **ios**

- **bool bad( )**
- **bool eof( )**
- **bool fail( )**
- **bool good( )**

zwracających w postaci wartości logicznej ustawienie bitów ios::badbit, ios::eofbit, ios::failbit i ios::goodbit w słowie stanu strumienia.

Są też metody pozwalające manipulować słowem stanu strumienia „ręcznie" — nie tylko odczytywać go, ale i modyfikować.

iostate rdstate() — Zwraca słowo stanu w postaci zmiennej typu **iostate**.

void clear(iostate stan = ios::goodbit) — Zeruje słowo stanu, co odpowiada ustawieniu stanu na good. Następnie ustawia znacznik stan, który musi być jedną ze stałych ios::badbit, ios::failbit itd. Domyślną wartością jest ios::goodbit, a zatem wywołanie bez argumentu „naprawia" strumień — patrz przykład w programie ***plikrw.cpp*** (str. 344).

void setstate(iostate stan) — Dodaje (przez „ORowanie") znacznik stan do słowa stanu. Argument musi być jedną ze stałych ios::badbit, ios::failbit itd. Równoważne wywołaniu '`clear( rdstate() | ios::stan )`'.

Generalnie, stan strumienia należy w poważnych programach sprawdzać po każdej operacji. Jeśli stan strumienia jest „zły", a więc funkcja **fail**, **bad** lub **eof** zwraca **true**, to każda następna operacja we/wy na tym strumieniu jest *ignorowana*, natomiast nie jest zgłaszany żaden błąd! Rozpatrzmy przykład:

P138: ***validan.cpp*** Sprawdzanie poprawności wczytywanych danych

```
1  #include <fstream>
2  #include <iostream>
3  using namespace std;
4
5  int main() {
6      const int DIM = 80;
7      char      name[DIM];
8      ifstream  inplik;
9      double    x;
10
11     do {
12         cout << "Plik wejsciowy: ";
13         cin.getline(name, DIM);
14
15         inplik.clear();
16         inplik.open(name);
17     } while (!inplik);
18
19     cout << "Plik = " << name << endl;
20     inplik.close();
21
22     do {
23         if (!cin) {
24             // wazna kolejnosc!
25             cin.clear();
26             cin.ignore(1024,'\n');
27         };
28         cout << "Podaj liczbe: ";
29         cin >> x;
30     } while (!cin);
31
32     cout << "Liczba = " << x << endl;
33 }
```

W linii 13 wczytujemy nazwę pliku. Następnie, w linii 16, próbujemy go otworzyć do czytania. Jeśli plik o takiej nazwie nie istnieje, strumień inplik będzie w stanie bad, zatem warunek pętli **do...while** będzie spełniony; program przejdzie do następnego jej obrotu i jeszcze raz zapyta o nazwę. Stan strumienia w dalszym ciągu będzie bad, więc w linii 15 naprawiamy go przed podjęciem kolejnej próby. Ważne jest to, że pętlę opuścimy tylko wtedy, gdy plik został prawidłowo otwarty i strumień inplik jest w stanie good.

Bardziej subtelna jest pętla w liniach 22-30, której zadaniem jest wczytanie liczby. W linii 29 usiłujemy wczytać liczbę na zmienną x. Jeśli się to nie powiedzie, bo na przykład użytkownik wpisał zamiast liczby litery, to stan strumienia cin będzie bad.

Zatoczymy zatem pętlę i podejmiemy kolejną próbę. Ale ponieważ stan strumienia jest zły, operacje wejścia na tym strumieniu będą ignorowane! Zatem musimy naprawić stan, co czynimy w linii 25. To jeszcze nie wszystko. Ponieważ bufor strumienia nie został „wyczytany", gdybyśmy ograniczyli się do naprawy strumienia, to następna operacja będzie czytać nie wprowadzoną, być może tym razem prawidłowo, liczbę, ale „śmieci" pozostałe w buforze z poprzedniej, nieudanej próby. I tak w nieskończoność będziemy czytać te same śmieci! Zatem musimy je usunąć przez wywołanie **ignore** w linii 26. Teraz program działa prawidłowo (zakładamy, że istnieje w aktualnym katalogu plik ***val.dat***):

```
Plik wejsciowy: val.txt
Plik wejsciowy: val
Plik wejsciowy: val.dat
Plik = val.dat
Podaj liczbe: s12
Podaj liczbe: @
Podaj liczbe: 12
Liczba = 12
```

Zauważmy też, że ważna jest kolejność naprawiania strumienia i wywoływania **ignore** z linii 25 i 26. Gdybyśmy wywołali **ignore** *przed* naprawieniem strumienia, to wywołanie to zostałoby *zignorowane*, bo przecież jest to operacja wejścia/wyjścia, a stan strumienia jest bad. A więc „śmieci" pozostałyby w strumieniu i znów wpadlibyśmy w nieskończoną pętlę, nie mogąc pozbyć się błędnych danych. Po odwróceniu kolejności linii 25 i 26 otrzymalibyśmy zatem:

```
Plik wejsciowy: val.dat
Plik = val.dat
Podaj liczbe: s12
Podaj liczbe: Podaj liczbe: Podaj liczbe: Podaj li
czbe: Podaj liczbe: Podaj liczbe:
```

gdzie wykonanie musieliśmy przerwać wciskając Ctrl-C.

Niestety, prawidłowa i pełna obsługa wszystkich możliwych błędów operacji wejścia i wyjścia to zadanie bardzo trudne i pracochłonne. Kod z tym związany może stanowić znaczną część (czasem większość!) całego programu.

16.7 Pliki wewnętrzne

Rolę plików mogą też pełnić napisy, czyli tablice znaków w stylu C i napisy w stylu C++, a więc obiekty klasy **string**.

16.7.1 Tablice znaków jako pliki wewnętrzne

Napis w stylu C jest ciągłym obszarem pamięci, do którego odnosimy się jak do tablicy znaków. Obszar ten może być traktowany jako plik, do którego dane mogą być zapisywane lub z którego mogą być odczytywane.

Aby korzystać z tego rodzaju „plików”, należy dołączyć plik nagłówkowy ***strstream***, który daje nam dostęp do klasy strumieniowej wejściowej **istrstream** i wyjściowej **ostrstream**.

Ten nagłówek został usunięty ze standardu, ale jest implementowany przez wszystkie kompilatory C++ dla zachowania wstecznej zgodności. Przy kompilacji mogą pojawić się ostrzeżenia!

Obiekt strumieniowy klasy wyjściowej **ostrstream** tworzymy na dwa sposoby:

- `ostrstream strum;` — bez argumentu konstruktora. Możemy wtedy wpisywać do niego dowolnie wiele znaków, stosując formatowanie, manipulatory itd. *Po zakończeniu* wpisywania wywołujemy na rzecz tego obiektu bezargumentową funkcję **str**, która zwraca wskaźnik typu **char*** do początku wynikowej tablicy znaków. Jeśli chcemy, aby był to prawidłowy C-napis (co nie zawsze jest konieczne), musimy sami zadbać o to, aby ostatnim wpisanym znakiem był znak '\0'; na przykład przez instrukcję `strum << ends` jako ostatnią instrukcję pisania. Zwrócona tablica znaków zaalokowana jest na stercie za pomocą funkcji **malloc**, zatem jeśli nie jest nam już potrzebna, usunąć ją trzeba za pomocą wywołania funkcji **free** (patrz rozdz. 12.5 na stronie 228). Nie ma tu metody **close**.

- `ostrstream strum(char* tab, size_t len);` — dane wpisywane będą do tablicy znaków tab; wielkość dostępnej pamięci określa drugi argument len (typu **size_t**, który jest pewnym typem całościowym). Po otwarciu strumienia możemy do niego pisać używając normalnych operacji, jak operator <<, funkcje **write**, **put** itd. Jeśli podczas zapisywania do strumienia liczba zapisanych bajtów miałaby przekroczyć len, to jako ostatni bajt (o indeksie len-1) powinien być wpisany znak NUL (czyli '\0'), stan strumienia powinien być bad, co oznacza, że dalsze operacje zapisywania będą ignorowane. Zabezpiecza nas to przed przypadkowym „mazaniem po pamięci”.

Przykład:

P139: ***intfilo.cpp*** Zapis do plików wewnętrznych

```
1  #include <iostream>
2  #include <strstream>
3  #include <cstdlib>   // free
4  using namespace std;
5
6  int main() {
7      // wersja "gumowa"
8      ostrstream napis1;
9      napis1 << "Poczatek, " << "dalszy ciag, "
10            << "koniec."    << ends;
11     char* n = napis1.str();
```

```
12      cout << "Napis jest: " << n << endl;
13      free(n);
14
15      // wersja tablicowa
16      char tab[30];
17      ostrstream napis2(tab,sizeof(tab));
18      napis2 << "Magda " << "Kasia " << "Marta" << ends;
19      cout << tab << endl;
20  }
```

Program ten wypisuje na ekranie

```
Napis jest: Poczatek, dalszy ciag, koniec.
Magda Kasia Marta
```

Zauważmy, że pamięci na napis n nie alokujemy. Jest ona alokowana automatycznie i jej adres jest zwracany przez metodę **str** (linia 11).

Podobnie można z tablicy znaków czytać, tworząc obiekt klasy **istrstream**.

- `istrstream strum(char* t, size_t len);` — „plikiem" będzie obszar pamięci o długości len od bajtu wskazywanego przez tab.
- 'istrstream istr(char* tab);' — „plikiem" będzie tu obszar od bajtu wskazywanego przez tab do znaku NUL, który będzie traktowany jak znak końca pliku.

Pliki wewnętrzne oparte na C-napisach są czasem używane do reprezentowania pliku dyskowego, szczególnie jeśli zachodzi potrzeba częstego „skakania" do różnych części pliku, co może być drogie, gdyż wymaga wielu operacji dostępu do pliku. Może wtedy być bardziej opłacalne wczytanie całego pliku do tablicy znaków i dalej używanie jej jak pliku wewnętrznego.

16.7.2 Napisy C++ jako pliki wewnętrzne

Podobnie jak tablice znaków, do reprezentowania plików wewnętrznych mogą być używane obiekty klasy **string**. Jest to łatwiejsze i bezpieczniejsze niż używanie tablic znaków.

Aby korzystać z tego udogodnienia, należy dołączyć, prócz pliku ***string***, plik nagłówkowy ***sstream***. Zdefiniowane tam są klasy strumieniowe wejściowa **istringstream** i wyjściowa **ostringstream** korzystające z napisów klasy **string**.

Strumień wyjściowy można utworzyć za pomocą konstruktora domyślnego:

```
ostringstream strm;
```

i pisać do niego jak do normalnego strumienia

```
strm << "To be" << " or not to be" << ", etc.";
```

Nie trzeba martwić się o przepełnienie, bo pamięć w miarę potrzeby jest alokowana automatycznie. Po zakończeniu pisania do strumienia, wywołujemy na jego rzecz metodę **str**, która zwraca napis (obiekt klasy **string**) zawierający rezultat, na przykład:

```
string s = strm.str();
cout << s << endl;
```

Można też skonstruować strumień z wpisanym już fragmentem w trybie dopisywania na końcu

```
ostringstream ostr("Jakis napis", ios::ate);
ostr << "Dalsza czesc";
```

a następnie dopisywać dalsze fragmenty, jak w przykładzie powyżej.

Istniejący obiekt klasy **string** też może być otwarty jako wejściowy plik wewnętrzny, jak w przykładzie poniżej:

P140: ***intstr.cpp*** Napisy C++ jako pliki wewnętrzne

```
1  #include <iostream>
2  #include <sstream>
3  using namespace std;
4
5  void words(const string& s) {
6      istringstream istr(s);
7
8      string word;
9      while ( istr >> word )
10         cout << word << endl;
11 }
12
13 int main() {
14     string s = "Bach Haydn";
15     ostringstream ostr(s, ios::ate);
16     ostr << " Chopin";
17     string s1 = ostr.str();
18     words(s1);
19 }
```

Najpierw, w linii 15, otwieramy wewnętrzny plik wyjściowy zainicjowany zawartością napisu `s`. Dodajemy do niego pewne dane (linia 16) i pobieramy wynikową zawartość do `s1`. Następnie `s1` jest przekazywane do funkcji **words**, która ten napis otwiera do czytania jako wejściowy plik wewnętrzny (linia 6). Z pliku tego czytamy kolejne słowa oddzielone znakiem odstępu, według normalnych zasad czytania ze strumienia. Zauważmy, że po osiągnięciu końca pliku, stan strumienia przejdzie w `bad`, co zakończy pętlę z linii 9-10. Program drukuje zatem

```
Bach
Haydn
Chopin
```

Pokażemy jeszcze, jak czytać z i zapisywać do plików tekstowych. Spójrzmy na przykład

P141: ***RWfile.cpp*** Reading and writing text files

```
#include <iostream>
#include <fstream>   // ifstream, ofstream
#include <string>
#include <sstream>   // stringstream
using namespace std;

int main() {
    string line{};

    ofstream outf{"RWfile.out"};                                    ①
    for (ifstream in{"RWfile.dat"}; getline(in, line);) {           ②
        cout << line  << "\n";
        string name;
        int height;
        double weight;
        istringstream str{line};                                    ③
        str >> name >> height >> weight;
        cout << name << ": height=" << height
             << ", weight=" << weight << '\n';
        outf << name << ": height=" << height                       ④
             << ", weight=" << weight << '\n';
    }
    outf.close();

      // again but in a while loop
    ifstream in{"RWfile.dat"};
    while (getline(in, line)) {
        cout << line  << "\n";
    }
}
```

W linii ① tworzymy obiekt outf typu **ofstream** ('of' od *output stream*). Obiekt ten reprezentuje strumień wyjściowy, jak cout ale związany z plikiem ***RWfile.out*** (który zostanie utworzony, jeśli nie istnieje). Jak widzimy w linii ④, używamy outf dokładnie jak cout, ale tekst jest zapisywany w pliku, a nie na ekranie. W linii ② tworzymy obiekt in typu **ifstream** — to jest strumień wejściowy, jak cin ale dla którego dane są wczytywane z pliku a nie z klawiatury. Funkcja **getline** (z nagłówka ***string*** header)

pobiera strumień wejściowy i obiekt typu **string** (w naszym przypadku line), czyta jeden wiersz do line. Zwraca otrzymany strumień, który jest konwertowalny do wartości logicznej, która będzie **false** gdy napotkany zostanie koniec pliku.
Object str (③) jest typu **istringstream** (z nagłówka ***sstream***) i reprezentuje strumień wejściowy, dla którego źródłem jest przekazany do konstruktora napis (w naszym przypadku line). Jak widzimy możemy używać tego obiektu jak cin aby wczytać dane z line.
Z następującymi danymi w pliku ***RWfile.dat***

```
Mary 167 56.5
Jane 162 55.7
Kate 170 59.1
```

program wypisuje na ekran i do pliku ***RWfile.out***

```
Mary 167 56.5
Mary: height=167, weight=56.5
Jane 162 55.7
Jane: height=162, weight=55.7
Kate 170 59.1
Kate: height=170, weight=59.1
Mary 167 56.5
Jane 162 55.7
Kate 170 59.1
```

— 17 —

Napisy

Z napisami spotkaliśmy się już w poprzednich rozdziałach. W tym podamy więcej szczegółów; w szczególności dokonamy przeglądu funkcji i metod pozwalających na sprawne posługiwanie się napisami w C/C++.

PODROZDZIAŁY:

17.1 C-napisy

W tradycyjnym C typ **char** i **char*** są generycznymi typami służącymi do bezpośrednich odniesień do pamięci. Niektóre operacje na napisach są więc bardzo podobne do podanych wcześniej operacji działających bezpośrednio na pamięci (patrz rozdz. 12.6 na stronie 229).

Napis jest traktowany jak tablica znaków zawierająca znak NUL na końcu. Nazwa NUL (przez jedno 'L') jest nazwą znaku, ale nie odpowiada żadnej nazwie zdefiniowanej w języku. Literałem tego znaku jest '\0' — *nie* jest to w żadnym przypadku NULL (taki symbol, definiowany za pomocą dyrektywy preprocesora, oznacza *wskaźnik* pusty w C).

Należy pamiętać, że

> jeśli inicjalizujemy napis zadeklarowany jako typu **char*** literałem napisowym, to *nie* będzie on modyfikowalny.

Aby napis był modyfikowalny, należy go zadeklarować jawnie jako tablicę:

```
1    char* s1   = "Ula";  // Niemodyfikowalny, wymiar 4
2    char  s2[] = "Ula";  // Modyfikowalny,    wymiar 4
3    char  s3[] = {'U', 'l', 'a', '\0'}       // Jak s2
```

Definicje C-napisów s2 i s3 są w zasadzie równoważne. Forma z linii 2 jest oczywiście wygodniejsza, bo wymaga mniej pisania. Zauważmy, że w obu pierwszych

formach kompilator sam zadba o wstawienie jako ostatniego znaku tablicy znaku NUL (czyli '\0'). Używając formy z linii 3 musimy sami o tym pamiętać. We wszystkich przypadkach tablica zawierać będzie cztery znaki: trzy litery słowa Ula i znak NUL. W dwóch ostatnich przypadkach jest to zwykła tablica, utworzona na stosie. Natomiast tablica utworzona w sposób przedstawiony w linii 1 jest alokowana gdzie indziej i jest *niemodyfikowalna*. Należy o tym pamiętać przy przekazywaniu tego rodzaju napisów do funkcji. Typem s1 jest **char***, a więc kompilator zgodzi się na wysłanie s1 do funkcji, których odpowiedni parametr *nie* był wcale zadeklarowany jako **const char***. Tym niemniej, próba modyfikacji takiego napisu przez funkcję skończy się załamaniem programu. Używając zatem formy z linii 1 lepiej, mimo że nie jest to przez język wymagane, samemu zadeklarować typ wskaźnika jako **const char***. Kompilator będzie wówczas sprawdzał, czy funkcje, do których napis ten wysyłamy, „obiecują", poprzez deklarację typu parametru jako **const char***, że napisu tego nie zmodyfikują.

Podobnie jest z literałami napisowymi użytymi jako argument funkcji; jeśli wywołamy funkcję **fun** o parametrze typu **char*** z literałem jako argumentem, na przykład `func("Jan")`, to chociaż parametr nie jest zadeklarowany jako niemodyfikowalny, próba zmiany zawartości napisu wewnątrz funkcji nie uda się!

Do funkcji działających na C-napisach mamy dostęp poprzez dołączenie pliku nagłówkowego ***cstring***. Funkcje operujące na pojedynczych znakach zawarte są w ***cctype***, przydatne funkcje konwertujące natomiast — w ***cstdlib***.

17.1.1 Funkcje operujące na C-napisach

Wymieńmy najważniejsze z funkcji dostarczanych w pliku nagłówkowym ***cstring*** (***string.h*** w C). Należą one do standardu C i zawsze są dostępne.

size_t strlen(const char* s) — podaje długość napisu s w znakach *nie* licząc znaku NUL na jego końcu. Typ **size_t** jest pewnym typem całościowym.

char* strcat(char* dest, const char* src) — dodaje zawartość napisu src do napisu dest (konkatenacja); zwraca dest. Użytkownik musi zadbać o to, by dest i src kończyły się znakiem NUL i żeby obszar pamięci zarezerwowany pod adresem wskazywanym przez dest był wystarczający do pomieszczenia obu napisów (i kończącego znaku NUL). Pierwszy znak NUL w napisie dest jest nadpisywany przez pierwszy znak scr i poczynając od tej pozycji przekopiowywane są znaki z src aż do kończącego go znaku NUL włącznie. Częsty błąd polega na dodawaniu do niezainicjowanego napisu

```
char s[100];
strcat(s,"jakis napis");
```

co jest błędem, bo program będzie szukał znaku NUL w napisie s, ale go nie znajdzie!

char* strncat(char* dest, const char* src, size_t n) — jest podobne do funkcji poprzedniej, tylko kopiuje co najwyżej n znaków, kończąc kopiowanie, jeśli napotkany i przekopiowany został znak NUL. Jeśli znak NUL nie został napotkany po

przekopiowaniu n znaków, to go dostawia jako n+1-szy znak. W takim przypadku zapisanych więc zostanie w sumie n+1 znaków! Na przykład

```
char t1[20] = {'\0'}; // konieczne !
strncat(t1,"123456789",5);
cout << t1 << " ma " << strlen(t1) << " znakow" << endl;
```

wydrukuje '12345 ma 5 znakow', co świadczy o tym, że w tablicy t1 zapisanych zostało 6 znaków: pięć cyfr '12345' i znak NUL.

char* strcpy(char* dest, const char* src) — kopiuje znaki od wskazywanego przez src do obszaru wskazywanego przez dest. Dotychczasowa zawartość obszaru pamięci od adresu dest jest nadpisywana. Kopiowanie kończy się, gdy przekopiowanym znakiem jest znak NUL. Użytkownik musi zatem zadbać o to, żeby pod adresem dest była zarezerwowana odpowiednia ilość pamięci i żeby napis src kończył się znakiem NUL! Funkcja zwraca dest. Na przykład funkcja **strcat** mogłaby być zaimplementowana następująco:

```
char* strcat(char* dest, const char* src) {
    char* s = dest + strlen(dest);
    strcpy(s, src);
    return dest;
}
```

za pomocą **strcpy** i **strlen**.

char* strncpy(char* dest, const char* src, size_t n) — kopiuje dokładnie n znaków od wskazywanego przez src do obszaru wskazywanego przez dest. Dotychczasowa zawartość obszaru pamięci od adresu dest jest nadpisywana. Gdy napotkany zostanie znak NUL, jest kopiowany i dalej wpisywane są do obszaru docelowego same znaki NUL, aż całkowita liczba zapisanych znaków będzie wynosić dokładnie n. Jeśli wśród n pierwszych znaków w napisie źródłowym src znaku NUL nie będzie, to przekopiowanych zostanie n znaków a NUL dostawiony *nie* będzie.

int strcmp(const char* s1, const char* s2) — porównuje leksykograficznie („słownikowo") dwa napisy, zwracając -1, jeśli napis s1 jest wcześniejszy niż s2, $+1$, jeśli s1 jest późniejszy, i 0, jeśli napisy są identyczne. Napis s1 jest wcześniejszy, jeśli zachodzi któryś z poniższych warunków:
a) napisy są takie same aż do pewnej pozycji, a na pierwszej pozycji, na której występuje różnica, znak w s1 jest numerycznie mniejszy od odpowiadającego znaku z s2;
b) napis s1 jest krótszy niż s2 i wszystkie jego znaki (nie licząc kończącego znaku NUL) są takie same jak znaki w s2 na odpowiadających pozycjach.
Oba napisy muszą oczywiście być zakończone znakiem NUL. Na przykład

```
if ( ! strcmp(s1,s2) )
    cout << "Takie same\n";
else
    cout << "Rozne\n";
```

zadziała, bo jakakolwiek wartość niezerowa zwrócona przez **strcmp** zostanie potraktowana jak **true**, a po zanegowaniu operatorem '!' jak **false**.
Przykład poniższy to program wykorzystujący funkcję **strcmp**. Funkcja **first_last** pobiera tablicę napisów i przez referencje, aby móc je modyfikować, dwa wskaźniki typu **char*** (linie 20-26). Następnie wpisuje do wskaźnika p adres początku napisu z tablicy najwcześniejszego leksykograficznie, a do q adres napisu ostatniego w tym porządku.

P142: *sorslo.cpp* Porównywanie napisów

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  void  first_last(char**,char*&,char*&);
6
7  int main() {
8
9      char *nam[] = { "Katarzyna", "Magdalena",
10                     "Alicja",     "Wanda",
11                    "Izabela",    "Aldona", "" },
12                  *p,*q;
13
14     first_last(nam,p,q);
15
16     cout << "Pierwsza: " << p << endl
17          << "Ostatnia: " << q << endl;
18 }
19
20 void first_last(char** s, char*& p, char*& q) {
21     p = q = *s;
22     while ( **++s ) {
23         if ( strcmp(*s, p) < 0 ) p = *s;
24         if ( strcmp(*s, q) > 0 ) q = *s;
25     }
26 }
```

Zauważmy, że nie przesłaliśmy do funkcji wymiaru tablicy napisów (liczby imion); zamiast tego jako ostatni napis w tablicy umieściliśmy (linia 11) napis pusty — złożony wobec tego tylko ze znaku NUL, który zostanie automatycznie dodany przez kompilator. Jest to technika bardzo często stosowana w wielu funkcjach bibliotecznych C (dla tablic napisów). Kod funkcji **first_last** jest tu bardzo zwięzły; mógłby być dłuższy, ale bardziej czytelny. Jednak zrozumienie go w podanej formie jest dobrym ćwiczeniem na posługiwanie się wskaźnikami. Program drukuje oczywiście 'Pierwsza: Aldona Ostatnia: Wanda'.

int strcoll(const char* s1, const char* s2) — działa jak funkcja **strcmp**, tylko uwzględnia lokalizm. Na przykład, jeśli wybrane jest polskie *locale*, to litera 'ą'

jest za 'a', natomiast w wersji francuskiej 'â' jest równoważne zwykłemu 'a' przy porównywaniu leksykalnym.

int strncmp(const char* s1, const char* s2, size_t n) — jak funkcja **strcmp**, tylko porównuje co najwyżej n znaków.

char* strchr(const char* s, int c) — zwraca wskaźnik do pierwszego wystąpienia znaku (**char**)c w napisie s lub wskaźnik pusty **nullptr**, jeśli tego znaku w tym napisie nie ma. Na przykład

```
cout << strchr("Daniel Defoe",'f') << endl;
```

wypisze 'foe'. Funkcja zliczająca liczbę wystąpień znaku (**char**)c w napisie s mogłaby mieć postać

```
int count(const char* s, int c) {
    int n = 0;
    while (s = strchr(s,c)) ++s,++n;
    return n;
}
```

W szczególności *można* za pomocą **strchr** szukać znaku NUL.

char* strrchr(const char* s, int c) — zwraca wskaźnik do ostatniego wystąpienia znaku (**char**)c w napisie s lub wskaźnik pusty **nullptr**, jeśli tego znaku w tym napisie nie ma.

size_t strspn(const char* s, const char* set) — zwraca długość najdłuższego początkowego podciągu w s który składa się wyłącznie ze znaków zawartych w set (w szczególności może to być zero lub długość całego napisu s). Na przykład

```
cout << strspn("sound and fury","os nudda") << endl;
```

wypisze 10.

size_t strcspn(const char* s, const char* set) — zwraca długość najdłuższego początkowego podciągu w s, który składa się wyłącznie ze znaków *nie* zawartych w napisie set. Na przykład

```
cout << strcspn("sound and fury","xyztdv") << endl;
```

wypisze 4.

char* strpbrk(const char* s, const char* set) — zwraca wskaźnik do pierwszego znaku w s, który jest zawarty w set, lub **nullptr**, jeśli w set nie ma znaków występujących w s. Na przykład

```
cout << strpbrk("Daniel Defoe","wKlor") << endl;
```

wypisze 'l Defoe'.

char* strstr(const char* s, const char* sub) — zwraca wskaźnik do pierwszego znaku w s, od którego rozpoczyna się w nim podciąg identyczny z sub (pierwsze jego wystąpienie). Funkcja zwraca wskaźnik zerowy, jeśli podciąg sub nie występuje w s.

```
cout << strstr("Daniel Defoe","De") << endl;
```

wypisze 'Defoe'.

char* strtok(char* str, const char* set) — funkcja ta jest „tokenizerem", czyli funkcją pozwalającą rozłożyć napis na leksemy („słowa") oddzielone znakami ze zbioru znaków w napisie set, które wobec tego traktowane są jako separatory (np. odstęp, przecinek, dwukropek, itd.). Działanie funkcji oparte jest na istnieniu wewnętrznego wskaźnika typu **char***. Kolejne wywołania **strtok** zwracają kolejne leksemy. Przy pierwszym wywołaniu str jest napisem, który chcemy rozłożyć; w kolejnych wywołaniach pierwszym argumentem powinien być wskaźnik zerowy **nullptr**, co informuje funkcję, że należy kontynuować przetwarzanie od końca ostatnio zwróconego leksemu. Pomiędzy wywołaniami nie wolno zmieniać napisu str — sama funkcja go jednak zmienia, co oznacza, że nie można przekazać napisu niemodyfikowalnego, np. zadanego jako literał napisowy. Wolno natomiast przy kolejnych wywołaniach zmieniać zbiór separatorów w set. Funkcja działa następująco:

Jeśli str nie jest **nullptr** (czyli kiedy rozpoczynamy analizę pewnego napisu), ignorowane są wszystkie wiodące znaki z str, które należą do zbioru separatorów set. Jeśli wszystkie znaki w str należą do set, czyli w napisie są same separatory, to funkcja zwraca **nullptr** i wskaźnik wewnętrzny ustawiany jest też na **nullptr**, co kończy przetwarzanie napisu. Jeśli natomiast tak nie było, to wewnętrzny wskaźnik ustawiany jest na pierwszy napotkany znak nie będący separatorem i dalej postępowanie jest takie jak w przypadku, gdy str jest **nullptr**.

Jeśli str jest **nullptr** i wskaźnik wewnętrzny też jest **nullptr**, to zwracany jest **nullptr** i stan wewnętrznego wskaźnika jest niezmieniany. Kończy to analizę jednego napisu.

Jeśli str jest **nullptr**, ale wskaźnik wewnętrzny nie jest **nullptr**, to funkcja szuka, poczynając od miejsca wskazywanego przez wewnętrzny wskaźnik, pierwszego wystąpienia separatora. Jeśli zostanie znaleziony, to jest nadpisywany znakiem pustym '\0', funkcja zwraca adres zawarty we wskaźniku wewnętrznym, a sam wewnętrzny wskaźnik jest przesuwany na pierwszy znak za wpisanym znakiem '\0'. Jeśli natomiast nie zostanie znaleziony, to funkcja zwraca adres zawarty we wskaźniku wewnętrznym, a sam wewnętrzny wskaźnik jest ustawiany na **nullptr**.

Brzmi to bardzo skomplikowanie, ale użycie nie jest takie trudne. Na przykład poniższy program

P143: *tok.cpp* Tokenizer w C

```
#include <iostream>
#include <cstring>
using namespace std;

int main() {
    char strin[] = "int* fun(char& c,double** wtab);";
    char separ[] = ")(,;";
    char* token;

    token = strtok(strin,separ);
    while (token != 0) {
```

```
12          cout << token << endl;
13          token = strtok(0,separ);
14      }
15  }
```

drukuje w kolejnych liniach
`int* fun`
`char& c`
`double** wtab`

Funkcje operujące na napisach są dobrym ćwiczeniem programistycznym. Na przykład niektóre ze standardowych funkcji udostępnianych przez nagłówek ***cstring*** mogą być zaimplementowane w pokazany poniżej sposób (i na wiele innych sposobów):

P144: ***emulstr.cpp*** Implementacja funkcji napisowych

```
1  char* Strcpy(char* target, const char* source) {
2      char* t = target;
3      while ( *t++ = *source++ );
4      return target;
5  }
6
7  char* Strcat(char* target, const char* source) {
8      char* t = target-1;
9      while ( *++t );
10     while ( *t++ = *source++ );
11     return target;
12 }
13
14 char* Strncat(char* target, const char* source, int n) {
15     char* t = target-1;
16     while ( *++t );
17     while ( (*t++ = *source++) && n--);
18     *(t-1) = '\0';
19     return target;
20 }
21
22 int Strlen(const char* source) {
23     const char* s = source;
24     while ( *s++ );
25     return s-source-1;
26 }
27
28 char* Strchr(const char* target, int c) {
29     while ( *target && *target++ != c );
30     return (char*)(*--target ? target : 0);
31 }
```

Można napisać te funkcje jeszcze zwięźlej, choć niekoniecznie bardziej zrozumiale...

17.1.2 Funkcje operujące na znakach

Po dołączeniu pliku nagłówkowego ***cctype*** mamy do dyspozycji szereg funkcji operujących na pojedynczych znakach.

Duża grupa funkcji, wszystkie o nazwach rozpoczynających się od `is`, to funkcje sprawdzające, czy znak przesłany jako argument spełnia określone kryteria. Argument jest typu **int**, ale brany pod uwagę jest tylko jego najmłodszy bajt. Wartością zwracaną jest zawsze wartość typu **int**: niezerowa oznacza **true**, zerowa **false**. Tak więc wszystkie te funkcje mają nagłówek typu

```
int isJakasWlasnosc(int c);
```

Nie ma gwarancji (i zwykle tak nie jest), że **true** odpowiada wartości 1; może to być *dowolna* wartość niezerowa.

Wymieńmy funkcje z tej grupy:

- **isalnum** — czy `(char)c` jest literą lub cyfrą?
- **isalpha** — czy `(char)c` jest literą?
- **isdigit** — czy `(char)c` jest cyfrą?
- **isxdigit** — czy `(char)c` jest cyfrą szesnastkową? Cyfry szesnastkowe to cyfry z zakresu 0-9 oraz litery, małe i duże, z zakresu A-F.
- **iscntrl** — czy `(char)c` jest znakiem kontrolnym? Znaki kontrolne to znaki o kodzie ASCII między 0 a 31 włącznie oraz znak o kodzie ASCII 127.
- **isprint** — czy `(char)c` jest znakiem drukowalnym? Znaki drukowalne to wszystkie znaki, które nie są znakami kontrolnymi.
- **isgraph** — czy `(char)c` jest znakiem widocznym? Znaki widoczne to wszystkie znaki drukowalne (patrz wyżej) z wyjątkiem znaku odstępu (o kodzie ASCII 32).
- **ispunct** — czy `(char)c` jest znakiem interpunkcyjnym? Znaki interpunkcyjne to wszystkie znaki widoczne (patrz wyżej) z wyjątkiem liter i cyfr.
- **isspace** — czy `(char)c` jest białym znakiem? Białe znaki to: '\t' (tabulacja pozioma, symbol HT, kod ASCII 9), '\r' (powrót karetki, CR, 13), '\n' (nowa linia, LF, 10), '\v' (tabulacja pionowa, VT, 11), '\f' (wysuw strony, FF, 12) i odstęp (SPC, 32).
- **isupper** — czy `(char)c` jest dużą literą?
- **islower** — czy `(char)c` jest małą literą?

Prócz tego biblioteka dołączana przez nagłówek ***cctype*** zawiera dwie przydatne standardowe funkcje, również typu **int** → **int**:

- **tolower** — zwraca dla znaku, który jest dużą literą, odpowiadający mu znak małej litery (H → h). Dla znaków nie będących dużą literą zwraca ten sam znak.
- **toupper** — zwraca dla znaku, który jest małą literą, odpowiadający mu znak dużej litery (n → N). Dla znaków nie będących małą literą zwraca ten sam znak.

Na przykład funkcja **uplow** w programie

P145: *upplow.cpp* Operacje na znakach

```
#include <iostream>
#include <cctype>
using namespace std;

int uplow(char* s) {
    int cnt = 0;
    do {
        if (isalpha(*s))
            if ( cnt == 0 || !isalpha(*(s-1))) {
                *s = (char)toupper(*s);
                cnt++;
            } else
                *s = (char)tolower(*s);
    } while (*s++);
    return cnt;
}

int main() {
    char napis[] = "to jEST DlUgI,dluGI nAPIs!";

    int ile = uplow(napis);
    cout << ile << " slow, napis = \'" << napis << "\'\n";
}
```

zamienia w przekazanym napisie wszystkie początkowe litery słów na duże, a pozostałe na małe, zwracając liczbę znalezionych słów. Uruchomienie tego programu daje:
5 slow, napis = 'To Jest Dlugi,Dlugi Napis!'.

17.1.3 Funkcje konwertujące

Plik nagłówkowy *cstdlib* dostarcza kilku przydatnych funkcji konwertujących. Stosowanie ich wymaga starannego sprawdzania możliwych błędów. Służy do tego globalna zmienna errno (może to być makro preprocesora, ze względu na programy wielowątkowe, ale z punktu widzenia programisty zachowuje się jak zmienna globalna). Aby móc z niej korzystać, należy dołączyć plik nagłówkowy *cerrno*.

double strtod(const char* str, char ptr)** — (*string to double*) zwraca liczbę typu **double** zapisaną w początkowej części napisu str. Wiodące białe znaki są ignorowane. Wczytywanie znaków kończy się po napotkaniu pierwszego „złego” znaku, to znaczy znaku, który nie może być traktowany jako kontynuacja zapisu wczytywanej liczby. Jest rozpoznawany zapis liczb w postaci liczb całkowitych dziesiętnych (jak 127), liczb w zapisie z kropką dziesiętną (jak 123,34) i liczb w formacie naukowym (z literą 'e', dużą lub małą, przed wykładnikiem potęgi dziesięciu hyfn 1.2E-11).

Wskaźnik str wskazuje napis zawierający na początku liczbę przeznaczoną do „wyczytania”. Za tą liczbą mogą występować w napisie inne znaki: pozostaną one w strumieniu wejściowym i będą mogły być wczytane przez następną operację czytania. Drugim argumentem funkcji powinien być *adres* wskaźnika typu **char*** — dlatego typem parametru jest **char****. Może to być adres pusty (czyli **nullptr** albo po prostu zero), ale nie jest to wskazane: lepiej przesłać do funkcji adres istniejącej zmiennej wskaźnikowej typu **char***.

Po prawidłowej konwersji zwracana jest „wyczytana” liczba w postaci wartości typu **double**. Jeśli ptr nie był pusty, do wskaźnika wskazywanego przez ptr wpisywany jest adres pierwszego znaku w napisie str *za* wczytaną reprezentacją liczby (w szczególności może to być adres kończącego napis str znaku '\0'. Jeśli posłaliśmy do funkcji **nullptr** jako drugi argument, to informacji tej nie dostaniemy. Wartość errno jest zero.

Jeśli napis nie zawiera legalnego zapisu liczby, to zwracane jest zero, a do wskaźnika wskazywanego przez ptr, jeśli nie był to wskaźnik pusty, wpisywany jest adres początku napisu str, czyli wartość samego str. Tę równość wskaźników można zatem sprawdzić w programie, gdy zwróconą wartością jest zero — jeśli wartości tych wskaźników są takie same, to znaczy, że zwrócone zero nie jest „prawdziwe”.

Jeśli napis zawiera legalny zapis liczby, ale liczba ta przekracza dopuszczalny zakres (nadmiar, ang. *overflow*), zwracana jest wartość ±HUGE_VAL z właściwym znakiem (ta stała oznacza inf — nieskończoność). Zmienna errno jest wtedy ustawiana na ERANGE. Do wskaźnika wskazywanego przez ptr wpisywany jest adres pierwszego znaku za reprezentacją liczby, tak jak w przypadku udanej konwersji.

Jeśli napis zawiera legalny zapis liczby, ale liczba ta ma za małą wartość bezwzględną, aby była odróżnialna od zera (niedomiar, ang. *underflow*), zwracane jest zero. Zmienna errno jest ustawiana na ERANGE, a do wskaźnika wskazywanego przez ptr wpisywany jest adres pierwszego znaku za reprezentacją liczby, tak jak w przypadku udanej konwersji.

Następujący programik ilustruje te definicje:

P146: ***strtod.cpp*** Funkcja konwertująca *strtod*

```
#include <iostream>
#include <iomanip>   // setw
#include <cstdlib>   // strtod
#include <cerrno>    // errno
using namespace std;

int main() {
```

```
8     char*  ptr;
9     double x;
10    char*  str;
11
12    cout << "ERANGE = " << ERANGE << endl;
13
14    // = 1 = OK
15    str = "-1.2e+2xxx";
16    x = strtod(str,&ptr);
17    cout << "=1=  str = " << str << "; x = "
18         << setw(4) << x << "; errno = " << setw(2)
19         << errno << "; ptr = " << ptr << endl;
20
21    // = 2 =  Not a Number
22    str = "abcdefghij";
23    x = strtod(str,&ptr);
24    cout << "=2=  str = " << str << "; x = "
25         << setw(4) << x << "; errno = " << setw(2)
26         << errno << "; ptr = " << ptr << endl;
27
28    // = 3 =  Overflow
29    str = "-9e+9999xx";
30    x = strtod(str,&ptr);
31    cout << "=3=  str = " << str << "; x = "
32         << setw(4) << x << "; errno = " << setw(2)
33         << errno << "; ptr = " << ptr << endl;
34
35    // = 4 =  Underflow
36    str = "-9e-9999xx";
37    x = strtod(str,&ptr);
38    cout << "=4=  str = " << str << "; x = "
39         << setw(4) << x << "; errno = " << setw(2)
40         << errno << "; ptr = " << ptr << endl;
41 }
```

Wydruk z tego programu:

```
ERANGE = 34
=1=  str = -1.2e+2xxx; x = -120; errno =  0; ptr = xxx
=2=  str = abcdefghij; x =    0; errno =  0; ptr = abcdefghij
=3=  str = -9e+9999xx; x = -inf; errno = 34; ptr = xx
=4=  str = -9e-9999xx; x =    0; errno = 34; ptr = xx
```

Na podobnej zasadzie działają funkcje:

long strtol(const char* str, char ptr)** — (*string to long*) zwraca liczbę typu **long** zapisaną w początkowej części napisu **str**. Działa tak jak opisana wcześniej

funkcja **strtod**, tyle że w przypadku nadmiaru zwraca nie ±HUGE_VAL, ale stałe LONG_MAX lub LONG_MIN. Niedomiar oczywiście wystąpić nie może.

long strtol(const char* str, char ptr, int base)** — jak funkcja poprzednia, ale zwraca liczbę typu **long** odczytując ją z napisu str w układzie o podstawie base. Podstawa może być liczbą od 2 do 36. Dla podstaw większych od dziesięciu jako cyfry interpretowane są kolejne litery (duże lub małe) poczynając od litery 'a', która oznacza cyfrę 10. Jeśli podstawa wynosi 16, reprezentacja napisowa liczby może się zaczynać od '0x' (lub '0X') — te znaki są wtedy ignorowane. Na przykład `strtol("J23",&ptr,20)` zwraca $7643 = 19 \cdot 400 + 2 \cdot 20 + 3$.

unsigned long strtoul(const char* str, char ptr)** — (*string to unsigned long*) zwraca liczbę typu **unsigned long** zapisaną w początkowej części napisu str. Działa tak jak dwuargumentowa funkcja **strtol**, tyle że w przypadku nadmiaru zwraca ULONG_MAX.

unsigned long strtoul(const char* str, char ptr, int base)** — jak trzyargumentowa funkcja **strtol**, ale zwraca wartość typu **unsigned long**.

Prócz wymienionych istnieją, ze względu na wsteczną zgodność, tradycyjne choć niezalecane funkcje konwertujące:

double atof(const char* str) — (*ascii to floating*) działa podobnie jak funkcja **strtod**, ale nie ma drugiego argumentu. W przypadku błędu zwraca zero, lub jeśli nastąpił nadmiar, ±HUGE_VAL. Ustawia, niezależnie od przyczyny niepowodzenia, wartość errno na ERANGE.

int atoi(const char* str) — (*ascii to integer*) działa podobnie jak funkcja **strtol**, ale zwraca wartość typu **int** (a nie **long**) i nie ma drugiego argumentu. Obsługa błędów podobnie jak dla funkcji **atof**.

long atol(const char* str) — (*ascii to long*) działa podobnie jak funkcja **strtol**, ale nie ma drugiego argumentu. Obsługa błędów podobnie jak dla funkcji **atof**.

17.2 Napisy w C++

W języku C++ zdefiniowana jest klasa **string**, dostępna po dołączeniu pliku nagłówkowego ***string***. Pozwala ona na tworzenie napisów i manipulowanie nimi łatwiej i bezpieczniej niż dla C-napisów. Napisy będące obiektami klasy **string** nazywać będziemy napisami C++, dla odróżnienia ich od C-napisów, czyli zwykłych tablic znaków zakończonych znakiem '\0'. Napisy C++ mogą zawierać dowolne znaki reprezentowalne w jednym bajcie, również znaki narodowe, na przykład polskie. W szczególności *mogą* zawierać znak '\0', niekoniecznie jako ostatni. Dla języków, w których nie da się reprezentować znaków w jednym bajcie (na przykład chińskiego), stosowane są inne klasy, którymi nie będziemy się zajmować.

Obiekty tej klasy są jednocześnie kolekcjami (znaków); dostępne zatem dla nich są narzędzia dostarczane przez bibliotekę standardową, a operujące właśnie na kolekcjach. Przykłady poznamy w tym rozdziale, ale szersze omówienie tego aspektu odłożymy do rozdziału24 na stronie 519.

Aby móc korzystać z klasy **string**, musimy włączyć plik nagłówkowy ***string***. Klasa definiuje między innymi:

- typ **string::size_type**, zwykle tożsamy z typem **unsigned**. Tego typu są długości napisów i podnapisów. Jeśli w programie jawnie używamy nazwy tego typu (co rzadko bywa niezbędne), może być konieczne kwalifikowanie jej nazwą klasy, a więc należy pisać **string::size_type**.
- stałą string::npos, równą największej możliwej wartości typu **string::size_type**. Żaden napis nie może mieć takiej długości, wobec tego wartość ta jest wykorzystywana jako sygnalizator błędów albo wartość domyślna oznaczająca *tyle znaków ile się da*.

17.2.1 Konstruktory

Obiekt klasy **string** można utworzyć na kilka sposobów:

string( )
string(const string& wzor, size_type start = 0, size_type ile = npos)
string(const char* wzor)
string(const char* wzor, size_type ile)
string(size_type ile, char c)
string(const char* start, const char* kon) — są przeciążonymi konstruktorami klasy **string**. W ostatnim z nich typem argumentów może być dowolny iterator wskazujący na znaki: w najprostszym przypadku są to po prostu wskaźniki do znaków C-napisu.

Na przykład

```
string s;
```

tworzy napis pusty;

```
string s1 = "Acapulco";
string s2("Acapulco");
```

tworzy napis zainicjowany kopią C-napisu;

```
string s("Acapulco",n);
```

tworzy napis zainicjowany kopią pierwszych n znaków C-napisu podanego jako pierwszy argument. Argument n jest typu **size_type**;

```
string s(n,'x');
```

tworzy napis zainicjowany n powtórzeniami znaku (w tym przypadku znaku 'x'). Argument n jest typu **size_type**. Dla n=npos zgłasza wyjątek length_error. Zauważmy, że *nie ma* konstruktora pobierającego pojedynczy znak jako jedyny argument. Nie jest to wielka strata, gdyż zawsze możemy użyć powyższego konstruktora w formie `s(1,'x')`.
Jeśli s jest obiektem klasy **string**, to

```
string s1(s);
```

tworzy napis zainicjowany kopią napisu s (jest to wywołanie konstruktora kopiującego);

```
string s2(s,n);
```

tworzy napis zainicjowany kopią napisu s poczynając od znaku na pozycji n, licząc pozycje, jak zwykle, od zera. Argument n jest typu **size_type**. Jeśli ma wartość większą lub równą długości napisu s, to zgłaszany jest wyjątek out_of_range. Zauważmy różnicę: jeśli s byłoby C-napisem, to n miałoby interpretację liczby znaków licząc od początku!

```
string s2(s,n,k);
```

tworzy napis zainicjowany kopią napisu s poczynając od pozycji n i uwzględniającej co najwyżej k znaków. Jeśli wartość n+k jest większa od długości napisu s to błędu nie ma: interpretowane to jest jako *wszystkie znaki od pozycji* n. W szczególności wartością k może być npos. Wszystkie trzy ostatnie przypadki są implementowane w postaci jednego konstruktora o dwóch parametrach domyślnych:

```
string(const string& s, size_type start = 0,
                        size_type   ile = npos);
```

Jest też konstruktor wykorzystujący jawnie fakt, że obiekt klasy **string** jest kolekcją znaków. Jego szczególnym przypadkiem jest konstruktor, którego dwoma argumentami są dwa *wskaźniki* do znaków w zwykłym C-napisie: nowo utworzony napis C++ zainicjowany zostanie ciągiem znaków od tego wskazywanego przez pierwszy argument (włącznie) do znaku wskazywanego przez drugi argument, ale *bez* niego (czyli *wyłącznie*). Na przykład

```
const char* cnapis = "0123456789";
string s(cnapis+1,cnapis+7);
cout << s << endl;
```

wydrukuje '123456'. Jest to przykład bardziej ogólnego mechanizmu iteratorów, o których powiemy więcej w rozdz. 24.1.2 na stronie 522. Na razie możemy iteratory traktować jako pewne uogólnienie wskaźników. Na przykład

```
string  s("Warszawa");
string  s1(s.begin()+3, s.end()-2);
cout << s1 << endl;
```

wydrukuje 'sza'. Metody **begin** i **end** zwracają iteratory wskazujące na pierwszy oraz na *pierwszy za ostatnim* znak napisu. Dodawanie do i odejmowanie od iteratorów liczb całkowitych działa podobnie jak dla wskaźników. Jak zwykle, pierwszy iterator wskazuje na pierwszy znak który ma być uwzględniony, podczas gdy drugi na pierwszy znak który ma już być opuszczony.

17.2.2 Metody i operatory

Dla obiektów klasy **string** zdefiniowane jest działanie operatora przypisania. Na obiekt tej klasy można przypisywać zarówno inne napisy C++, jak i C-napisy oraz pojedyncze znaki.

```
const char* cstr = "strin";
string s1, s2, s3, s(" C++");
s1 = cstr;
s2 = 'g';
s3 = s;
cout << s1 << s2 << s3 << endl;
```

wydrukuje 'string C++'. Przypisanie jest głębokie, co znaczy, że na przykład po przypisaniu `s1=s2` obiekt `s1` jest całkowicie niezależny od obiektu `s2`: późniejsze zmiany `s2` nie wpływają na `s1` i *vice versa*.

Podobnie jak w Javie napisy można składać („konkatenować") za pomocą przeciążonego operatora dodawania. „Dodanie" do napisu C++ innego napisu C++, C-napisu lub znaku powoduje utworzenie nowego napisu C++ będącego złożeniem argumentów. Pamiętać trzeba tylko, aby zawsze jednym z argumentów takiego dodawania był napis C++. Na przykład

```
string s1 = "C";
const char* cn = "string";

string s = s1 + '-' + cn;
cout << s << endl;
```

utworzy i wypisze 'C-string', bo wynikiem pierwszego złożenia będzie napis C++ zawierający 'C-', który następnie zostanie złożony z C-napisem 'string'. Natomiast konstrukcja

```
const char* cn = "C";
string s1 = "string";

string s = cn + '-' + s1;
cout << s << endl;
```

byłaby błędna, gdyż pierwsze „dodawanie" dotyczyłoby C-napisu i znaku, a nie napisu C++.

Operator '+=' dodaje prawy argument, który może być napisem C++, C-napisem lub pojedynczym znakiem, do napisu C++ będącego lewym argumentem.

Napis C++ może też być traktowany jako tablica znaków. Operator indeksowania działa zgodnie z oczekiwaniem:

```
string s("Basia");
s[0] = 'K';
for (int i = 0; i < 5; i++) cout << s[i];
```

wyświetli napis 'Kasia', a więc wyrażenie `s[i]` jest referencją do i-tego znaku napisu, licząc oczywiście od zera. Nie sprawdzany jest przy tym zakres i (za to działanie operatora indeksowania jest bardzo szybkie).

W sposób zgodny z oczekiwaniem działają też operatory porównania '==', '!=', '>', '>=', '<', '<='. Jednym z argumentów może być C-napis. Porównania dokonywane są według porządku leksykograficznego. Dzięki temu w poniższym programie

P147: ***krols.cpp*** Sortowanie napisów

```
#include <iostream>
#include <string>
#include <iomanip>
using namespace std;

void insertionSort(string[],int);

int main() {
    int i;
    string krolowie[] = {
          string("Zygmunt"),    string("Michal"),
          string("Wladyslaw"), string("Anna"),
          string("Jan"),        string("Boleslaw")
                        };

    const int ile = sizeof(krolowie)/sizeof(string);

    insertionSort(krolowie, ile);

    for ( i = 0; i < ile; i++ )
        cout << setw(10) << krolowie[i] << endl;
}

void insertionSort(string a[], int wymiar) {
    if ( wymiar <= 1 ) return;

    for ( int i = 1 ; i < wymiar ; ++i ) {
        int j = i;
        string v = a[i];
        while ( j >= 1 && v < a[j-1] ) {
            a[j] = a[j-1];
            j--;
        }
        a[j] = v;
    }
}
```

napisy mogą być traktowane przez funkcję sortującą tak jak typy numeryczne;

porównaj ten program z programem ***krol.cpp*** (str. 247). Program drukuje:

```
     Anna
 Boleslaw
      Jan
   Michal
Wladyslaw
  Zygmunt
```

Dla obiektów klasy **string** zdefiniowano też działanie standardowych operatorów wstawiania i wyjmowania ze strumienia, '<<' i '>>'. Jak zwykle operator '>>' działa tak, że pomijane są wiodące białe znaki, a wczytywanie kończy się po napotkaniu pierwszego białego znaku za napisem — nie da się więc wczytać w ten sposób napisu złożonego z wielu słów.

Klasa **string** posiada też szereg *metod* pozwalających na łatwe manipulowanie napisami (metody, a więc wywoływane zawsze na rzecz konkretnego obiektu):

size_type size( )
size_type length( ) — zwracają długość napisu. Na przykład jeśli `s="Ula"`, to `s.size()` zwróci 3.

bool empty( ) — zwraca, w postaci wartości logicznej, odpowiedź na pytanie *czy napis jest pusty?*

char& at(size_type n) — zwraca referencję do n-tego znaku (licząc od zera) z napisu, na rzecz którego została wywołana. Zakres *jest* sprawdzany: jeśli n jest większe od lub równe długości napisu, wysyłany jest wyjątek out_of_range. Metoda ta zatem ma działanie podobne do operatora indeksowania, ale, ze względu na sprawdzanie zakresu, jest mniej efektywna, choć bardziej bezpieczna. Na przykład `string("Ula").at(2)` zwraca referencję do litery 'a'.

void resize(size_type n, char c = '\0') — zmienia rozmiar napisu na n. Jeśli n jest mniejsze od aktualnej długości napisu, pozostałe znaki są usuwane. Jeśli n jest większe od długości napisu, napis jest uzupełniany do długości n znakami c — domyślnie znakami '\0'.

void clear( ) — usuwa wszystkie znaki z napisu, pozostawiając go pustym. Równoważna wywołaniu metody **resize(0)**.

string substr(size_type start = 0, size_type ile = npos) — zwraca napis będący podciągiem napisu na rzecz którego metoda została wywołana. Podciąg składa się ze znaków od pozycji start i liczy ile znaków. Jeśli start+ile jest większe niż długość napisu, to błędu nie ma; do podciągu brane są wszystkie znaki napisu od tego na pozycji start. Na przykład

```
string s("Pernambuco");
cout << s.substr(5,3) << endl;
```

wypisze 'mbu'.

size_type copy(char cn[], size_type ile, size_type start = 0) — kopiuje *do* C-napisu cn podciąg złożony z ile znaków, poczynając od tego na pozycji start. Zwraca liczbę przekopiowanych znaków. Znak '\0' nie jest dostawiany. Zwróćmy uwagę na kolejność argumentów start i ile — odwrotną niż w metodzie **substr**. Na przykład

```
char nap[] = "xxxxxx";
string s("Barbara");
string::size_type siz = s.copy(nap,3,2);
cout << "Skopiowano " << siz << " znaki: \n"
     << nap << endl;
```

wypisze 'Skopiowano 3 znaki: rbaxxx'.

void swap(string s1) — zamienia napis s1 z tym, na rzecz którego metodę wywołano. Na przykład

```
string s("Arles"), s1("Berlin");
s.swap(s1);
cout << s << " " << s1 << endl;
```

wypisze 'Berlin Arles'.

string& assign(const string& wzor)
string& assign(const char* wzor)
string& assign(string wzor, size_type start, size_type ile)
string& assign(const char* wzor, size_type ile)
string& assign(size_type ile, char c)
string& assign(const char* start, const char* kon) — ustala zawartość napisu i zwraca referencję do niego (zastępuje operator przypisania). Argumenty mają podobną postać jak dla konstruktorów. W ostatniej metodzie typem argumentów może być iterator wskazujący na znaki: na przykład są to po prostu wskaźniki do znaków C-napisu.

W najprostszym przypadku argumentem jest inny napis w postaci napisu C++ lub C-napisu; tak więc po

```
string s1("xxx"), s2("Zuzia");
s1.assign(s2);
s2.assign("Kasia");
```

wartością s1 będzie 'Zuzia' a zmiennej s2 'Kasia'.
Tak jak dla konstruktorów, jako argument można podać podnapis napisu C++ o podanej pozycji początku i długości — za duża długość znaczy *aż do końca*. Można też podać podnapis C-napisu złożony z podanej liczby znaków licząc od początku:

```
string s1("0123456789"), s2;
const char* p = "0123456789";
s1.assign(s1,2,5);
s2.assign(p, 5);
cout << s1 << " " << s2 << endl;
```

wypisze '23456 01234'.
Za pomocą metody **assign** można utworzyć napis złożony z ile powtórzeń znaku (piąta forma z wymienionych powyżej). Można też użyć wskaźników do znaków w C-napisie jako iteratorów wyznaczających podciąg; trzeba tylko pamiętać, że wskazywany podnapis *nie* zawiera wtedy znaku wskazywanego jako górne ograniczenie podciągu:

```
string s1("0123456789"), s2;
const char* p = "0123456789";
s1.assign(5,'x');
s2.assign(p+3,p+5);
cout << s1 << " " << s2 << endl;
```

wypisze 'xxxxx 34'.

string& insert(size_type gdzie, const string* wzor)
string& insert(size_type gdzie, const char* wzor)
string& insert(size_type gdzie, string wzor, size_type start, size_type ile)
string& insert(size_type gdzie, const char* wzor, size_type ile)
string& insert(size_type gdzie, size_type ile, char c) — modyfikuje napis i zwraca referencję do niego, wstawiając na pozycji o indeksie gdzie znaki z innego napisu, opisywanego przez pozostałe argumenty. Znaki od pozycji gdzie są „przesuwane" w prawo za fragment wstawiony. Znaczenie argumentów określających ciąg znaków do wstawienia jest takie samo jak dla metody **assign**. Na przykład

```
string s1("mama"), s2("plastyka");
const char* p = "temat";
s1.insert(2,p,2).insert(6,s2,4,4);
cout << s1 << endl;
```

wypisze 'matematyka'.

Prócz tych form istnieją jeszcze trzy inne formy tej metody:

iterator insert(iterator gdzie, char c)
void insert(iterator gdzie, size_type ile, char c)
void insert(iterator gdzie, const char* start, const char* kon) — gdzie pierwszym argumentem jest iterator (uogólniony wskaźnik) do znaku w napisie, przed który wstawiany jest ciąg określany przez pozostałe argumenty. W trzeciej z tych form rolę dwóch ostatnich argumentów mogą pełnić nie tylko wskaźniki do znaków, ale ogólnie iteratory wskazujące na znaki (na przykład iteratory typu **string::iterator**). Na przykład

```
string s1("abbccdd");
s1.insert(s1.insert(s1.begin()+5,'c')+1,2,'d');
cout << s1 << endl;
```

wypisze 'abbcccdddd'.

string& append(const string& wzor)
string& append(const string& wzor, size_type start, size_type ile)

string& append(const char* wzor)
string& append(const char* wzor, size_type ile)
string& append(size_type ile, char c)
string& append(const char* start, const char* kon) — modyfikuje napis i zwraca referencję do niego, dodając na końcu tego napisu napis określany przez argumenty. Znaczenie argumentów określających ciąg znaków do wstawienia jest takie samo jak dla metod **insert**. W ostatniej metodzie typem argumentów może być dowolny iterator wskazujący na znaki: tu są to po prostu wskaźniki do znaków C-napisu.

string& erase(size_type start = 0, size_type ile = npos)
iterator erase(iterator start)
iterator erase(iterator start, iterator kon) — usuwa fragment napisu, zwracając referencję do zmodyfikowanego napisu lub iterator odnoszący się do zmodyfikowanego napisu i wskazujący na pierwszy znak *za* fragmentem usuniętym. Pierwsza forma usuwa ile znaków (domyślnie npos, czyli wszystkie) od pozycji start (domyślnie od pozycji zerowej). Druga forma usuwa wszystkie znaki od pozycji wskazywanej przez iterator start, a trzecia od znaku wskazywanego przez iterator start do znaku *poprzedzającego* znak wskazywany przez iterator kon. Na przykład

```
string s("0123456789");
string::iterator it = s.erase(s.begin()+3,s.end()-3);
cout << s << " " << *it << endl;
```

wypisze '012789 7'.

string& replace(size_type start, size_type ile, const string& wzor)
string& replace(size_type start, size_type ile, const string& wzor, size_type s, size_type i)
string& replace(size_type start, size_type ile, const char* wzor, size_type i)
string& replace(size_type start, size_type ile, const char* wzor)
string& replace(size_type start, size_type ile, size_type i, char c)
string& replace(iterator start, iterator kon, const string& wzor)
string& replace(iterator start, iterator kon, const char* wzor)
string& replace(iterator start, iterator kon, const char* wzor, size_type i)
string& replace(iterator start, iterator kon, size_type i, char c)
string& replace(iterator start, iterator kon, const char* st1, const char* kn1) — usuwa fragment napisu określony pierwszymi dwoma argumentami i wstawia na to miejsce napis określony pozostałymi argumentami. W ostatniej z tych metod typem dwóch ostatnich argumentów może być dowolny iterator wskazujący na znaki: w najprostszym przypadku są to po prostu wskaźniki do znaków C-napisu. Zasady określania napisów lub podnapisów są te same co dla metod **insert**. Metody **replace** zwracają referencję do zmodyfikowanego napisu. Na przykład

```
string s("0123456789");
const char* p("abcdef");
s.replace(0,2,p,2).replace(s.end()-2,s.end(),p+4,p+6);
cout << s << endl;
```

wypisze 'ab234567ef'.

size_type find(const string* s, size_type start = 0)
size_type find(const char* p, size_type start = 0)
size_type find(const char* p, size_type start, size_type ile)
size_type find(char c, size_type start = 0)
size_type rfind(const string* s, size_type start = npos)
size_type rfind(const char* p, size_type start = npos)
size_type rfind(const char* p, size_type start, size_type ile)
size_type rfind(char c, size_type start = npos) — szukają, poczynając od pozycji start, podnapisu określonego przez pozostałe argumenty. Rodzina metod **rfind** działa analogicznie, ale przeglądanie odbywa się od pozycji startowej w kierunku początku napisu. Wszystkie metody zwracają pozycję (indeks) pierwszego znaku poszukiwanego podnapisu w napisie przeszukiwanym. Jeśli przeszukiwanie zakończyło się porażką, zwracane jest npos. W przykładzie poniżej **find** szuka w napisie 'abc345abcAB' poczynając od pozycji 3 (czyli od cyfry '3') napisu złożonego z dwóch pierwszych znaków C-napisu p (czyli napisu 'ab'):

```
string s("abc345abcAB");
const char* p("abcdef");
string::size_type i = s.find(p,3,2);
cout << s.substr(i-1,5) << endl;
```

Fragment wypisuje '5abcA'.

size_type find_first_of(/* args */)
size_type find_last_of(/* args */)
size_type find_first_not_of(/* args */)
size_type find_last_not_of(/* args */) — mają typ wartości i argumentów takie same jak odpowiednie metody **find** i **rfind**. Pierwsze dwie metody szukają, poczynając od pozycji start, pierwszego wystąpienia jakiegokolwiek znaku *należącego* do napisu określonego przez pozostałe argumenty. Kierunek przeszukiwania dla metod z **_last_** w nazwie jest od pozycji start wstecz, a dla metod z **_first_** w nazwie — do przodu. Metody z drugiej pary, te z **_not_** w nazwie, działają podobnie, ale szukają wystąpienia znaku *nie* należącego do napisu określonego przez pozostałe argumenty. Jeśli odpowiedni znak został znaleziony, zwracana jest jego pozycja; jeśli nie, zwracane jest npos. Na przykład

```
1  string s("abc123.,!");
2  const char* p = "!.,?:1234";
3  string::size_type i = s.find_first_of(p);
4  string::size_type k = s.find_last_not_of(p,s.size()-1,5);
5  string s1(s.begin()+i,s.begin()+k+1);
6  cout << i << " " << k << " " << s1 << endl;
```

wypisze '3 5 123'. W linii czwartej jako drugi argument podaliśmy `s.size()-1`, bo przeszukiwanie odbywa się do tyłu, a więc zacząć trzeba od znaku ostatniego, a nie od pierwszego. Uwzględniamy przy tym tylko 5 pierwszych znaków ze wzorca p, a więc znaki '`!.,?:`'. W linii piątej tworzymy nowy obiekt klasy **string** i inicjujemy go wycinkiem napisu s. W drugim argumencie dodajemy do `s.begin()+k`

jedynkę, gdyż wycinek zawiera znaki tylko do *poprzedzającego* ten wskazywany przez drugi iterator.

int compare(const string& wzor)
int compare(size_type start, size_type ile, const string& wzor)
int compare(size_type start, size_type ile, const string& wzor, size_type s, size_type ile1)
int compare(const char* p)
int compare(size_type start, size_type ile, const char* p, size_type i = npos) — porównują napis lub jego podciąg określony przez start i ile z napisem wyznaczonym przez pozostałe argumenty. Wynikiem jest -1, jeśli napis porównywany jest leksykograficznie wcześniejszy od napisu podanego jako argument, zero, jeśli są identyczne, a $+1$, jeśli jest leksykograficznie późniejszy.

void push_back(char c) — wstawia na koniec napisu znak c — `s.push_back(c)` działa jak wywołanie `s.insert(s.end(),c)`, tyle że jest bezrezultatowe (metoda **insert** zwraca przy takim wywołaniu iterator). Ten sam efekt można uzyskać za pomocą metody **append** lub operatora '+='.

const char* c_str( ) — zwraca wskaźnik do stałego C-napisu złożonego ze znaków napisu C++, na rzecz którego była wywołana. C-napis kończy się znakiem '\0', a zatem może być argumentem funkcji operujących na zwykłych C-napisach. Pamiętać tylko należy, że zwracany wskaźnik jest typu **const**, a więc w razie konieczności modyfikacji uzyskany C-napis trzeba przekopiować do zwykłej, modyfikowalnej tablicy znaków.

iterator begin( ) — zwraca iterator (uogólniony wskaźnik, rozdz. 24.1.2, str. 522) wskazujący na pierwszy znak napisu.

iterator end( ) — zwraca iterator wskazujący na znak pierwszy *za* ostatnim napisu.

reverse_iterator rbegin( )
reverse_iterator rend( ) — zwraca iterator „odwrotny" wskazujący na początek i koniec napisu w odwrotnym porządku. Na przykład po

```
string s1("korab");
string s2(s1.rbegin(),s1.rend());
```

s2 będzie zawierać napis 'barok'.

Prócz metod klasy **string** biblioteka włączana za pomocą pliku nagłówkowego ***string*** dostarcza bardzo przydatną funkcję (a więc *nie* metodę klasy):

istream& getline(istream& str, string& s)
istream& getline(istream& str, string& s, char eol) — wczytuje ze strumienia str jedną linię tekstu i wstawia ją do napisu s. Linia może zawierać białe znaki (prócz znaku końca linii). Znak końca linii jest wyjmowany ze strumienia, ale nie jest włączany do wynikowego napisu. Znak, który ma pełnić rolę znaku końca linii, można podać jako trzeci argument funkcji — domyślnie jest nim '\n'. Funkcja zwraca referencję do strumienia str, tak więc nieco dziwna konstrukcja

```
string s1,s2;
getline(cin,s1) >> s2;
```

zadziała i wpisze pierwszy wczytany wiersz do napisu `s1`, a pierwsze słowo następnego wiersza do napisu `s2`.

— 18 —

Wzorce

Mechanizm **wzorców** (zwanych też **szablonami**) to cecha charakterystyczna języka C++. Szablony pozwalają na abstrakcyjne definiowanie funkcji i klas w terminach parametrów, którymi są *typy danych*. Na podstawie szablonu kompilator może sam wygenerować wiele definicji konkretnych funkcji i klas wtedy, kiedy będą potrzebne. W ten sposób działa standardowa biblioteka C++, która tak naprawdę składa się w dużej mierze właśnie z szablonów — dlatego jest znana pod tradycyjną nazwą STL: *Standard Template Library*.

Choć ogólna idea szablonów jest jedna, szczegóły zależą od tego czy mamy do czynienia z funkcjami czy z klasami: szablony funkcji rozpatrywaliśmy już w rozdz. 11.14 na stronie 200; tu zatem powiemu o szablonach klas.

Podrozdziały:

18.1 Szablony klas

Na podobnej zasadzie co szablony funkcji można tworzyć szablony całych klas, wraz z konstruktorami, destruktorami, polami i metodami. Składnia jest analogiczna:

```
template <typename T, typename M>
class Klasa {
    // tu uzywamy typow T i M
};
```

definiuje szablon klasy **Klasa** parametryzowany dwoma typami. Jak teraz utworzyć obiekt klasy wygenerowanej z tego wzorca dla konkretnych typów? Nie możemy po prostu napisać

```
Klasa x;
```

bo kompilator nie wiedziałby, jakie typy przypisać parametrom **T** i **M** w szablonie. Musimy zatem zażądać jawnie utworzenia na podstawie szablonu i skompilowania konkretnej klasy. Robimy to, podobnie jak dla funkcji, podając nazwy konkretnych typów (klasowych lub wbudowanych) jako argumenty dla wzorca, czyli w nawiasach kątowych. O ile dla funkcji mogliśmy tak zrobić, ale często nie musieliśmy, bo na podstawie typów argumentów wywołania kompilator sam mógł wydedukować potrzebne typy dla parametrów szablonu, o tyle dla klas musimy to robić jawnie, na przykład:

```
Klasa<double,int> x;
```

W ten sposób zażądaliśmy wygenerowania kodu klasy na podstawie szablonu **Klasa** poprzez zamianę wszystkich wystąpień parametru **T** na **double**, a wystąpień parametru **M** na **int**. Utworzona klasa ma nazwę **Klasa<double,int>**. W dalszym ciągu programu możemy tej nazwy używać: klasa została już wygenerowana i przy następnych pojawieniach się tej nazwy żadna nowa klasa nie będzie już tworzona. Jeśli natomiast pojawi się nazwa szablonu, ale z innymi typami

```
Klasa<int,Osoba> z;
```

to oczywiście utworzona zostanie nowa klasa, nie mająca nic wspólnego z poprzednią (prócz tego, że obie zostały wygenerowane z tego samego szablonu); jej nazwą będzie **Klasa<int,Osoba>**.

Parametrem wzorca klasy może też być wartość określonego typu. Na przykład

```
template <typename T, int size>
class Klasa {
    // w definicji uzywamy typu 'T'
    // i wartosci calkowitej 'size'
};
```

Konkretną wersję takiej klasy otrzymamy na przykład definiując obiekt

```
Klasa<Osoba,100> t;
```

Nazwą tej klasy będzie oczywiście **Klasa<Osoba,100>**. Zwróćmy uwagę, że podając inny argument szablonu (na przykład 150 zamiast 100), otrzymalibyśmy inną, całkowicie niezależną klasę.

Pewien kłopot sprawia czasem definiowanie poza szablonem klasy metod w nim zadeklarowanych. Definiując taką metodę trzeba, jak pamiętamy, podać jej nazwę kwalifikowaną. Jeśli jest to szablon, to jako nazwę kwalifikowaną klasy podajemy nazwę szablonu z nazwami parametrów szablonu, już bez słowa kluczowego **class** lub **typename**:

```
1   template <typename T, int size>
2   class Klasa {
3       void metoda1() {
4           // definicja metody1
5       }
6       T* metoda2(double); // tylko deklaracja
7
8       // ...
9   };
10
11  //
12  // ...
```

```
13      //
14
15      // definicja metody metoda2
16      template <typename T, int size>
17      T* Klasa<T,size>::metoda2(double x) {
18          // cialo definicji
19      }
```

W tym przykładzie metoda **metoda1** jest zdefiniowana bezpośrednio wewnątrz szablonu, a metoda **metoda2** poza nim. W podobny sposób można poza klasą definiować konstruktory i destruktory.

Rozpatrzmy bardziej praktyczny przykład wzorca klasy opisującej stos (ang. *stack*) implementowany za pomocą tablicy (dla zachowania przejrzystości bez obsługi błędów). Musimy zatem zaimplementować metody pozwalające na:

- dodanie elementu na wierzchołek (**push**);
- zdjęcie i zwrócenie elementu z wierzchołka (**pop**);
- sprawdzanie, czy stos jest pusty (**empty**).

Konkretyzacje szablonu **Stos** będą różnić się tylko typem elementów kładzionych na stos i wymiarem alokowanej na elementy stosu tablicy:

P148: *stos.cpp* Szablon stosu realizowanego w postaci tablicy

```
1  #include <iostream>
2  #include <string>
3  #include <typeinfo>
4  using namespace std;
5
6  template <typename Data, int size>
7  class Stos {
8      Data* data;
9      int   top;
10 public:
11     Stos();
12     bool empty() const;
13     void push(Data);
14     Data pop();
15     ~Stos();
16 };
17
18 template <typename Data, int size>
19 Stos<Data,size>::Stos() {
20     data = new Data[size];
21     top = 0;
22 }
```

```
23
24 template <typename Data, int size>
25 inline bool Stos<Data,size>::empty() const {
26     return top == 0;
27 }
28
29 template <typename Data, int size>
30 inline void Stos<Data,size>::push(Data dat) {
31     data[top++] = dat;
32 }
33
34 template <typename Data, int size>
35 inline Data Stos<Data,size>::pop() {
36     return data[--top];
37 }
38
39 template <typename Data, int size>
40 inline Stos<Data,size>::~Stos() {
41     delete [] data;
42 }
43
44   // szablon funkcji globalnej
45 template <typename Data, int size>
46 void oproznij(Stos<Data,size>* p_stos) {
47     cout << "Stos typu " << typeid(Data).name() << ": ";
48     while ( ! p_stos->empty() ) {
49         cout << p_stos->pop() << " ";
50     }
51     cout << endl;
52 }
53
54 int main() {
55     Stos<int,20> stos_i;
56     stos_i.push(11);
57     stos_i.push(36);
58     stos_i.push(49);
59     stos_i.push(92);
60
61     Stos<string,15> stos_s;
62     stos_s.push("Ala");
63     stos_s.push("Ela");
64     stos_s.push("Ola");
65     stos_s.push("Ula");
66
67     oproznij(&stos_i);
68     oproznij(&stos_s);
```

```
69 }
```

Parametrami szablonu **Stos** są typ danych **Data** i liczba całkowita size określająca maksymalny wymiar stosu. Dla przejrzystości opuściliśmy tu obsługę błędów.

Wewnątrz szablonu deklarujemy konstruktor, destruktor i potrzebne metody. W przypadku tak prostym można je było zdefiniować bezpośrednio wewnątrz szablonu, ale w celach dydaktycznych ich definicje następują poza szablonem klasy (linie 18-42). Ponieważ definicje są poza klasą, a funkcje są bardzo proste, kompilator prawdopodobnie będzie umiał je rozwinąć (patrz rozdz. 11.10, str. 179). Dlatego w definicjach szablonów metod użyliśmy modyfikatora **inline**.

Dla stosu zaimplementowanego za pomocą tablicy o ustalonym rozmiarze powinniśmy jeszcze pomyśleć o obsłudze błędów, jakie mogą pojawić się, gdy próbujemy położyć na pełnym już stosie dodatkowy element lub zdjąć element ze stosu pustego. Dla zachowania przejrzystości w tym programie tego już nie robimy.

W liniach 44-52 definiujemy wzorzec funkcji globalnych **oproznij** służących do zdjęcia po kolei wszystkich elementów stosu i wydrukowania ich. Parametrem tych funkcji będzie wskaźnik do stosu typu określonego przez pewną klasę konkretną uzyskaną z szablonu **Stos**. Funkcja korzysta znów z operatora **typeid** do wyświetlenia nazwy typu argumentu (wyłącznie w celach dydaktycznych).

W funkcji **main** definiujemy dwa stosy: stos stos_i liczb całkowitych o maksymalnym rozmiarze 20 oraz stos napisów, stos_s, o maksymalnym wymiarze 15. Kładziemy na każdy z nich po parę elementów, a następnie wywołujemy funkcję **oproznij** (linie 67 i 68). Wydruk jest następujący

```
Stos typu i: 92 49 36 11
Stos typu Ss: Ula Ola Ela Ala
```

Wewnętrzne nazwy typów (w naszym przykładzie były to **i** dla **int** i **Ss** dla **string**) jak zwykle mogą zależeć od kompilatora.

Zauważmy, że wywołując funkcję **oproznij** nie podaliśmy argumentów szablonu. Równie dobrze moglibyśmy napisać

```
oproznij<int,20>(&stos_i);
oproznij<string,15>(&stos_s);
```

Widzimy jednak, że kompilator nie potrzebował tej podpowiedzi, aby ustalić, na podstawie typu argumentu wywołania, odpowiedni typ i wymiar potrzebny do konkretyzacji szablonu funkcji **oproznij**.

Zwróćmy jeszcze uwagę na wspomniany już uprzednio fakt: do definicji takiego szablonu klasy jak nasz przykładowy **Stos** należy nie tylko typ danych (u nas **Data**), ale i rozmiar (u nas size). Wobec tego klasy **Stos<int,10>** i **Stos<int,11>** byłyby dwiema zupełnie różnymi klasami, definiującymi dwa całkowicie odrębne typy danych.

— 19 —

Przeciążanie operatorów. Semantyka przenoszenia i inteligentne wskaźniki

Przeciążanie operatorów to sposób na nadanie znaczenia takich operatorów, jak '+', '&' czy '<<' w sytuacji, gdy ich argumentami są obiekty klas przez nas definiowanych. Przeciążanie nie jest zwykle absolutnie konieczne, choć bywa bardzo wygodne (zamiast przeciążać operatory, można definiować zwykłe funkcje i metody wykonujące te same zadania). Na przykład w C czy w Javie użytkownik nie może przeciążać operatorów, a nie można przecież powiedzieć, że z tego powodu są to języki ubogie. Przeciążanie operatorów występuje natomiast w Pythonie, Haskellu czy C#.

Z przeciążaniem operatorów jest też związana semantyka przenoszenia, którą zajmiemy się w ostatniej części tego rozdziału.

Podrozdziały:

19.1 Wstęp

Użycie w tekście programu operatorów (takich jak '+', '->', '&&' itd.) powoduje tak naprawdę wywołanie funkcji z argumentami będącymi wartościami wyrażeń stojących

po obu ich stronach (lub po jednej, dla operatorów jednoargumentowych). Tak więc podczas opracowania instrukcji

```
c = a + b;
```

wywołana zostanie funkcja mająca na celu dodanie a i b. Jaka to będzie funkcja, zależy od typu a i b; jeśli oba argumenty są typu **double**, to wywołana zostanie inna funkcja niż ta służąca do dodania dwóch liczb typu **int**. Jeśli a jest typu **int**, a b typu **double**, to, jak wiemy, najpierw wartość zmiennej a (ale nie sama zmienna!) zostanie skonwertowana do typu **double** i następnie zostanie wywołana funkcja dodająca wartości typu **double** (i wynik będzie też tego typu). Automatyczne konwersje i wybór odpowiedniej funkcji będą przebiegać „samoczynnie" dla typów wbudowanych. Dla argumentów będących obiektami klas przez nas samych zdefiniowanych, znaczenie tych operatorów jest jednak niezdefiniowane. Zatem, jeśli chcemy korzystać z tych operatorów w takim kontekście, musimy je sami określić.

Możemy to zrobić określając (w specjalny sposób) w definiowanej przez nas klasie odpowiednie metody lub definiując przeznaczone do tego celu funkcje globalne, najczęściej, choć nie jest to konieczne, zaprzyjaźnione z naszą klasą.

Taką operację nazywamy **przeciążaniem** lub **przeładowaniem** operatora, w analogi do przeciążania funkcji, czyli definiowania kilku funkcji o tej samej nazwie. Tak jak dla funkcji, właściwa wersja operatora znajdowana jest przez kompilator na podstawie typu argumentów.

Operatory, które mogą być przeciążone (przeładowane), to:

Tablica 19.1: Operatory które mogą być przeciążone

+	−	*	/	%	^	&	|	~	!
=	<	>	+=	−=	*=	/=	%=	^=	&=
|=	≪	≫	≪=	≫=	==	!=	<=	>=	&&
||	++	−−	,	−>*	−>	new	delete	()	[]

przy czym operatory '&', '*', '−', '+' mogą być przeładowane w obu wersjach: jedno- i dwuargumentowej. Jednoargumentowe operatory zwiększenia i zmniejszenia, '++' i '−−', można przeładowywać z kolei w obu ich odmianach: przyrostkowej i przedrostkowej. Operatory '=', '->','()' i '[]' są w pewien sposób specjalne i nimi zajmiemy się osobno. Przeciążać można też operatory '**new**' i '**delete**', ale jest to kwestia dość delikatna i często zależna od implementacji; nie będziemy się zatem nią zajmować.

Nie można natomiast przeciążać operatorów '.', '.*', '::', '?:' i **sizeof**.

Niektóre operatory są dostarczane przez system dla (prawie) każdej klasy, nawet jeśli sami nie przeciążyliśmy ich w żaden sposób. Należą do nich operatory '&' (pobrania adresu), '=' (przypisania), ',' (przecinek) oraz **new** i **delete**. Pierwszego lepiej nie przeciążać, ostatnich trzech zwykle nie ma potrzeby.

Jakkolwiek nie przeciążalibyśmy operatorów, ich priorytety i sposób wiązania (od lewej do prawej lub od prawej do lewej, patrz rozdz. 9.1 na stronie 121) pozostają takie same jak dla standardowych operatorów oznaczonych tym samym symbolem. Tak więc w wyrażeniu

```
a = b + c * d;
```

operacja przypisana symbolowi '$*$' zastosowana zostanie przed operacją przypisaną symbolowi '+', niezależnie od tego, jak zdefiniowane są w tym kontekście te dwie operacje. Podobnie, niezależnie od tego, jak przedefiniujemy operator przypisania '=', wyrażenie

```
a = b = c;
```

jest równoważne wyrażeniu

```
a = ( b = c );
```

a nie

```
( a = b ) = c;
```

bo operator przypisania wiąże od prawej.

Również liczba argumentów przeciążonego operatora nie jest dowolna: istnieje na przykład tylko jedna standardowa wersja operatora '%' (reszta z dzielenia), w której operator ten jest dwuargumentowy. Nie można zatem przeciążyć tego operatora tak, aby był jednoargumentowy. A więc operatory dwuargumentowe w wersji przeładowanej muszą być też dwuargumentowe, a jednoargumentowe — jednoargumentowe.

Przeładowując operatory należy starać się stosować zasadę *of least astonishment* — najmniejszego zaskoczenia — tzn. tak przeładowywać operatory, by można było łatwo przewidzieć ich zachowanie na podstawie analogii z normalnym znaczeniem danego operatora. Na przykład, jeśli przeładowywujemy operator dodawania, to zwracany wynik powinien być typu tego samego co argumenty i powinien *nie* być l-wartością (a zatem na przykład *nie* powinien zwracać wyniku przez referencję). Dodawanie bowiem nie powinno „dać się postawić" po lewej stronie przypisania.

Podobnie, jeśli przeładowaliśmy operator '+', to powinniśmy przeciążyć również operator '+=' i pomyśleć, czy nie byłoby naturalne przeładować też odejmowanie ('−'), odejmowanie z przypisaniem ('−=') i/lub operatory zmniejszenia i zwiększenia.

Z drugiej strony, gdybyśmy definiowali klasę **String** naśladującą klasę **String** lub **StringBuilder** z Javy, naturalne byłoby zastąpienie funkcji scalającej napisy (konkatenacji) przeładowanym operatorem dodawania: w tym przykładzie nie byłoby natomiast intuicyjnie oczywistej interpretacji odejmowania czy zmniejszania.

19.2 Przeciążenia za pomocą funkcji globalnych

Operator przeciążać można za pomocą zdefiniowanych globalnie funkcji. Definicja taka wygląda jak normalna definicja funkcji: specjalna jest tylko nazwa tej funkcji. Rozpatrzmy osobno przeciążanie operatorów dwu- i jedno argumentowych.

19.2.1 Operatory dwuargumentowe

Globalną funkcję przeładowującą operator dwuargumentowy definiujemy jako funkcję dwóch argumentów, z których co najmniej jeden musi być typu zdefiniowanego w programie (a nie typu wbudowanego). Nazwą tej funkcji musi być '`operator`', gdzie '' jest symbolem operatora, a więc jednym z symboli wymienionych w tabeli operatorów (str. 386) będącym symbolem któregoś z operatorów dwuargumentowych, jak na przykład symbolem '+' lub '<<'. Zauważmy, że nazwa tej funkcji jest specjalna: po pierwsze słowo '**operator**' jest tu słowem kluczowym i musi być tak właśnie napisane, a po drugie symbol nie będący literą, cyfrą lub znakiem podkreślenia normalnie nie byłby dozwolony w identyfikatorze (nazwie) funkcji.

Tak więc deklaracja takiej funkcji ma postać

```
Typ operator@(Typ1, Typ2);
```

gdzie co najmniej jeden z typów parametrów jest zdefiniowany przez nas w programie (nie jest typem wbudowanym), a więc jego nazwa jest nazwą zdefiniowanej przez nas klasy/struktury. Typ **Typ** wartości zwracanej może być dowolny. Zamiast symbolu '' powinien oczywiście być użyty symbol tego operatora, który chcemy przeciążyć. Funkcja ta często powinna być zaprzyjaźniona z naszą klasą, jeśli chcemy, aby miała dostęp do jej prywatnych lub chronionych składowych. Nie jest to jednak konieczne.

Tak zdefiniowaną funkcję można wywołać jawnie, „po nazwie", ale zwykle się tego nie robi, bo cały sens przeładowywania operatorów polega na tym, by używać tych operatorów zamiast jawnego wywoływania funkcji. Funkcja ta wywołana będzie „samoczynnie", gdy napotkane będzie wyrażenie

```
a @ b
```

i zarówno symbol '', jak i typy zmiennych a i b będą odpowiadać deklaracji/definicji funkcji. Wyrażenie to jest zatem równoważne jawnemu wywołaniu naszej funkcji:

```
operator@(a,b)
```

W poniższym przykładzie definiujemy bardzo prostą klasę **Modulo**:

P149: ***modsev.cpp*** Przeciążenie operatora dodawania

```
1 #include <iostream>
2 using namespace std;
3
4 struct Modulo {
5     static const int modul;
6
7     int numb;
8
9     Modulo() : numb(0)
```

```
10      { }
11
12      Modulo(int numb) : numb(numb%modul)
13      { }
14 };
15 const int Modulo::modul = 7;
16
17 Modulo operator+(Modulo m, Modulo n) {
18      return Modulo(m.numb + n.numb);
19 }
20
21 int main() {
22      Modulo m(5), n(6), k;
23
24      k = m + n;
25      cout << m.numb << " + " << n.numb
26           << " (mod " << Modulo::modul
27           << ") = " << k.numb << endl;
28
29      k = operator+(m,n);
30      cout << m.numb << " + " << n.numb
31           << " (mod " << Modulo::modul
32           << ") = " << k.numb << endl;
33 }
```

Obiekty tej klasy reprezentują liczby całkowite modulo 7 (każda liczba jest reprezentowana resztą, jaką daje przy dzieleniu przez 7; na przykład 11 i 32 z tego punktu widzenia są równe, bo przy dzieleniu przez 7 dają resztę 4). Istnieje zatem tylko siedem różnych liczb tej klasy $[0, \ldots, 6]$. Jeśli dodajemy tego rodzaju liczby, to wynik też powinien mieć taką postać; na przykład

$$5 + 6 \equiv 4 \pmod 7$$

Przedefiniowujemy więc dodawanie liczb typu **Modulo**. W liniach 17-19 definiujemy funkcję o nazwie **operator**+, z parametrami typu **Modulo**. Funkcja ta będzie wywołana, jeśli w programie pojawi się operator '+', a po obu jego stronach będą wyrażenia o wartościach typu **Modulo**. Tak jest w linii 24. Zatem wywołana zostanie nasza funkcja i zwróci obiekt klasy **Modulo** reprezentujący sumę argumentów, bo tak została zdefiniowana.

W linii 29 widzimy, że jeśli się uprzemy, to możemy naszą funkcję wywołać normalnie, „po nazwie". W obu przypadkach dostaniemy ten sam rezultat

```
5 + 6 (mod 7) = 4
5 + 6 (mod 7) = 4
```

W przykładzie tym przeładowanie odejmowania, zwiększenia czy mnożenia byłoby jak najbardziej naturalne; nie zrobiliśmy tego, aby przykład był krótszy. Zauważmy, że

typem zwracanym przy dodawaniu jest **Modulo**, a więc rezultat jest zwracany przez wartość i *nie* jest l-wartością (w zgodzie ze zwykłym rozumieniem dodawania liczb typów wbudowanych).

W podobny sposób można przeładować inne operatory dwuargumentowe, jak operator dzielenia, reszty z dzielenia, odejmowania itd.

Jednym z najczęściej przeciążanych operatorów jest operator '<<'. Jaki jest typ argumentów tego operatora? Po lewej stronie mamy identyfikator obiektu klasy **ostream** reprezentującego strumień wyjściowy. Po prawej zaś obiekt typu **int**, **double** itd. W tych przypadkach kompilator znajdzie odpowiednią funkcję, bo dla typów wbudowanych są one częścią biblioteki. Jeśli prawym argumentem będzie obiekt typu przez nas zdefiniowanego, to odpowiedniej funkcji nie będzie i wystąpi błąd ... chyba że taką funkcję dostarczymy. Jaki powinien być jej typ zwracany? Jeśli mamy zamiar pisać tylko

```
cout << zzz;
```

gdzie zzz jest identyfikatorem obiektu naszej klasy, to w zasadzie typ zwracany nie ma znaczenia. Ale jeśli chcielibyśmy używać operatora '<<' kaskadowo

```
cout << zzz << " " << yyy;
```

to wartość wyrażenia '`cout << zzz`' powinna być referencją do obiektu strumienia, tego samego, który stoi po lewej stronie operatora.

Tak więc, jeśli chcemy „nauczyć" program, jak wypisywać obiekty naszej klasy **Klasa** do strumienia wyjściowego (związanego z ekranem komputera lub plikiem) za pomocą operatora '<<', powinniśmy zdefiniować funkcję o prototypie

```
ostream& operator<<(ostream&, const Klasa&);
```

Pierwszy parametr, typu **ostream&**, nie może być ustalony (**const**), bo podczas zapisu stan obiektu strumieniowego zmienia się. Nie może też być typu **ostream**, bo przekazywanie argumentu przez wartość wymagałoby kopiowania, a konstruktor kopiujący w klasie **ostream** jest prywatny; z tego samego powodu rezultat musi być zwracany przez referencję a nie przez wartość.

Obiekt klasy **Klasa** (drugi parametr) może być przesłany przez wartość lub referencję. W pierwszym przypadku trzeba zastanowić się, czy potrzebny jest odpowiedni konstruktor kopiujący, aby przekazanie przez wartość (związane z odkładaniem *kopii* na stosie) miało sens. W drugim przypadku typ parametru nie musi wprawdzie, ale może i najczęściej powinien, być ustalony, bo funkcja będzie miała dostęp do oryginału, a jej zadaniem jest przecież zwykle wypisanie informacji o obiekcie, a nie jego zmiana.

Jeśli przy wypisywaniu informacji o obiekcie potrzebny jest dostęp do prywatnych lub chronionych składowych, to funkcję tę należy też zadeklarować jako zaprzyjaźnioną wewnątrz klasy:

```
class Klasa {
    // ...
    friend ostream& operator<<(ostream&, const Klasa&);
}
```

Funkcja powinna zwracać, poprzez wykonanie instrukcji **return**, swój pierwszy argument, czyli referencję do obiektu reprezentującego strumień wyjściowy, do którego nastąpił zapis (nie musi to być cout; może to być dowolny obiekt klasy **ostream** lub klasy z niej dziedziczącej).

Rozpatrzmy zatem przykład:

P150: *opwyj.cpp* Przeciążenie operatora wstawiania do strumienia

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class Osoba {
6      string nazw;
7      int    wiek;
8  public:
9      Osoba(const string& nazw, int wiek)
10         : nazw(nazw), wiek(wiek)
11     { }
12
13      // ... inne skladowe
14
15     friend ostream& operator<<(ostream&, const Osoba&);
16 };
17
18 ostream& operator<<(ostream& str, const Osoba& k) {
19     return str << k.nazw << " (" << k.wiek << " lat)";
20 }
21
22 int main() {
23     Osoba t[] = {  Osoba("Ola",18), Osoba("Ula",26),
24                    Osoba("Ala",35), Osoba("Ela",11)  };
25
26     for (int i = 0; i < 4; i++)
27         cout << t[i] << endl;
28 }
```

W linii 15, wewnątrz definicji klasy **Osoba**, deklarujemy zaprzyjaźnioną funkcję przeciążającą operator '<<' dla obiektów tej klasy. Dzięki temu funkcja ta, *nie* będąc składową klasy, będzie miała bezpośredni dostęp do prywatnych składowych nazw i wiek, o czym przekonuje nas wydruk z tego programu

```
Ola (18 lat)
Ula (26 lat)
Ala (35 lat)
Ela (11 lat)
```

Sama funkcja przeciążająca jest zdefiniowana w liniach 18-20 zgodnie z zasadami, o jakich mówiliśmy wyżej. Jest to funkcja globalna, nie będąca składową klasy, więc oczywiście nie jest wywoływana na rzecz obiektu: obiekt, o który nam chodzi, jest przesyłany przez argument (jest to obiekt występujący po prawej stronie operatora '<<'). Wewnątrz funkcji nie istnieje wobec tego wskaźnik **this**. Zauważmy, że rezulatatem całego wyrażenia w instrukcji **return** jest obiekt-strumień str, który właśnie powinien zostać zwrócony: dlatego cała treść funkcji mieści się tu w tej jednej instrukcji **return**.

Operator dwuargumentowy możemy przeciążyć tak, aby *oba* jego argumenty były typu zdefiniowanego przez nas, przy czym każdy z nich może być innego typu. W poniższym przykładzie operator dodawania przeciążony jest tak, aby móc dodawać wektory (obiekty klasy **Vector**) do wektorów — wynikiem jest wtedy wektor — i wektory do punktów: wynikiem jest wtedy punkt. W tym celu dostarczamy *dwóch* definicji przeciążonego operatora dodawania: właściwy zostanie wybrany przez kompilator na podstawie typów argumentów dodawania (linie 50 i 53):

P151: *vecpoin.cpp* Przeciążanie operatorów binarnych

```
1  #include <iostream>
2  using namespace std;
3
4  class Point;
5
6  class Vector {
7      double x, y, z;
8  public:
9      Vector(double x = 0, double y = 0, double z = 0)
10         : x(x), y(y), z(z)
11     { }
12     friend Point    operator+(const Point&,  const Vector&);
13     friend Vector   operator+(const Vector&, const Vector&);
14     friend ostream& operator<<(ostream&, const Vector&);
15 };
16
17 class Point {
18     double x, y, z;
19 public:
20     Point(double x = 0, double y = 0, double z = 0)
21         : x(x), y(y), z(z)
22     { }
23     friend Point operator+(const Point&, const Vector&);
```

```
24      friend ostream& operator<<(ostream&, const Point&);
25  };
26
27  Point operator+(const Point&  p, const Vector& v) {
28      return Point(p.x+v.x, p.y+v.y, p.z+v.z);
29  }
30
31  Vector operator+(const Vector& v1, const Vector& v2) {
32      return Vector(v1.x+v2.x, v1.y+v2.y, v1.z+v2.z);
33  }
34
35  ostream& operator<<(ostream& str, const Point& p) {
36      return str << "P(" << p.x << "," << p.y
37                 << ","  << p.z << ")";
38  }
39
40  ostream& operator<<(ostream& str, const Vector& v) {
41      return str << "V[" << v.x << "," << v.y
42                 << ","  << v.z << "]";
43  }
44
45  int main() {
46
47      Vector v1(1,1,1), v2(2,2,2);
48      Point  p1(1,2,3);
49
50      Vector v = v1 + v2;
51      cout << "v: " << v << endl;
52
53      Point  p = p1 + v;
54      cout << "p: " << p << endl;
55  }
```

Ponieważ funkcje przeciążające operatory dodawania i wstawiania do strumienia korzystają z bezpośredniego dostępu do prywatnych składowych klas **Vector** i **Point**, zostały w definicji klas zadeklarowane jako zaprzyjaźnione. Zauważmy, że deklaracja zapowiadająca w linii 4 jest tu konieczna, bo w definicji klasy **Vector** jest używana nazwa **Point** i odwrotnie. Obie funkcje przeciążające dodawanie zwracają wynik przez wartość; w ten sposób zapewniamy, że wynik *nie* jest l-wartością: jest to zachowanie, jakiego właśnie spodziewamy się po dodawaniu.

19.2.2 Operatory jednoargumentowe

Analogicznie można przeciążać operatory jednoargumentowe: funkcja (globalna) o prototypie

```
Typ operator@(Typarg);
```

definiuje przeciążenie funkcji o jednym argumencie (którym musi być obiekt klasy zdefiniowanej w programie). Symbol '' musi zatem odpowiadać symbolowi któregoś z operatorów jednoargumentowych, na przykład '&' czy '!'. Wyrażenie

```
@a
```

gdzie a jest identyfikatorem obiektu (lub referencji do obiektu) naszej klasy, będzie wtedy równoważne wywołaniu

```
operator@(a)
```

W poniższym programie przeciążamy jednoargumentowy operator '!'. Klasa **AClass** ma pole typu napisowego i jak widzimy z definicji funkcji w liniach 13-15, operator '!' będzie zwracał wartość typu logicznego równą **true**, jeśli napis ten jest dłuższy niż pięcioznakowy.

P152: *oneargop.cpp* Przeciążenie operatora jednoargumentowego

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  struct AClass {
6      string name;
7
8      AClass(const string& name)
9          : name(name)
10     { }
11 };
12
13 bool operator!(const AClass& c) {
14     return c.name.size() > 5;
15 }
16
17 int main() {
18     AClass t[] = { AClass("Marlon"), AClass("Henry"),
19                    AClass("Dave"),   AClass("Horatio"),
20                    AClass("Sue"),    AClass("Alice")   };
21
22     for (int i = 0; i < 6; ++i)
23         if ( !t[i] ) cout << t[i].name << endl;
24 }
```

W linii 23 drukujemy imiona odpowiadające poszczególnym obiektom z tablicy t; zastosowanie operatora '!' powoduje, że wydrukowane zostaną tylko te imiona, które liczą ponad pięć liter:

```
Marlon
Horatio
```

19.3 Przeciążenia za pomocą metod klasy

Operatory, tak dwu- jak i jednoargumentowe, można też przeciążać za pomocą metod klasy. W tym jednak przypadku, ponieważ są to metody, jeden z argumentów będzie przesłany niejawnie: będzie nim ten obiekt, a dokładniej wskaźnik do tego obiektu, na rzecz którego metoda została wywołana; wewnątrz metody możemy odnosić się do niego poprzez wskaźnik **this**. Tak więc operatory dwuargumentowe będziemy definiować jako metody jednoparametrowe, a operatory jednoargumentowe jako metody bezparametrowe.

Niektóre operatory są nieco „specjalne" i *muszą* być przeciążane jako metody, a nie jako funkcje globalne. Należą do nich operatory:

- przypisania ('=') — dwuargumentowy, a więc przeciążany za pomocą metody jednoparametrowej; to samo dotyczy wariantów przypisania '+=', '*=' itd.;
- wywołania ('()') — jako jedyny operator może mieć dowolną liczbę parametrów;
- indeksowania ('[]') — dwuargumentowy, a więc przeciążany za pomocą metody jednoparametrowej;
- wyboru składowej przez wskaźnik ('->') — jednoargumentowy, a więc przeciążany za pomocą metody bezparametrowej.

Tymi operatorami zajmiemy się zatem osobno w podrozdziale 19.4.

19.3.1 Operatory dwuargumentowe

Przy definiowaniu operatora (funkcji operatorowej) dwuargumentowego za pomocą metody nie podaje się pierwszego parametru: niejawnie będzie nim wskaźnik do obiektu, na rzecz którego funkcja będzie wywołana. Obiektem tym będzie zawsze ten, który znajduje się po *lewej* stronie operatora. Obiekt po prawej będzie argumentem metody.

Załóżmy, że w klasie **Klasa** zdefiniowaliśmy metodę o prototypie

```
Typ operator@(Typarg);
```

lub

```
Typ operator@(Typarg&);
```

Wtedy, jeśli a jest identyfikatorem obiektu klasy **Klasa**, a b jest typu **Typarg**, wyrażenie

```
a @ b
```

będzie równoważne wywołaniu metody na rzecz obiektu a

```
a.operator@(b)
```

i przesłaniu (przez wartość lub referencję) b jako argumentu. Symbol '' oznacza tu któryś z symboli operatorów dwuargumentowych wymienionych w tabeli operatorów (str. 386).

Rozpatrzmy przykład klasy **Modulo**, której jedną wersję poznaliśmy w programie ***modsev.cpp*** (str. 388). Operator dodawania dwóch obiektów klasy **Modulo** przeciążymy w tej nowej wersji za pomocą metody:

P153: ***modsev1.cpp*** Przeciążanie za pomocą metody

```
#include <iostream>
using namespace std;

class Modulo {
    int numb;
public:
    static const int modul;
    Modulo() : numb(0)
    { }

    Modulo(int numb) : numb(numb%modul)
    { }

    Modulo operator+(const Modulo&) const;
    friend ostream& operator<<(ostream&, const Modulo&);
};
const int Modulo::modul = 7;

inline Modulo Modulo::operator+(const Modulo& n) const {
    return Modulo(numb + n.numb);
}

ostream& operator<<(ostream& str, const Modulo& n) {
    return str << n.numb;
}

int main() {
    Modulo m(5), n(6), k;

    k = m + n;
    cout << m << " + " << n
         << " (mod " << Modulo::modul
         << ") = " << k << endl;

    k = k + 8;
    cout << "k + 8 (mod " << Modulo::modul
```

```
37            << ") = " << k << endl;
38 }
```

W liniach 19-21 przeciążamy operator dodawania (zadeklarowany wewnątrz klasy w linii 14). Metoda ma jeden parametr typu **const Modulo&**, a zatem będzie wywołana zawsze, gdy napotkane zostanie w programie wyrażenie

```
m + k
```

gdzie m i k są obiektami klasy **Modulo**. Wywołanie będzie na rzecz obiektu m, a k zostanie przesłane jako pierwszy i jedyny jawny argument metody **operator+** (przez referencję, bo tak zostało to zadeklarowane w linii 14 i następnie zdefiniowane). Tego typu sytuacja zachodzi w linii 30.

Zauważmy, że jeśli przeciążamy operator dwuargumentowy za pomocą metody, to odpowiednia funkcja zostanie wywołana, gdy po *lewej* stronie operatora pojawia się obiekt klasy. Zatem *nie* można w ten sposób przeładować operatora '<<', bo dla niego po lewej stronie mamy obiekt strumieniowy klasy **ostream** i to w niej musielibyśmy dokonać tego przeciążenia. Dlatego przeciążenia operatora '<<' dokonaliśmy za pomocą zaprzyjaźnionej funkcji globalnej (linie 23-25).

Tajemnicze zjawisko zachodzi w linii 35 ('k=k+8'). Po lewej stronie operatora dodawnia jest obiekt klasy **Modulo**, ale po prawej mamy po prostu liczbę typu **int**. Takie dodawanie byłoby „obsłużone" przez metodę w klasie **Modulo** o prototypie

```
Modulo operator+(int);
```

ale jej nie dostarczyliśmy (choć, oczywiście, moglibyśmy to zrobić). Tymczasem błędu nie ma; wynik programu to

```
5 + 6 (mod 7) = 4
k + 8 (mod 7) = 5
```

co wygląda prawidłowo, bo $4 + 8 = 12 \equiv 5 \pmod 7$. Dlaczego wszystko zadziałało? Wyjaśni się to w rozdziale o konwersjach (20). W skrócie: w takiej sytuacji kompilator postara się przekonwertować liczbę całkowitą na obiekt klasy **Modulo** i dopiero wtedy wykonać dodawania. Jak się przekonamy, taka konwersja jest możliwa dzięki istnieniu konstruktora przyjmującego jeden argument typu **int** (linie 11-12).

Jako drugi przykład rozpatrzmy klasę opisującą listę, trochę inną niż ta z programu ***simplista.c*** (str. **??**). Obiekt klasy **List** ma tylko jedną składową wskaźnikową, head, której wartość to adres pierwszego elementu listy. Same elementy tej listy są obiektami wewnętrznej klasy (struktury) **Node** — linie 6-13. Klasa ta jest zdefiniowana w sekcji prywatnej klasy **List**, a więc nie jest widoczna z zewnątrz. Obiekty klasy **Node** zawierają dane (w postaci jednej liczby typu **int**) oraz wskaźnik do następnego elementu listy.

P154: *lista.cpp* Listy z przeciążonymi operatorami

```
#include <iostream>
using namespace std;

class List {

    struct Node {
        int    elem;
        Node* next;

        Node(int elem, Node* next = 0)
            : elem(elem), next(next)
        { }
    };

    Node* head;

public:
    List()
        : head(0)
    { }

    List& operator+(int elem) {
        Node* w = new Node(elem);
        if (head) {
            Node *h = head;
            while (h->next) h = h->next;
            h->next = w;
        } else {
            head = w;
        }
        return *this;
    }

    List& operator-(int elem) {
        head = new Node(elem,head);
        return *this;
    }

    int operator!() const {
        int cnt = 0;
        for (Node* h = head; h ; h = h->next, ++cnt);
        return cnt;
    }

```

```
45      ~List() {
46          Node *prev, *curr = head;
47          while (curr) {
48              prev = curr;
49              curr = curr->next;
50              cerr << "usuwanie: " << prev->elem << endl;
51              delete prev;
52          }
53      }
54
55      friend ostream& operator<<(ostream&, const List&);
56  };
57
58  ostream& operator<<(ostream& s, const List& L) {
59          for(List::Node* h = L.head ; h ; h = h->next)
60              s << h->elem << " ";
61          return s;
62  }
63
64  int main() {
65      List list;
66
67      list + 1;
68      list + 2 - 0 - (-1);
69      cout <<  list+3  << endl;
70      cout << "List ma " << !list << " elementow" << endl;
71  }
```

Do dodawania elementów do listy użyliśmy tu przeciążenia operatorów dodawania i odejmowania. Operator '+' opisany funkcją **operator+** (linie 22-32) tworzy nowy element listy na podstawie liczby będącej prawym argumentem operatora, czyli pierwszym i jedynym argumentem metody. Utworzony element jest następnie „doczepiany" na koniec listy (kod w liniach 25-27). Zauważmy, że metoda zwraca przez referencję obiekt, na rzecz którego została wywołana ('*this'). Tak więc na przykład, jeśli list jest obiektem klasy **List**, opracowanie wyrażenia

```
list + 5
```

spowoduje:

- wywołanie metody **operator+** z argumentem 5;
- utworzenie obiektu klasy **Node** zawierającego jako składową elem liczbę 5, a jako składową next wskaźnik pusty (linia 23; wykorzystujemy tu wartość domyślną konstruktora z linii 10);
- dołączenie tego elementu na koniec listy;

- zwrócenie referencji do tak zmodyfikowanej listy.

W ten sposób wartością całego tego wyrażenia jest referencja do listy, a zatem możliwe jest kaskadowe dodanie do niej następnego elementu; wyrażenie

```
list + 5 + 7
```

to dodanie najpierw elementu 5 do listy i następnie do tak zmodyfikowanej listy dodanie jeszcze elementu 7. Ta konstrukcja użyta została w liniach 68-69 programu.

Podobnie przeciążony jest operator odejmowania (linie 34-37). Różnica polega tylko na tym, że teraz nowy element dodawany jest na początek listy, czyli staje się jej „głową".

W liniach 39-43 przeciążony został operator '`!`'. Jest on jednoargumentowy, a więc definiująca go metoda jest bezparametrowa. Jak widać z definicji, zwraca on liczbę elementów listy — użycie jego widzimy w linii 70 programu (patrz też następny podrozdział). Wydruk pokazuje działanie tego programu:

```
-1 0 1 2 3
List ma 5 elementow
usuwanie: -1
usuwanie: 0
usuwanie: 1
usuwanie: 2
usuwanie: 3
```

Na wydruku widzimy też ślad działania destruktora, który usunął wszystkie utworzone węzły po wyjściu sterowania z funkcji **main**. Sam wydruk listy został sporządzony za pomocą przeciążonego operatora wstawiania do strumienia; funkcja przeciążająca (linie 58-62) została zaprzyjaźniona z klasą, gdyż potrzebuje dostępu nie tylko do składowej head ale i do nazwy **Node** zadeklarowanej w sekcji prywatnej klasy **List**. Ta funkcja, jak już mówiliśmy, musi być funkcją globalną, a nie metodą.

19.3.2 Operatory jednoargumentowe

Operatory jednoargumentowe definiowane są przez metody bezparametrowe. Argumentem jest niejawnie przesyłany przez wskaźnik obiekt na rzecz którego wywoływana jest metoda, czyli na który działa operator. Tak więc dla operatorów jednoargumentowych (prefiksowych) zapis

```
@a
```

oznacza wywołanie

```
a.operator@()
```

a deklaracja metody definiującej ten operator ma postać

```
Typ operator@();
```

Przykład takiego przeciążenia widzieliśmy w już programie ***lista.cpp*** (str. 398). Pojawienie się wyrażenia '`!list`' (linia 70) spowodowało wywołanie na rzecz obiektu list metody **operator!** (zdefiniowanej w liniach 39-43).

Jednoargumentowe operatory '`++`' i '`--`' wymagają specjalnego omówienia, gdyż występują w dwóch odmianach — przyrostkowej i przedrostkowej. Jeśli przeładujemy je tak jak zwykłe operatory jednoargumentowe, to domyślnie będą to operatory przedrostkowe, a więc prawidłowy zapis wyrażeń z ich użyciem będzie miał postać

```
++a; --b;
```

Takie operatory powinny być definiowane w ten sposób, aby miały coś wspólnego ze zwiększeniem (zmniejszeniem) argumentu i zwracały l-wartość, bo tak jest dla ich standardowej implementacji dla liczb. Deklaracja metody definiującej przeciążenie preinkrementacji będzie wyglądała tak:

```
Typ operator++();
```

Jak w takim razie zdefiniować metodę przeciążającą taki operator w jego wersji przyrostkowej? Nazwa musiałaby być taka sama i również musiałaby to być metoda bezparametrowa! Aby kompilator mógł odróżnić sytuację, gdy chcemy przeciążyć operator przyrostkowy, definiując metodę dodajemy „sztuczny" (pozorny) argument typu **int**, którego *nie* używamy w ciele funkcji definiującej przeciążenie. Zatem w deklaracji/definicji nie trzeba mu nawet nadawać nazwy; unikniemy w ten sposób ostrzeżeń kompilatora o nieużywanych zmiennych.

Deklaracja metody przeciążającej operator postinkrementacji będzie więc miała postać:

```
Typ operator++(int);
```

Jeśli w ogóle potrzebny jest nam operator postdekrementacji lub postinkrementacji, to naturalne będzie zdefiniowanie go tak, aby zwracana wartość była identyczna z argumentem, a efekt zwiększenia/zmniejszenia, cokolwiek by to miało znaczyć w danym kontekście, był efektem ubocznym. Innymi słowy, preinkrementacja to „zwiększ i pobierz", a postinkrementacja to „pobierz i zwiększ". Wartość zwracana przez operatory przyrostkowe *nie* powinna być l-wartością.

Zwróćmy uwagę, że operatory '−' i '+' występują też w wersji jednoargumentowej: operator '+' to operator *„no-op"*, czyli operator, który nic nie robi, a operator '−' to operator zmiany znaku. Te wersje też można przeciążyć: ponieważ są jednoargumentowe, więc przeciążamy je oczywiście za pomocą metod bezargumentowych.

W poniższym programie przeciążone są wszelkiego rodzaju operatory „z minusem": jedno- i dwuargumentowe operatory '−' i operatory `--`; te ostatnie zarówno w wersji przedrostkowej, jak i przyrostkowej:

P155: *minus.cpp* Przeciążenia operatorów z „minusem"

```
#include <iostream>
using namespace std;

class A {
    int data;
public:
    A(int data = 0) : data(2*(data/2)) { }

    const A operator-() const {
        return A(-data);
    }

    const A operator-(const A& a) const {
        return A(data - a.data);
    }

    A& operator--() {
        ----data;
        return *this;
    }

    const A operator--(int) {
        A x(data);
        ----data;
        return x;
    }

    friend ostream& operator<<(ostream&,A);
};

ostream& operator<<(ostream& strum, A d) {
    return strum << d.data;
}

int main() {
    A data(7);

    cout << "a.   data    = " <<   data   << endl;
    cout << "b.   data--  = " <<   data-- << endl;
    cout << "c.   data    = " <<   data   << endl;
    cout << "d. --data    = " << --data   << endl;
    cout << "e.   data    = " <<   data   << endl;
    cout << "f.  -data    = " <<  -data   << endl;
    cout << "g.   data    = " <<   data   << endl;
```

```
45 }
```

W klasie **A** jest tylko jedno pole data typu **int**. Przechowywana tam liczba jest zawsze parzysta: jedyny konstruktor dba o to, aby tak było. Operatory zwiększenia i zmniejszenia są tak zdefiniowane, że dodają do lub odejmują od danej przechowywanej w składowej obiektu zawsze 2. Widzimy w tym programie przeciążenie:

- jednoargumentowego operatora '−', czyli operatora zmiany znaku (linie 9-11). Metoda zwraca przez wartość nowy obiekt klasy, w którym odwrócony jest znak składowej data. Zwracana wartość nie jest zatem l-wartością. Obiekt *this* (ten, dla którego nastąpiło wywołanie) nie jest modyfikowany, więc metoda zdeklarowana jest jako stała;
- dwuargumentowego operatora '−', czyli odejmowania (linie 13-15). Metoda definiuje odejmowanie obiektów klasy **A**. Zwraca nowy obiekt klasy, w którym składowa data jest różnicą składowych data argumentów. Zwracana wartość nie jest l-wartością;
- jednoargumentowego operatora predekrementacji '−−' (linie 17-20). Metoda zmniejsza wartość składowej data o dwa i zwraca referencję do obiektu, na rzecz którego została wywołana. Zwracana wartość *jest* więc l-wartością;
- jednoargumentowego operatora postdekrementacji (linie 22-26). „Sztuczny" argument typu **int**, z którego metoda w żaden sposób nie korzysta, służy tylko do poinformowania kompilatora, że chodzi nam o formę przyrostkową operatora zmniejszenia. Metoda zmniejsza wartość składowej data o dwa, ale zwraca *kopię* obiektu, na rzecz którego została wywołana *sprzed* modyfikacji. Zwracana wartość nie jest l-wartością.

Prócz tego, poprzez zaprzyjaźnioną funkcję globalną przeciążamy operator wstawiania do strumienia '<<' (linie 31-33). Wydruk tego programu

```
a.   data    = 6
b.   data--  = 6
c.   data    = 4
d. --data    = 2
e.   data    = 2
f.  -data    = -2
g.   data    = 2
```

wskazuje, że wszystkie przeciążenia działają zgodnie z przewidywaniem.

Jako jeszcze jeden przykład rozpatrzmy program

P156: *tabinc.cpp* Przeciążanie operatora zwiększenia

```
1 #include <iostream>
2 #include <cstring>  // memcpy
3 using namespace std;
```

```
4
5  class Tablica {
6      int size;
7      int* tab;
8  public:
9      Tablica(int size, const int* t)
10         : size(size),
11           tab((int*)memcpy(new int[size], t,
12                            size*sizeof(int)))
13     { }
14
15     Tablica(const Tablica& t)
16         : size(t.size),
17           tab((int*)memcpy(new int[size], t.tab,
18                            size*sizeof(int)))
19     { }
20
21     ~Tablica() { delete [] tab; }
22
23     Tablica& operator++();
24     Tablica operator++(int);
25     void showTab(const char* nap);
26 };
27
28 Tablica& Tablica::operator++() {
29     for (int i = 0; i < size; ++i)
30         ++tab[i];
31     return *this;
32 }
33
34 Tablica Tablica::operator++(int) {
35     Tablica t(*this);
36     ++*this;
37     return t;
38 }
39
40 void Tablica::showTab(const char* nap) {
41     cout << nap;
42     for (int i = 0; i < size; i++)
43         cout << tab[i] << " ";
44     cout << endl;
45 }
46
47 int main() {
48     int tab[] = {1,2,3,4};
49
```

```
50      Tablica T(4,tab);
51      T.showTab("Tablica wyjsciowa T: ");
52      Tablica t = ++T;
53      t.showTab("  Po t = ++T t jest: ");
54      T.showTab("           a T jest: ");
55
56      Tablica S(4,tab);
57      S.showTab("Tablica wyjsciowa S: ");
58      Tablica s = S++;
59      s.showTab("  Po s = S++ s jest: ");
60      S.showTab("           a S jest: ");
61  }
```

Zdefiniowana w tym programie klasa **Tablica** ma pole wskaźnikowe. Składową jest tu wskaźnik do alokowanej dynamicznie tablicy liczb całkowitych (dlatego musieliśmy zadbać o zwalnianie zaalokowanego obszaru pamięci w destruktorze). W ten sposób obiekt jest niewielki, a dla każdej tablicy alokujemy tylko tyle miejsca w pamięci, ile trzeba, ale nie więcej. Drugą składową każdego obiektu jest wymiar tablicy.

Zauważmy sposób kopiowania tablic w konstruktorach (linie 11 i 17). Kopiujemy tu cały obszar pamięci, a nie element po elemencie — taki sposób kopiowania tablic jest dla typów prostych znacznie efektywniejszy, ale może *nie* działać jeśli elementami tablicy są obiekty klas!

Zauważmy, że *przyrostkowy* operator zwiększania (postinkrementacji), który jest zdefiniowany w liniach 34-38, zwraca obiekt identyczny z argumentem, ale sprzed modyfikacji. Zwiększenie jest tu efektem ubocznym, widocznym dopiero przy następnym użyciu obiektu. W definicji postinkrementacji, w linii 36, wykorzystaliśmy wcześniej przeciążony (linie 28-32) *przedrostkowy* operator zwiększania (preinkrementacji). W linii 35, za pomocą konstruktora kopiującego, tworzymy kopię całego obiektu, następnie wywołujemy na rzecz *this operator preinkrementacji, po czym zwracamy (linia 37) utworzoną wcześniej kopię.

Program drukuje:

```
Tablica wyjsciowa T: 1 2 3 4
  Po t = ++T t jest: 2 3 4 5
           a T jest: 2 3 4 5
Tablica wyjsciowa S: 1 2 3 4
  Po s = S++ s jest: 1 2 3 4
           a S jest: 2 3 4 5
```

W klasie tej, mimo że zawiera pola wskaźnikowe, *nie* przeciążyliśmy operatora przypisania tak, aby przypisanie było głębokie, a więc nie dotyczyło wskaźnika zawartego w obiekcie, ale danych we wskazywanej przez ten wskaźnik tablicy. W tego typu klasie takie przeciążenie powinno być zwykle zdefiniowane — dokładniej przeciążaniem operatora '=' zajmiemy się w następnym podrozdziale.

19.4 Operatory „specjalne”

Operatory '=', '->', '()' i '[]' są w pewien sposób specjalne, zatem omówimy je po kolei w osobnych podrozdziałach. Wszystkie one mogą być przeciążone wyłącznie za pomocą metod — nigdy za pomocą globalnych funkcji.

19.4.1 Operator przypisania

Operator przypisania '=' powinien być przeładowany praktycznie zawsze, gdy w klasie występują pola wskaźnikowe (należy wtedy prawie zawsze zdefiniować też destruktor i konstruktor kopiujący). W przeciwnym razie użyty zostanie domyślny, dostarczony przez system operator przypisania, który skopiuje pole po polu zawartość obiektu z prawej strony przypisania do obiektu stojącego po lewej stronie, co prawdopodobnie nie jest tym, o co nam chodzi, jeśli występują w naszej klasie składowe wskaźnikowe. Wtedy bowiem nie chcemy zwykle kopiować wskaźników, ale raczej to, na co one wskazują. To, na co one wskazują, choć jest logicznie częścią obiektu, nie należy do niego fizycznie: obiekt tylko „przechowuje” odpowiedni adres — domyślnie tylko ten adres zostanie przekopiowany.

Żaden domyślny operator przypisania *nie* zostanie przez system wygenerowany, jeśli klasa zawiera pola ustalone (**const**), referencyjne lub pola będące obiektami klasy, w której operator przypisania jest prywatny, lub, z podobnych powodów, w ogóle go nie ma.

W takich przypadkach, nawet jeśli klasa nie zawiera pól wskaźnikowych, operator przypisania trzeba przeciążyć (jeśli w ogóle zamierzamy korzystać z przypisań obiektów klasy).

Zobaczmy, jakiego rodzaju błędy mogą pojawić się, gdy operator przypisania nie został przeciążony dla klasy z polem wskaźnikowym:

P157: *ovrlderr.cpp* Brak odpowiedniego operatora przypisania

```
#include <iostream>
#include <cstring>   // strcpy, strlen
using namespace std;

struct A {
    char* name;

    A(const char* s)
        : name(strcpy(new char[strlen(s)+1],s)) {
        cerr << "  ctor: " << (void*)name << endl;
    }

    A(const A& k)
```

```
14          : name(strcpy(new char[strlen(k.name)+1],k.name)) {
15          cerr << "cpctor: " << (void*)name << endl;
16      }
17
18      ~A() {
19          cerr << "  dtor: "  << (void*)name << endl;
20          delete [] name;
21      }
22  };
23
24  A ob1("ob1");
25
26  int main() {
27      cerr << "MAIN" << endl;
28      A ob2(ob1);
29      A ob3 = ob2; // copy-ctor
30
31      ob1 = ob3;
32
33      cerr << "  ob1.name: " << (void*)ob1.name << endl;
34      cerr << "  ob3.name: " << (void*)ob3.name << endl;
35
36      cerr << "THE END" << endl;
37  }
```

W liniach 8-11 definiujemy konstruktor tworzący obiekt na podstawie przesłanego napisu. W prawidłowy sposób dba on, aby tworzony obiekt zawierał wskaźnik do specjalnie zaalokowanego obszaru pamięci, a napis wskazywany przez adres będący argumentem konstruktora został tam przekopiowany. Użyta w linii 9 konstrukcja jest podobna do tej, jakiej użyliśmy w programie ***osoba3.cpp*** (str. 302). Całość tej konstrukcji umieściliśmy w liście inicjalizacyjnej (patrz rozdz. 15.3.2, str. 304). Za pomocą

```
new char[strlen(s)+1]
```

alokujemy pamięć o rozmiarze odpowiadającym długości napisu `s` (plus jeden na znak '\0'). Funkcja **strcpy** kopiuje napis z `s` do zaalokowanej pamięci, której adres jest jej pierwszym argumentem, i zwraca tenże adres. Ten właśnie adres (zwrócony przez funkcję **strcpy**) jest użyty do zainicjowania składowej `name` tworzonego obiektu, zgodnie ze składnią właściwą liście inicjalizacyjnej (funkcje takie jak **strlen** czy **strcpy** są dokładniej omówione w rozdz. 17.1 na stronie 355).

Podobnie jest zbudowany konstruktor kopiujący, zdefiniowany w liniach 13-16 (patrz rozdz. 15.3.1 na stronie 296).

W liniach 18-21 definiujemy destruktor. Wydaje się, że klasa jest kompletna. Jednak uruchamiając program widzimy, że coś jest źle:

```
ctor: 0x804b008
```

```
MAIN
cpctor: 0x804b018
cpctor: 0x804b028
  ob1.name: 0x804b028
  ob3.name: 0x804b028
THE END
  dtor: 0x804b028
  dtor: 0x804b018
  dtor: 0x804b028
```

Po pierwsze, po wyjścu z funkcji **main** wywoływany jest trzy razy destruktor; to nas nie dziwi, bo utworzone zostały trzy obiekty, ale kłopot w tym, że dwa z nich zwalniają pamięć pod tym samym adresem 0x804b028. Natomiast pamięć pod adresem 0x804b008 która pierwotnie zawierała napis związany z obiektem ob1 w ogóle nie została zwolniona!

Łatwo widać, dlaczego tak się stało. Podczas przypisywania w linii 31 zawartość obiektu ob3 została przekopiowana do obiektu ob1. W szczególności została przekopiowana wartość składowej name, czyli adres napisu zaalokowanego w konstruktorze kopiującym podczas tworzenia obiektu ob3 (linia 29). Zatem po tym przypisaniu obiekty ob1 i ob3 są, co prawda, rozłączne, ale zawierają ten sam adres napisu w swoich składowych name (jak widać z wydruku, wynosi on tu 0x804b028). Co gorsza, adres pierwotnie pamiętany w składowej name obiektu obj1 (w tym przykładzie 0x804b008) uległ zamazaniu i jest już nie do odzyskania, zatem pamięć wskazywana przez ten adres nigdy nie będzie mogła być zwolniona! W momencie gdy obiekty są usuwane (w naszym przypadku po zakończeniu wykonywania funkcji **main**), uruchamiany jest destruktor zwalniający pamięć przydzieloną na napis wskazywany przez składową name. Zatem destruktor zostanie u nas wywołany dwa razy z tym samym adresem: raz przy usuwaniu obiektu ob3, a potem przy usuwaniu obiektu ob1. Może to spowodować załamanie programu (choć nie musi; tak czy owak jest to błąd o nieprzewidywalnych skutkach).

A zatem musimy tak przedefiniować operator przypisania, aby uniknąć tego rodzaju kłopotów. Zrobić to można wyłącznie za pomocą metody, nigdy funkcji globalnej lub statycznej funkcji składowej.

Zadaniem przypisania jest sensowne przepisanie zawartości obiektu po prawej stronie znaku '=' do obiektu po lewej, którym będzie *this, bo metoda definiująca przypisanie będzie wywołana, jak zwykle dla operatorów dwuargumentowych, właśnie na rzecz obiektu stojącego po lewej stronie operatora.

W ciele metody musimy zwykle zadbać, aby dynamicznie zaalokowane obiekty (jak tablice, czyli również C-napisy), do których wskaźniki są składowymi obiektu, zostały we właściwy sposób przekopiowane. Ponieważ przypisanie `a=a` jest zawsze legalne, należy przy tym uważać, żeby nie zlikwidować obiektów (tablic, napisów) wskazywanych przez składowe wskaźnikowe przed ich kopiowaniem. Aby z kolei możliwe było kaskadowe przypisanie (czyli `a=b=c`), operator powinien zwracać referencję do obiektu po lewej stronie, czyli tego, na rzecz którego został wywołany (`return *this`). Argument (czyli obiekt po prawej stronie przypisania) jest często przekazywany przez referencję, zwykle ustaloną; unika się wtedy niepotrzebnego kopiowania. Zatem dla

klasy **A** metoda przeciążająca operator przypisania miałaby nagłówek

```
A& operator=(const A&);
```

Jako przykład rozpatrzmy następujący program:

P158: *ovrldeq.cpp* Przeciążanie operatora przypisania

```
#include <iostream>
#include <cstring>   // strcpy, strlen
using namespace std;

struct A {
    char* name;

    A() : name(new char[1]) {
        cerr << "dfctor: " << (void*)name << endl;
        name[0] = '\0';
    }

    A(const char* s)
        : name(strcpy(new char[strlen(s)+1],s)) {
        cerr << "  ctor: " << (void*)name << endl;
    }

    A(const A& k)
        : name(strcpy(new char[strlen(k.name)+1],k.name)) {
        cerr << "cpctor: " << (void*)name << endl;
    }

    A& operator=(const A& k) {

        if (this == &k) return *this;

        cerr << "delete: " << (void*)name << endl;
        delete [] name;
        name = strcpy(new char[strlen(k.name)+1],k.name);
        cerr << "   op=: " << (void*)name << endl;
        return *this;
    }

    ~A() {
        cerr << "  dtor: "  << (void*)name << endl;
        delete [] name;
    }
};

```

```
40 A ob1("ob1");
41
42 int main() {
43     cerr << "MAIN" << endl;
44     A ob2(ob1);
45     A ob3 = ob2; // copy-ctor
46
47     ob1 = ob3;
48
49     cerr << "  ob1.name: " << (void*)ob1.name << endl;
50     cerr << "  ob3.name: " << (void*)ob3.name << endl;
51
52     cerr << "THE END" << endl;
53 }
```

Ta klasa jest w zasadzie taka sama jak ta z poprzedniego programu. Dopisaliśmy tu konstruktor bezargumentowy (domyślny), aby klasa była bardziej kompletna (linie 8-11). Zauważmy, że nawet alokując napis pusty, czyli składający się z jednego znaku (znaku '\0'), alokujemy ten jeden bajt w postaci tablicy, a nie pojedynczej zmiennej. Przyczyną jest postać destruktora, który używa zawsze formy tablicowej (z nawiasami kwadratowymi) do zwalniania pamięci.

W liniach 23-32 definiujemy metodę przeciążającą operator przypisania. Zauważmy konstrukcję tej funkcji, gdyż jest ona typowa:

- typem wartości zwracanej jest referencja do obiektu klasy, dla której definiujemy przeciążenie. Wartością zwracaną powinna być zawsze referencja do tego obiektu, na rzecz którego metoda jest wywołana. Instrukcja powrotu powinna zatem mieć postać `return *this` (wyrażenie `*this` jest równoważne nazwie obiektu wskazywanego przez this);
- zawsze najpierw sprawdzamy (linia 25), czy obiekt, na rzecz którego metoda jest wywołana, czyli lewy argument operatora '=', nie jest tym samym obiektem, który wystąpił po prawej stronie przypisania. Robimy to porównując adresy tych obiektów (wartość this jest tu adresem obiektu, na rzecz którego metoda była wywołana). Jeśli adresy te są takie same, czyli przypisanie ma postać 'a=a', to zwracamy *this i działanie metody kończy się, bo nie ma nic do zrobienia;
- mając gwarancję, że nie zachodzi przypadek 'a=a', usuwamy napis (tablicę) wskazywany przez składową wskaźnikową (linia 28);
- alokujemy nową pamięć i kopiujemy tam tablicę (napis) z obiektu będącego prawym argumentem przypisania; adres przydzielonego obszaru pamięci zapamiętujemy w składowej wskaźnikowej obiektu, na który następuje przypisanie;
- zwracamy *this.

Program główny nie różni się od poprzedniego. Wynik jest jednak nieco inny:

```
  ctor: 0x804b008
MAIN
cpctor: 0x804b018
cpctor: 0x804b028
delete: 0x804b008
   op=: 0x804b008
  ob1.name: 0x804b008
  ob3.name: 0x804b028
THE END
  dtor: 0x804b028
  dtor: 0x804b018
  dtor: 0x804b008
```

Teraz w czasie przypisania z linii 47 (’ob1=ob3’) najpierw zwolniona została pamięć dotychczas zajmowana przez napis związany z obiektem ob1 (pod adresem 0x804b008), a następnie został przydzielony nowy obszar pamięci na napis. W tym przypadku system zaalokował ten sam obszar, który przed chwilą został zwolniony, ale nie jest to istotne — równie dobrze mógł to być obszar pod zupełnie innym adresem. Ważne jest, że teraz nie ma żadnych wycieków pamięci — każdy zaalokowany przez **new** segment pamięci jest zwalniany. Nie pojawia się też problem zwalniania pamięci już raz zwolnionej — wszystkie destruktory wywołują **delete** z adresami obszarów jeszcze nie zwolnionych.

Przypomnijmy tu na marginiesie, że wyrażenie

```
A a = b;
```

nie oznacza przypisania; jest to inna forma deklaracji/definicji nowej zmiennej klasy z wywołaniem konstruktora kopiującego, a więc równoważna ’A a(b)’. Aby operator ’=’ oznaczał przypisanie, po lewej stronie musi stać wyrażenie będące l-wartością i oznaczające istniejącą już wcześniej zmienną.

Inny przykład przeciążenia operatora przypisania znajdujemy w poniższym programie:

P159: *op.cpp* Klasa tablicowa z przeciążonym operatorem przypisania

```
1  #include <iostream>
2  #include <cstring>   // memcpy
3  using namespace std;
4
5  class Tabint {
6      static int ID;
7      int        id;
8      int      size;
9      int      *tab;
10 public:
11     Tabint(const int *t, int size)
12         : id(++ID), size(size),
```

```
13          tab((int*)memcpy(new int[size], t,
14                           size*sizeof(int))) {
15      cout << "Konstruktor: id = " << id << endl;
16  }
17
18  Tabint(const Tabint& t)
19      : id(++ID), size(t.size),
20        tab((int*)memcpy(new int[size], t.tab,
21                           size*sizeof(int))) {
22      cout << "Konstr. kop: " << t.id << "-->" << id
23                                                << endl;
24  }
25
26  ~Tabint() {
27      cout << "Usuwamy:    id = " << id << endl;
28      delete [] tab;
29  }
30
31  Tabint& operator=(const Tabint& t) {
32      cout << "Przypisanie: " << id << "<--" << t.id
33                                                << endl;
34      if (this != &t) {
35          delete [] tab;
36          size = t.size;
37          tab = (int*)memcpy(new int[size], t.tab,
38                             size*sizeof(int));
39      }
40      return *this;
41  }
42 };
43 int Tabint::ID = 0;
44
45 int main() {
46     int tab[] = {1,2,3};
47     int size  = sizeof(tab)/sizeof(int);
48
49     Tabint* pt1 = new Tabint(tab,size);
50     Tabint   t2 = *pt1;
51     Tabint   t3(t2);
52
53     *pt1 = t2;
54 }
```

Zamiast funkcji **strcpy** kopiującej C-napisy użyliśmy tu funkcji **memcpy** o podobnym działaniu. Pozwala ona kopiować duże obszary pamięci „za jednym zamachem”, bez użycia pętli kopiującej tablice element po elemencie. Z wydruku

```
Konstruktor: id = 1
Konstr. kop: 1-->2
Konstr. kop: 2-->3
Przypisanie: 1<--2
Usuwamy:    id = 3
Usuwamy:    id = 2
```

zauważamy, że element pierwszy (utworzony w linii 49) nie jest usuwany. Dzieje się tak dlatego, że jako jedyny został utworzony na stercie (przez operator **new**) i jawnie nie usunięty. Pozostałe obiekty zostały utworzone na stosie, a więc będą usunięte automatycznie, co będzie się oczywiście wiązało z wywołaniem destruktora.

19.4.2 Operator indeksowania

Operator '`[]`' (indeksowania) jest dwuargumentowy: w wyrażeniu '`a[i]`' nazwa `a` jest nazwą pierwszego argumentu, a `i` drugiego. Funkcja przeciążająca ten operator musi być metodą jednoparametrową; będzie wywoływana na rzecz `a`, które musi zatem być nazwą obiektu klasy, dla której definiujemy przeciążenie, natomiast indeks `i` będzie przesłany jako argument. Argument ten (indeks) *nie musi* być typu całościowego, choć najczęściej jest.

Deklaracja metody przeciążającej operator '`[]`' powinna mieć postać

```
Typ operator[](Typ_arg);
```

gdzie **Typ** jest typem funkcji (wartości zwracanej), a **Typ_arg** jest typem indeksu. Metoda ta będzie wywołana na rzecz obiektu `a`, gdy pojawi się wyrażenie

```
a[i]
```

i `a` będzie nazwą obiektu klasy, w której dokonaliśmy przeciążenia, a typ `i` jest zgodny z typem **Typ_arg**. Wywołanie to będzie formalnie miało postać

```
a.operator[](i)
```

Funkcja przeciążająca operator '`[]`' może wykonywać dowolne zadanie, ale rozsądne jest, jeśli robi coś pojęciowo podobnego do wyłuskania wartości poprzez indeks. Ważne jest wtedy takie zdefiniowanie przeładowania, by obiekt zwracany był l-wartością (co możemy uzyskać zwracają go przez referencję), czyli by było możliwe przypisanie do niego, a zatem aby mógł występować również po lewej stronie przypisania — takie jest bowiem normalne zachowanie operatora indeksowania. Zatem oba poniższe wyrażenia powinny być poprawne:

```
x = arr[i];
arr[j] = y;
```

Prosta klasa **Litera** z poniższego programu demonstruje zarówno przeciążanie operatora indeksowania, jak i przeciążanie operatora przypisania, konstruktor kopiujący i destruktor: a więc komplet tego co jest wymagane przy istnieniu składowych wskaźnikowych.

P160: *litera.cpp* Przeciążanie operatora indeksowania

```
#include <iostream>
#include <cstring>
using namespace std;

class Litera {
    char* napis;
public:
    Litera(const Litera& k)
        : napis(strcpy(new char[strlen(k.napis)+1],
                       k.napis))
    { }

    Litera(const char* napis)
        : napis(strcpy(new char[strlen(napis)+1],
                       napis))
    { }

    Litera& operator=(const Litera&);
    char& operator[](int);

    ~Litera() { delete [] napis; }

    friend ostream& operator<<(ostream&,const Litera&);
};

char& Litera::operator[](int n) {
    int len = strlen(napis);
    if ( n < 0 || n >= len )
        // zwracamy odnośnik do NUL jeśli zły indeks
        return napis[len];
    else
        return napis[n];
}

Litera& Litera::operator=(const Litera& k) {
    if (this == &k ) return *this;
    delete [] napis;
    napis = strcpy(new char[strlen(k.napis)+1],k.napis);
    return *this;
}

ostream& operator<<(ostream& str, const Litera& k) {
    return str << k.napis;
}
```

```
45
46 int main() {
47     Litera a("Kasia");
48     cout << "a=" << a << endl;
49
50     char c = a[100];
51     if (c == '\0') cout << "Zly zakres!" << endl;
52     else           cout << "znak: " << c << endl;
53
54     c = a[0];
55     if (c == '\0') cout << "Zly zakres!" << endl;
56     else           cout << "znak: " << c << endl;
57
58     a[0] = 'B';
59     cout << "a=" << a << endl;
60 }
```

Jak widać z definicji w liniach 26-33, działanie operatora indeksowania polega tu na dostarczeniu, przez referencję, znaku napisu o odpowiednim indeksie (pozycji). Metoda ta jest o tyle sensowna, że sprawdza zakres i w przypadku, gdy indeks jest za duży albo za mały, zwraca referencję do znaku '\0', zamiast zwracać jakąś przypadkową wartość. Wydruk z tego programu to

```
a=Kasia
Zly zakres!
znak: K
a=Basia
```

Widzimy, że rzeczywiście indeksowanej nazwy obiektu klasy **Litera** możemy używać tak jak nazwy tablicy: zarówno po prawej, jak i po lewej stronie przypisania.

Spójrzmy na przykład, w którym indeks nie jest całkowity:

P161: *zakr.cpp* Indeksowanie liczbą rzeczywistą

```
1 #include <iostream>
2 using namespace std;
3
4 class Zakresy {
5     int    licz[3];
6     double min, max;
7 public:
8     Zakresy(double min, double max)
9         : min(min), max(max) {
10        licz[0]=licz[1]=licz[2]=0;
11    }
12
13    int& operator[](double x) {
```

```
14          int i;
15
16          if      ( x >  max ) i = 2;
17          else if ( x >= min ) i = 1;
18          else                 i = 0;
19
20          return licz[i];
21      }
22
23      friend ostream& operator<<(ostream&, const Zakresy&);
24  };
25
26  ostream& operator<<(ostream& str, const Zakresy& z) {
27      return str << "Zakres = [" << z.min << ", " << z.max
28                 << "]:  " << "Ponizej " << z.licz[0]
29                 << "; w zakr. " << z.licz[1] << "; Powyzej "
30                 << z.licz[2];
31  }
32
33  int main() {
34      Zakresy zakres(3.0, 5.5);
35      double x;
36      cout << "Podaj liczby (zero konczy)" << endl;
37
38      while ( ( cin >> x) && x ) zakres[x]++;
39
40      cout << zakres << endl;
41      cout << "x = 4.7 -> licz = " << zakres[4.7] << endl;
42  }
```

Klasa **Zakresy** definiuje składowe min, max oraz trzyelementową tablicę liczb całkowitych licz, której zadaniem jest zliczanie „danych" poniżej zakresu [min, max], wewnątrz tego zakresu oraz powyżej tego zakresu. Operator indeksowania jest przeciążony za pomocą metody o nazwie **operator[]** pobierającej argument typu **double**. Działanie jej polega na tym, że jeśli zakres jest obiektem klasy **Zakresy**, to dla x typu **double** wyrażenie `zakres[x]` jest referencją do odpowiedniego elementu tablicy licz będącej składową tego obiektu: `licz[0]`, jeśli wartość x leży poniżej min, `licz[1]`, jeśli wartość ta leży w zakresie definiowanym przez min i max, a `licz[2]`, jeśli wypada powyżej max. A zatem użyte w linii 38 wyrażenie `zakres[x]++` zwiększa odpowiedni element tablicy licz w zależności od wartości x. Natomiast wartość `zakres[4.7]` w linii 41 to wartość tego elementu tablicy licz, którego indeks odpowiada podanej wartości (4.7 mieści się w zakresie [3,5.5], wobec tego jest to aktualna wartość licz[1]). Wydruk z tego programu

```
Podaj liczby (zero konczy)
6  8.9  1  3.4  5  3.1
1    2  3    4  5    6  4.6  2.8  0
```

```
Zakres = [3, 5.5]:  Ponizej 4; w zakr. 7; Powyzej 3
x = 4.7 -> licz = 7
```

potwierdza prawidłowość przeciążenia. W linii 38 w warunku zakończenia pętli **while** użyta została koniunkcja wyrażeń logicznych, dzięki której wczytywanie zakończy się zarówno wtedy gdy stan strumienia cin przejdzie w *bad* (na skutek błędu lub dojścia do końca pliku), jak i wtedy, gdy wczytana zostanie liczba 0 (zero). Indeksowanie w tym programie zwraca referencję do składowej prywatnej. Nie jest to praktyka godna polecenia: tu została użyta wyłącznie w celach dydaktycznych.

19.4.3 Operator wywołania

Operator wywołania '`()`' jest jedynym, który może mieć dowolną liczbę argumentów. Przeciążać go można tylko za pomocą metody. Metoda ta ma nazwę **operator()** — tu nawiasy są częścią nazwy, za którą to nazwą w zwykły sposób umieszczamy listę, być może pustą, parametrów ujętą w następną parę nawiasów. Deklaracja takiej metody ma więc postać

```
Typ operator()(Typ_arg1, Typ_arg2, Typ_arg3);
```

przy czym liczba i typ parametrów są dowolne. Wywołanie metody będzie miało miejsce po napotkaniu wyrażenia

```
obj(a,b,c)
```

gdzie obj jest nazwą pewnego obiektu klasy, w której dokonaliśmy przeciążenia, a a, b i c są typów zgodnych z typami parametrów metody. Wywołanie będzie na rzecz obiektu obj i będzie miało postać

```
obj.operator()(a,b,c)
```

Zauważmy, że postać wyrażenia `obj(a,b,c)` jest taka sama jak postać normalnego wywołania funkcji, z tą tylko różnicą, że obj jest tu nazwą *obiektu*, a nie funkcji. Takie obiekty można zatem „wywoływać” jak funkcje — nazywamy je zatem **obiektami funkcyjnymi** (ang. *callable object, functor*) lub **obiektami wywoływalnymi**. Wywołanie nie musi zwracać l-wartości, tak jak nie zwraca l-wartości większość „normalnych” funkcji (zwracających wynik przez wartość). Równie dobrze może zwrócić odniesienie (referencję), która *jest* l-wartością.

W poniższym przykładzie dzięki przeciążeniu operatora '`()`' uzyskana została prosta implementacja dynamicznie alokowanych tablic trzywymiarowych (dynamicznie, a więc których wymiary mogą być ustalone dopiero podczas wykonania). W tym przypadku typem zwracanym jest odnośnik (referencja) do liczby typu **int** (elementu macierzy) tak, że metoda *zwraca* l-wartość; wywołanie z trzema argumentami zastępuje potrójne indeksowanie tablicy trzywymiarowej.

P162: *arr3dim.cpp* Przeciążanie operatora wywołania

```
#include <iostream>
#include <cstring>
#include <iomanip>
using namespace std;

class Arr3D {
    int  dim1, dim2, dim3;
    int* arr;
public:
    Arr3D() : dim1(1),dim2(1),dim3(1),
              arr(new int[1])
    { }

    Arr3D(int dim1, int dim2, int dim3)
        : dim1(dim1), dim2(dim2), dim3(dim3),
          arr(new int[dim1*dim2*dim3])
    { }

    Arr3D(const Arr3D& t)
        : dim1(t.dim1), dim2(t.dim2), dim3(t.dim3),
          arr((int*)memcpy(new int[dim1*dim2*dim3],
                           t.arr,
                           dim1*dim2*dim3*sizeof(int)))
    { }

    ~Arr3D() { delete [] arr; }

    Arr3D& operator=(const Arr3D&);
    int& operator()(int,int,int);
};

int& Arr3D::operator()(int n1, int n2, int n3)  {
    return *(arr + n1*dim2*dim3 + n2*dim3 + n3);
}

Arr3D& Arr3D::operator=(const Arr3D& t) {
    if ( this != &t ) {
        delete [] arr;
        dim1 = t.dim1;
        dim2 = t.dim2;
        dim3 = t.dim3;
        arr  = (int*)memcpy(new int[dim1*dim2*dim3],
                            t.arr,
                            dim1*dim2*dim3*sizeof(int));
```

```
45      }
46      return *this;
47  }
48
49  int main() {
50      int dim1 = 1000, dim2 = 1000, dim3 = 50;
51
52      Arr3D T1(dim1,dim2,dim3), T2;
53
54      for (int i = 0; i < dim1; i++)
55          for (int j = 0; j < dim2; j++)
56              for (int k = 0; k < dim3; k++)
57                  T1(i,j,k) = i+j+k;
58      T2 = T1;
59
60      cout << "T2(999, 999 , 2) = "
61           << setw(4) << T2(999,999,2) << endl;
62
63      cout << "T2(  0,    0, 9) = "
64           << setw(4) << T2(  0,  0,9) << endl;
65  }
```

Zauważmy, że w klasie tej zdefiniowaliśmy konstruktor kopiujący, destruktor i przeciążyliśmy operator przypisania: jest to konieczne, bo klasa zawiera pole wskaźnikowe. Obiekty tej klasy reprezentują tablice trzywymiarowe (o trzech indeksach). Tak naprawdę, jak widzimy z definicji klasy, tablica arr jest wskaźnikiem do tablicy jednowymiarowej o wymiarze równym iloczynowi wymiarów dim1, dim2 i dim3. Dostęp do tej tablicy z zewnątrz mamy wyłącznie za pomocą metody **operator()** (linie 32-34), a więc za pomocą wyrażeń typu `T(i,j,k)`, gdzie T jest nazwą obiektu klasy **Arr3D**. Metoda traktuje argumenty jak indeksy i wylicza na ich podstawie odpowiadający żądanemu elementowi indeks w tablicy jednowymiarowej arr (jak widać, do obliczenia tego indeksu nie jest potrzebny pierwszy wymiar dim1 tablicy trzywymiarowej). Ponieważ zwracana jest referencja do odpowiedniego elementu tablicy, wyrażenie `T(i,j,k)` wygląda i zachowuje się jak element `T[i][j][k]` „normalnej” tablicy trzywymiarowej, w szczególności *jest* l-wartością, a zatem może występować również po lewej stronie przypisania, jak w linii 57.

W programie tworzymy obiekt T1 klasy **Arr3D** odpowiadający trzywymiarowej tablicy o wymiarach $1000 \times 1000 \times 50$ oraz drugi obiekt T2, tworzony za pomocą konstruktora domyślnego, a więc o wymiarach $1 \times 1 \times 1$ (linia 52). Wewnątrz obiektu T1 tablica jest reprezentowana jako jednowymiarowa tablica 50 milionów elementów, ale ponieważ jest to składowa prywatna, użytkownik klasy nie musi o tym wiedzieć. W liniach 54-57 zapełniamy tę tablicę — każdy element jest sumą odpowiadających mu indeksów. W linii 58 dokonujemy przypisania dwóch obiektów klasy **Arr3D**, aby sprawdzić prawidłowość przeciążenia operatora przypisania. Następnie drukujemy dwie przykładowe wartości elementów

```
T2( 999, 999 , 2) = 2000
T2(   0,     0, 9) =    9
```

aby przekonać się, że wszystko przebiega poprawnie i wartości `T(i,j,k)` można rzeczywiście traktować jak elementy trzywymiarowej tablicy. Na marginesie zauważmy, że operowanie na dużych macierzach wymaga uwagi i pewnego doświadczenia; na przykład zmieniając w liniach 54-56 kolejność pętli otrzymalibyśmy równoważny program, który jednak na większości maszyn wykonywałby się 4-6 razy wolniej.

19.4.4 Operator wyboru składowej przez wskaźnik

Operator wyboru składowej przez wskaźnik ('->') jest przeciążany jako operator *jednoargumentowy*: musi być zatem przeciążany jako bezparametrowa metoda klasy. Niejawnym argumentem jest więc obiekt, którego nazwa pojawia się po *lewej* stronie operatora — na rzecz tego właśnie obiektu metoda zostanie wywołana, bez przekazywania żadnych jawnych argumentów. Zauważmy, że po lewej stronie operatora '->' stoi w tym przypadku obiekt, a nie, jak normalnie, wskaźnik (przypomnijmy, że operator bezpośredniego wyboru składowej, operator „kropka", w ogóle nie może być przeciążany)

Wartością zwracaną przez metodę przeciążającą musi być wartość wskaźnikowa (adres), która następnie zostanie użyta jako lewy operand „zwykłego" operatora '->'. Oczywiście, możliwa jest też sytuacja, że zwrócony zostanie obiekt, dla którego znowu przeciążony jest operator ->, który z kolei zwraca wskaźnik (lub znowy taki obiekt...).

Tak więc, jeśli obj jest nazwą obiektu klasy, w której przeciążony został operator '->', to wyrażenie

```
obj->b
```

jest równoważne

```
temp = obj.operator->(), temp->b
```

Innymi słowy:

- wywoływana jest metoda **operator->** na rzecz obiektu obj;
- zwrócony wynik, tu oznaczony przez temp, musi być typu wskaźnikowego, a więc wartością temp jest adres pewnego obiektu pewnej klasy (niekoniecznie tej, dla której przeciążyliśmy operator '->');
- z obiektu wskazywanego przez temp wybierana jest składowa o nazwie b (taka składowa w klasie tego obiektu musi oczywiście istnieć). Wartością całego wyrażenia jest wartość tej składowej.

W poniższym przykładzie dla obiektu AB klasy **Segment** wyrażenie `AB->x` zwraca współrzędną x-ową jednego z dwu punktów — obiektów klasy **Point** — określających odcinek. Typem zwracanym przez metodę **operator->** w klasie **Segment** jest *wskaźnik do ustalonego obiektu klasy* **Point** (a *nie* klasy **Segment**):

P163: *ovrlskl.cpp* Przeciążanie operatora wyboru składowej

```
#include <iostream>
using namespace std;

struct Point {
    int x, y;

    Point(int x = 0, int y = 0) : x(x), y(y)
    { }

    double r2() const { return x*x + y*y; }
};

struct Segment {
    Point A, B;

    Segment(Point A = Point(), Point B = Point())
        : A(A), B(B)
    { }

    const Point* operator->() const {
        return (A.r2() < B.r2()) ? &A : &B;
    }
};

ostream& operator<<(ostream& str, const Point& A) {
    return str << "P[" << A.x << "," << A.y << "]";
}

ostream& operator<<(ostream& str, const Segment& AB) {
    return str << AB.A << "--" << AB.B;
}

int main() {

    Point    A(1,0),  B(8,6),  C(4,3);
    Segment AB(A,B), BC(B,C), CA(C,A);

    cout << "AB = " << AB << ": AB->x = "
                    << AB->x << endl;

    cout << "BC = " << BC << ": BC->y = "
                    << BC->y << endl;

    cout << "CA = " << CA << ": CA->x = "
```

```
45                          << CA->x << endl;
46 }
```

Metoda przeciążająca zwraca wskaźnik do tego z końców odcinka, który leży bliżej początku układu współrzędnych (linia 21). Z obiektu (punktu) wskazywanego przez ten wskaźnik, za pomocą „zwykłego" operatora '->', wybierana jest składowa x lub y. Widzimy to w liniach 38-45, które drukują wynik:

```
AB = P[1,0]--P[8,6]: AB->x = 1
BC = P[8,6]--P[4,3]: BC->y = 3
CA = P[4,3]--P[1,0]: CA->x = 1
```

Na przykład BC w liniach 41-42 jest nazwą obiektu klasy **Segment** reprezentującego odcinek o końcach opisywanych przez obiekty B i C klasy **Point**. W klasie **Segment** operator '->' został przeciążony (linie 20-22) i zwraca adres jednego z określających ten odcinek punktów. To właśnie z tego obiektu klasy **Point** wybrana zostanie składowa y. Zauważmy, że w klasie **Segment**, której obiektem jest BC, składowej o nazwie y w ogóle nie ma. Ponadto BC jest tu nazwą obiektu, a nie wskaźnika, więc normalnie trzeba by tu było użyć kropki a nie „strzałki".

19.5 Semantyka przenoszenia

Poczynając od standardu C++11, istnieje druga forma referencji, oznaczana *dwoma*, a nie jednym znakiem '&'. Takie referencje nazywane są **r-referencjami**. Takie referencje mogą być związane („być inną nazwą") tylko z r-wartościami, czyli obiektami tymczasowymi, bez dostępnego użytkownikowi adresu. L-wartości też mają oczywiście wartość, ale prócz tego mają dobrze określoną lokalizację w pamięci; r-wartości natomiast to „tylko wartości".

Rozważmy następujące instrukcje

```
1     int i = 2;
2     int &r1 = i;            // 'normalna' referencja do l-wartości
3     int &&r2 = i;           // Źle - prawa strona to l-wartość
4     int &r3 = i + 2;        // Źle - prawa strona to r-wartość
5     const int &r4 = i + 2; // OK - const-referencja do r-wartości
6     int &&r5 = i + 2;       // OK
```

Linia 2 jest prawidłowa, bo zmienna i jest l-wartością a r1 jest „normalną" l-referencją. Jednakże linia 3 jest nielegalna, bo zmienna i jest l-wartością – w żadnym przypadku nie jest zmienną tymczasową! – i nie można jej związać z r-referencją. Linia 4 jest też nieprawidłowa, bo wyrażenie i+2 nie jest l-wartością, więc nie można go związać ze zwykłą referencją. Można jednak, jak wiemy, związać obiekt tymczasowy z referencją *do stałej*, jak w linii 5. W końcu ostatnia linia jest prawidłowa, bo r-referencję związujemy z obiektem tymczasowym.

Pracując z l- i r-wartościami warto pamiętać jakie funkcje (operatory) zwracają l-wartości, a jakie r-wartości. L-wartości są zwracane przez operator przypisania, indeksowania, dereferencji, *prefiksowej* inkrementacji i dekrementacji (ale nie postfiksowej).

Wyniki takich operacji mogą być związane z normalną, l-referencją. Z drugiej strony, r-wartości są zwracane przez operatory arytmetyczne, porównania, bitowe, *postfiksową* inkrementację/dekrementację – zwracane wartości mogą być związane z r-referencją.

R-referencje są przede wszystkim używane jako parametry funkcji. Gdy taki parametr jest zadeklarowany jako r-referencja, w wywołaniu, jako odpowiadający temu parametrowi argument, musi pojawić się r-wartość, czyli obiekt tymczasowy. Wiemy zatem, że za chwilę on „zniknie" i nikt nie będzie miał do niego dostępu. Jest więc bezpiecznie przejąć („ukraść") jego zasoby bez konieczności ich kopiowania. Typowym przykładem może być pole wskaźnikowe wskazujące, na przykład, na tablicę, która logicznie należy do obiektu, ale fizycznie jest poza nim, gdzieś na stercie. Normalnie w takich sytuacjach musieliśmy definiować konstruktor kopiujący, operator przypisania oraz destruktor, żeby kopiować nie wskaźniki, ale tę tablicę, alokując oczywiście za każdym razem pamięć na nią (i zwalniając ją w destruktorze). Jeśli jednak wiemy, że „oryginał" (w konstruktorze kopiującym czy operatorze przypisania) nigdy już nie będzie dla nikogo dostępny, *możemy* przekopiować tylko wskaźnik — musimy tylko zadbać, aby obiekt tymczasowy pozostawić w stanie pozwalającym na jego usunięcie bez żadnych złych skutków. Mówimy wtedy, że nasza klasa wspiera **semantykę przenoszenia**.

Może się zdarzyć, że mamy l-wartość, którą, na przykład, chcemy przekazać jako argument do funkcji; wiemy jednak, że tej zmiennej już nie będziemy więcej potrzebować. W takich sytuacjach możemy „poprosić" kompilator, żeby potraktował naszą l-wartość jak r-wartość i pozwolił na jej przeniesienie i pozostawienie w stanie co prawda nieokreślonym, ale legalnym i usuwalnym. Może to uczynić kopiowanie zasobów zbędnym i prowadzić do bardziej efektywnego kodu. Taką konwersję l-wartości do r-wartości wykonuje funkcja **std::move** (z nagłówka ***utility***).

Jak więc zapewnić, aby nasza klasa wspierała semantykę przenoszenia?
Musimy zdefiniować konstruktor przenoszący z parametrem zadeklarowanym jako r-referencja. Oczywiście *nie* do stałej, bo przecież właśnie zamierzamy zmodyfikować otrzymany obiekt tymczasowy — chcemy „zawłaszczyć" jego zasoby (poprzez kopiowanie wskaźników) i pozbawić go kontroli nad zawłaszczonymi zasobami (poprzez „wyzerowanie" wskaźników). W analogiczny sposób przeciążamy **przenoszący operator przypisania**.

W poniższym przykładzie definiujemy klasę **Arr** która jest „opakowaniem" zwykłej tablicy liczb całkowitych. Jedynymi polami tej klasy są wymiar tablicy i *wskaźnik* na nią — sama tablica jest tu „zasobem", który logicznie należy do obiektu, ale fizycznie jest alokowany gdzieś na stercie.
W linii ① definiujemy „normalny" konstruktor, a w linii ② konstruktor kopiujący. Pobiera on obiekt tego samego typu, którego nie może oczywiście zmienić — aby zapewnić, by nowo tworzony obiekt i obiekt przesłany do konstruktora zachowały niezależność, musimy zaalokować tablicę i przekopiować elementy z tablicy należącej do przesłanego obiektu do tablicy należącej do *tego* obiektu

P164: *rmoveassign.cpp* Semantyka przenoszenia

```
1  #include <cstring>
2  #include <iostream>
3  #include <utility>   // move
4
5  using std::cout; using std::endl;
6
7  class Arr {
8      size_t size;
9      int*    arr;
10 public:
11     Arr(size_t s, const int* a)                    ①
12         : size(s),
13           arr(static_cast<int*>(
14               std::memcpy(new int[size],a,
15                           size*sizeof(int))))
16     {
17         cout << "ctor from array\n";
18     }
19     Arr(const Arr& other)                          ②
20         : size(other.size),
21           arr(static_cast<int*>(
22               std::memcpy(new int[size],other.arr,
23                           size*sizeof(int))))
24     {
25         cout << "copy-ctor\n";
26     }
27     Arr(Arr&& other) noexcept                      ③
28         : size(other.size), arr(other.arr)
29     {
30         other.size = 0;
31         other.arr = nullptr;
32         cout << "move-ctor\n";
33     }
34     Arr& operator=(const Arr& other) {             ④
35         if (this == &other) return *this;
36         int* a = new int[other.size];
37         memcpy(a,other.arr,other.size*sizeof(int));
38         delete [] arr;
39         size = other.size;
40         arr = a;
41         cout << "copy-assign\n";
42         return *this;
43     }
44     Arr& operator=(Arr&& other) noexcept {         ⑤
```

```
45          if (this == &other) return *this;
46          delete [] arr;
47          size = other.size;
48          arr  = other.arr;
49          other.size = 0;
50          other.arr  = nullptr;
51          cout << "move-assign\n";
52          return *this;
53      }
54      ~Arr() {
55          delete [] arr;
56      }
57      friend std::ostream& operator<<(std::ostream& str,
58                                      const Arr& a) {
59          if (a.size == 0) return cout << "Empty";
60          str << "[ ";
61          for (size_t i = 0; i < a.size; ++i)
62              str << a.arr[i] << " ";
63          return str << "]";
64      }
65  };
66
67  Arr replicate(Arr a) {                                    ⑥
68      cout << "In replicate\n";
69      return a;
70  }
71
72  int main() {
73      cout << "**** 0 ****\n";
74      int a[]{1,2,3,4};
75      Arr arr(std::size(a),a);
76      cout << "arr : " << arr  << endl;
77
78      cout << "**** 1 ****\n";
79      Arr arr1 = replicate(arr);
80      cout << "arr1: " << arr1 << endl;
81      cout << "arr : " << arr  << endl;
82
83      cout << "\n**** 2 ****\n";
84      arr = arr1;
85      cout << "arr : " << arr  << endl;
86      cout << "arr1: " << arr1 << endl;
87
88      cout << "\n**** 3 ****\n";
89      Arr arr2 = replicate(std::move(arr));
90      cout << "arr2: " << arr2 << endl;
```

```
91      cout << "arr : " << arr  << endl;
92
93      cout << "\n**** 4 ****\n";
94      arr  = replicate(std::move(arr2));
95      cout << "arr : " << arr  << endl;
96      cout << "arr2: " << arr2 << endl;
97
98      cout << "\n**** 5 ****\n";
99      arr2 = std::move(arr);
100     cout << "arr2: " << arr2 << endl;
101     cout << "arr : " << arr  << endl;
102 }
```

W linii ③ definiujemy konstruktor przenoszący, który będzie użyty jeśli obiekt other jest tymczasowy. Konstruktor po prostu kopiuje oba pola, w tym wskaźnik. Teraz wskaźnik w *tym* obiekcie wskazuje na dokładnie tę samą tablicę, co wskaźnik w obiekcie other. Nie ma żadnej alokacji pamięci, żadnego kopiowania elementów tablicy. Musimy tylko zadbać, aby destruktor obiektu other nie zwolnił tablicy, którą właśnie „zawłaszczyliśmy"! W tym celu „zerujemy" wskaźnik w obiekcie other, tak, żeby **delete** w destruktorze niczego nie usuwało.
W podobny sposób implementujemy (⑤) operator przenoszącego przypisania.

Zauważmy, że obie funkcje przenoszące (Konstruktor i operator przypisania) są zadeklarowane jako **noexcept**: w ten sposób „obiecujemy", że nie zgłoszą żadnych wyjątków — rzeczywiście, nie powinny, bo tylko kopiują zmienne typów prostych. To jest istotne, bo wiele funkcji z Biblioteki Standardowej nie skorzysta z możliwości przenoszenia (tracąc wynikające stąd korzyści) jeśli odpowiednie funkcje nie zapewniają, że nie zgłoszą wyjątków.
Zauważmy też funkcję **replicate** (⑥): pobiera ona i zaraz zwraca obiekty typu **Arr**.

Przeanalizujmy więc działanie programu.
***** 0 *****: tworzymy obiekt typu **Arr** używając pierwszego konstruktora i wypisujemy go — dostajemy wydruk

```
**** 0 ****
ctor from array
arr : [ 1 2 3 4 ]
```

***** 1 *****: teraz wołamy fukcję **replicate** przekazując arr przez wartość. Ponieważ arr jest l-wartością, do wykonania kopii, która ma zostać położona stosie wykorzystany będzie zwykły konstruktor kopiujący. Funkcja zwraca jednak przez wartość, a więc obiekt tymczasowy, który będzie użyty do zainicjowania obiektu arr1 — wywołany zatem zostanie konstruktor przenoszący

```
**** 1 ****
copy-ctor
In replicate
move-ctor
```

```
arr1: [ 1 2 3 4 ]
arr : [ 1 2 3 4 ]
```

***** 2 *****: w przypisaniu `arr=arr1` obiekt po prawej jest l-wartością, zatem użyte będzie normalne przypisanie kopiujące

```
**** 2 ****
copy-assign
arr : [ 1 2 3 4 ]
arr1: [ 1 2 3 4 ]
```

***** 3 *****: teraz przekazujemy do funkcji **replicate** obiekt arr, ale jako r-wartość (skonwertowaną przez funkcję **std::move**). Zatem konstruktor przenoszący zostanie zastosowany do wykonania kopii, która będzie położona na stosie. Zwrócony przez funkcję obiekt tymczasowy użyty jest do zainicjowania obiektu arr2 — znów więc zastosowany będzie konstruktor przenoszący. Zauważmy, że obiekt arr został „wyzerowany"!

```
**** 3 ****
move-ctor
In replicate
move-ctor
arr2: [ 1 2 3 4 ]
arr : Empty
```

***** 4 *****: teraz przekazujemy arr2 jako r-wartość, a tymczasowy obiekt zwracany przypisujemy do istniejącego obiektu arr. Tak więc konstruktor przenoszący będzie użyty do utworzenia obiektu tymczasowego i przenoszący operator przypisania do przypisania na arr; zmienna arr2 będzie wyzerowana

```
**** 4 ****
move-ctor
In replicate
move-ctor
move-assign
arr : [ 1 2 3 4 ]
arr2: Empty
```

***** 5 *****: tu przypisujemy arr zrzutowane na r-wartość (przez **move**) na arr2; przenoszące przypisanie będzie użyte i arr będzie wyzerowane

```
**** 5 ****
move-assign
arr2: [ 1 2 3 4 ]
arr : Empty
```

Jak pamiętamy, konstruktor kopiujący i operator kopiującego przypisania są tworzone automatycznie przez kompilator (jeśli nie są **delete**d). Sytuacja z odpowiednimi

funkcjami przenoszącymi jest inna: jeśli klasa definiuje konstruktor kopiujący i/lub kopiujący operator przypisania i/lub destruktor, to konstruktor i przypisanie przenoszące nie będą utworzone, a wtedy kiedy są potrzebne, będzie użyta odpowiednia funkcja kopiująca.

Jeśli klasa nie definiuje swoich składowych kopiujących – konstruktor kopiujący, operator kopiującego przypisania, destruktor — kompilator utworzy składowe przenoszące jeśli tylko wszystkie pola mogą być przeniesione: pola typów prostych mogą (bo przenoszenie jest równoważne kopiowaniu) a pola typów obiektowych nie zawsze (na szczęście **string**i mogą).

Jeśli jednak klasa definiuje konstruktor przenoszący i/lub przenoszący operator przypisania, to „zwykłe", kopiujące wersje tych operacji będą oznaczone jako **delete**d — sami je musimy zdefiniować, jeśli są potrzebne.

Ogólnie, gdy choć jedna z funkcji kontrolujących kopiowanie/przenoszenie powinna mieć niedomyślną implementację, powinniśmy zdefiniować wszystkie pięć (konstruktory kopiujący i przenoszący, kopiujące i przenoszące operatory przypisania, oraz destruktor).

Jest nawet możliwe przeciążanie metod w taki sposób, że odpowiednia wersja zostanie wybrana na podstawie tego, czy obiekt na rzecz którego ją wywołujemy jest tymczasowy, czy nie. Przeciążenie dla wywołań na l-wartościach jest zaznaczony pojedynczym znakiem '&' za listą parametrów, a takie dla wywołań na r-wartościach — podwójnym znakiem '&', jak w przykładzie poniżej

P165: *RrefMet.cpp* Przeciążanie metod dla wywołań na l- i r-wartościach

```
#include <iostream>

struct X {
    void fun()  & { std::cout << "L-value\n"; }
    void fun() && { std::cout << "R-value\n"; }
};

int main() {
    X x{};
    x.fun();     // wywołanie fun() na l-wartości
    X{}.fun();   // wywołanie fun() na r-wartości
}
```

który drukuje

```
L-value
R-value
```

19.6 Inteligentne wskaźniki

Alokacja i dealokacja pamięci przez **new**/**delete** wymaga uwagi i jest bardzo podatna na błędy. Nowe cechy C++, w szczególności r-wartości i semantyka przenoszenia, umożliwiły znacznie ulepszoną implementację tak zwanych inteligentnych wskaźników.

Są to obiekty klas konkretyzowanych z szablonów **shared_ptr** oraz **unique_ptr**. Wewnętrznie zawierają one wskaźniki do obiektów (w zasadzie dowolnych typów). Dzięki odpowiednim przeciążeniom operatorów mogą one składniowo i semantycznie zachowywać się (do pewnego stopnia, dotyczy to w szczególności wskaźników typu **unique_ptr**) jak wskaźniki do obiektów którymi „zawiadują".

19.6.1 Inteligentne wskaźniki typu *unique_ptr*

Obiekty typu **unique_ptr** (nazywajmy je dalej u-wskaźnikami) są z założenia jedynymi „właścicielami" zawiadywanymi obiektami. Gdy są niszczone, zwalniane, albo zaczynają zawiadywać innym obiektem, obiekt przez nie zawiadywany też jest usuwany i są zwalniane związane z nim zasoby. Dlatego u-wskaźniki nie mogą być kopiowane ani podlegać kopiującym przypisaniom — doprowadziłoby to bowiem do sytuacji, gdy dwa obiekty zawiadują tym samym obiektem. Mogą jednak być przenoszone. Ten, z którego przenosimy traci „posiadanie" obiektu (jest „zerowany"), podczas gdy ten do którego przenosimy przejmuje posiadanie i odpowiedzialność za zarządzane zasoby (i ewentualnie zwalnia te, którymi zawiadywał wcześniej).

U-wskaźniki mogą być tworzone na kilka sposobów, na przykład:

```
std::unique_ptr<int>    empty; // holds nullptr
std::unique_ptr<int>    pi(new int(1));
std::unique_ptr<Person> pp(new Person("Mary", 2001));
std::unique_ptr<int[]>  pa(new int[4]{1,2,2}); // array
*pi = 21;
pp->setName("Kate");
pa[2] = 3;
```

Zauważmy, że pa reprezentuje wskaźnik na tablicę, wobec tego jest dla niego przeciążony operator indeksowania (**operator[]**), za pomocą którego możemy mieć dostęp do poszczególnych elementów; za to operator dereferencji * i dostępu do składowych -> nie jest dla u-wskaźników tablicowych określony. Wskaźniki nietablicowe jednakże mogą być używane składniowo i semantycznie jak zwykłe wskaźniki, choć fakt, że nie mogą być kopiowane i przypisywane powoduje, że nie mają pełnej ani semantyki wartości ani wskaźnikowej. W każdym razie, gdy zawiadywany obiekt ma być zniszczony, odpowiednia wersja operatora **delete** będzie automatycznie użyta — **delete[]** dla tablic i **delete** dla obiektów nietablicowych.

Jeśli żadna z tych form nie jest właściwa, użytkownik może przekazać własny tak zwany **deleter**. Musi to być funkcja (obiekt funkcyjny) niczego nie zwracająca a pobierająca (zwykły) wskaźnik odpowiedniego typu, jak w przykładzie poniżej. Zauważmy, że typ własnego deletera musi być częścią typu samego u-wskaźnika — musimy go przekazać, aby właściwie skonkretyzować szablon **unique_pointer**! W pro-

gramie poniżej jako deletera używamy obiektu funkcyjnego własnej klasy **Del** w pierwszym przypadku, a lambdy w drugim

P166: ***delunique.cpp*** Własny deleter wskaźnika typu *unique_ptr*

```
#include <functional>
#include <iostream>
#include <memory>
#include <string>

using std::unique_ptr; using std::string;

template <typename T>
struct Del {
    void operator()(T* p) {
        std::cout << "Del deleting " << *p << '\n';
        delete p;
    }
};

int main() {
    {
        unique_ptr<string, Del<string>>
            us{new string{"Hello"}, Del<string>{}};
    }
    std::cout << "us is now out of scope\n";

    {
        unique_ptr<double, std::function<void(double*)>>
            ud{new double{7.5},
               [](double* p) {
                   std::cout << "Deleting " << *p << '\n';
                   delete p;
               }
            };
    }
    std::cout << "ud is now out of scope\n";
}
```

Program drukuje

```
Del deleting Hello
us is now out of scope
Deleting 7.5
ud is now out of scope
```

Obiekty typu **unique_ptr** można też tworzyć za pomocą funkcji **make_unique**.

Obiekt, którym wskaźnik ma zawiadywać jest wtedy tworzony przez tę funkcję, a przekazujemy do niej tylko inicjujące wartości albo argumenty dla konstruktora. Jeśli tworzymy tablicę, to przekazać można tylko jej wymiar — nie ma sposobu, aby tę tablicę od razu zainicjować przekazanymi wartościami. Wadą tej funkcji jest też to, że nie da się wtedy przekazać deletera — jest to jednak konieczne raczej rzadko.

```
std::unique_ptr<int>    pi = std::make_unique<int>(19);
std::unique_ptr<Person> pp =
                  std::make_unique<Person>("Mary", 2001);
std::unique_ptr<int[]>  pa = std::make_unique<int[]>(3);
for (int i = 0; i < 3; ++i) pa[i] = i;
```

U-wskaźniki nie mogą być kopiowane ani podlegać kopiującym przypisaniom, gdyż pogwałciłoby to wyłączność posiadania. Mogą jednak być przenoszone. Poniższy program

P167: ***ownunique.cpp*** Przenoszenie posiadania

```
#include <iostream>
#include <memory>      // smart pointers
#include <string>
#include <utility>     // move

using std::unique_ptr; using std::string;

template <typename T>
struct Del {
    void operator()(T* p) {
        std::cout << "Del deleting " << *p << '\n';
        delete p;
    }
};

template <typename T>
void print(const T* p) {
    if (p) std::cout << *p << " ";
    else   std::cout << "null ";
}

int main() {
    unique_ptr<string, Del<string>>
            p1{new string{"abcde"}, Del<string>{}},
            p2{new string{"vwxyz"}, Del<string>{}};

    print(p1.get()); print(p2.get()); std::cout << '\n'; ①
    std::cout << "Now moving\n";
    p1 = std::move(p2);                                  ②
```

```
30      std::cout << "After moving\n";
31      print(p1.get()); print(p2.get()); std::cout << '\n'; ③
32      std::cout << "Exiting from main\n";
33  }
```

drukuje

```
abcde vwxyz
Now moving
Del deleting abcde
After moving
vwxyz null
Exiting from main
Del deleting vwxyz
```

Zauważmy, że funkcja **get** (linie ① i ③) zwraca „goły” wskaźnik zawiadywany przez u-wskaźnik: nie powinniśmy go przypisywać do żadnej zmiennej, gdyż łatwo wtedy by było naruszyć zasadę jednego właściciela.
Po przenoszącym przypisaniu (linia ②), jak możemy się przekonać z wydruku, obiekt zawiadywany przez p1 jest niszczony i p1 przejmuje własność obiektu zawiadywanego wcześniej przez p2 — ten z kolei jest „zerowany”.

Inne ważne metody u-wskaźników to między innymi (**T** oznacza tu typ obiektu zawiadywanego przez wskaźnik):

T* release() — zwalnia zasoby zawiadywane przez *ten* wskaźnik, który jest „zerowany”; funkcja zwraca goły wskaźnik, którym zawiadywała (lub **nullptr** jeśli wskaźnik był pusty) – teraz użytkownik przejmuje całkowitą odpowiedzialność za wskazywany obiekt, w szczególności za jego usunięcie w odpowiednim czasie;

T* reset(a_pointer = nullptr) — usuwa zawiadywany obiekt (jeśli był) i przejmuje "pod opiekę” przekazany wskaźnik (być może, lub domyślnie, **nullptr**).

Metoda **reset** zilustrowana jest w następującym programie:

P168: ***uniquereset.cpp*** Resetowanie wskaźników *unique_ptr*

```
#include <iostream>
#include <memory>

using std::unique_ptr; using std::ostream; using std::cout;

template <typename T>
struct Del {
    void operator()(T* p) {
        cout << "Del deleting " << *p << '\n';
        delete p;
    }
};
```

```
13
14 struct Klazz {
15     Klazz()  { cout << "Ctor Klazz\n"; }
16     ~Klazz() { cout << "Dtor Klazz\n"; }
17     friend ostream& operator<<(ostream& s, const Klazz& k) {
18         return s << "object of type Klazz";
19     }
20 };
21
22 int main() {
23     std::cout << "Creating  u-pointer\n";
24     std::unique_ptr<Klazz, Del<Klazz>>
25                             p(new Klazz{}, Del<Klazz>{});
26     std::cout << "Resetting u-pointer\n";
27     p.reset(new Klazz{});
28     std::cout << "Releasing and deleting\n";
29     p.reset(); // or reset(nullptr)
30 }
```

Jak widzimy, gdy u-wskaźnik jest resetowany, nowy obiekt jest tworzony najpierw a dopiero potem stary jest niszczony przez wywołanie deletera; dla typów obiektowych będzie jeszcze, oczywiście, wywołany destruktor.

U-wskaźniki są często stosowane jako elementy kolekcji. Następujący program demonstruje jak wypełnić wektor u-wskaźnikami. Zwróćmy uwagę, że inteligentne wskaźniki typu klas bazowych mogą odnosić się do obiektów klas pochodnych. Wywołania polimorficzne działają zgodnie z przewidywaniami, jak dla zwykłych wskaźników. Tu umieszczamy w wektorze jeden wskaźnik do obiektu klasy bazowej **B** i trzy do obiektów klasy pochodnej **D**:

P169: ***uniquederiv.cpp*** U-pointers and polymorphism

```
1 #include <iostream>
2 #include <vector>
3 #include <memory>
4
5 struct B {
6     virtual void f() { std::cout << "f from B\n"; }
7     virtual ~B() { }
8 };
9 struct D : B {
10     D() { std::cout << "Ctor D\n"; }
11     void f() override { std::cout << "f from D\n"; }
12     ~D(){ std::cout << "Dtor D\n"; }
13 };
14
15 int main() {
```

```
16      {
17          std::vector<std::unique_ptr<B>> vec;
18          vec.push_back(std::make_unique<B>());
19          vec.push_back(std::make_unique<D>());
20          vec.emplace_back(std::make_unique<D>());
21          std::unique_ptr<B> d{new D};
22          vec.push_back(std::move(d));
23          for (const auto& up : vec) up->f();
24      }
25      std::cout << "now vec is out of scope\n";
26  }
```

Zauważmy, że gdy wektor wychodzi z zakresu, wszystkie jego elementy (u-wskaźniki) zwalniają zawiadywane przez siebie zasoby, tak, że nie powstaje żaden wyciek pamięci; program drukuje

```
Ctor D
Ctor D
Ctor D
f from B
f from D
f from D
f from D
Dtor D
Dtor D
Dtor D
vec out of scope
```

Tak jak zwykłe wskaźniki, tak też u-wskaźniki mogą być używane w kontekście wymagającym wartości logicznej; wskaźnik pusty (czyli spełniający `p.get() == nullptr`) jest interpretowany jako **false**, w przeciwnym wypadku jako **true**.

19.6.2 Inteligentne wskaźniki typu *shared_ptr*

Inteligentne wskaźniki typu **shared_ptr** (s-wskaźniki) dzielą własność zasobu reprezentowanego przez zwykły wskaźnik. Są wyposażone w mechanizm zliczania referencji — tworzone są specjalne struktury danych z licznikiem, pozwalającym na zliczanie s-wskaźników odnoszących się do tego samego zasobu. Kiedy taki wskaźnik wychodzi z zakresu licznik jest obniżany o jeden, a kiedy osiąga wartość zero zasób jest zwalniany. Podobnie, po przypisaniu takich wskaźników, `p = q`, licznik związany z zasobem zawiadywanym przez `p` jest zmniejszany (bo `p` już nie będzie się do niego odnosić), natomiast ten związany z zasobem zawiadywanym przez `q` jest zwiększany, bo teraz również `p` do niego się odnosi. Możemy poznać stan liczników wywołując metodę **use_count**.

Niektóre z tych własności s-wskaźników są zilustrowane następującym programem:

P170: *sharedcount.cpp* Liczniki związane z zasobami zawiadywanymi przez s-wskaźniki

```
#include <iostream>
#include <memory>
using std::shared_ptr; using std::cout; using std::ostream;

class Klazz {
    char c;
public:
    Klazz(char c)
        : c{c} { cout << "Ctor " << c << '\n'; }
    ~Klazz()   { cout << "Dtor " << c << '\n'; }
    friend ostream& operator<<(ostream& s, const Klazz& k) {
        return s << k.c;
    }
};

void f(shared_ptr<Klazz> p) {
    cout << "In f: p=" << *p << ", count="
         << p.use_count() << '\n';
}

int main() {
    shared_ptr<Klazz> p = std::make_shared<Klazz>('A');
    shared_ptr<Klazz> q{new Klazz{'B'}};
    cout << "p=" << *p << ", count=" << p.use_count() << '\n';
    cout << "q=" << *q << ", count=" << q.use_count() << '\n';
    f(p);
    cout << "p=" << *p << ", count=" << p.use_count() << '\n';
    cout << "Now assigning p = q\n";
    p = q;
    cout << "After assignment\n";
    cout << "p=" << *p << ", count=" << p.use_count() << '\n';
    cout << "q=" << *q << ", count=" << q.use_count() << '\n';
    cout << "Exiting from main\n";
}
```

który drukuje

```
Ctor A
Ctor B
p=A, count=1
q=B, count=1
In f: p=A, count=2
p=A, count=1
Now assigning p = q
```

```
Dtor A
After assignment
p=B, count=2
q=B, count=2
Exiting from main
Dtor B
```

Jak widzimy, można utworzyć s-wskaźnik posyłając zwykły wskaźnik do konstruktora. Konstruktor domyślny tworzy wskaźnik pusty, podobnie jak w przypadku u-wskaźników. Dla s-wskaźników istnieje też, analogiczna do **make_unique**, funkcja **make_shared**.

Po utworzeniu każda ze zmiennych p i q odnosi się do innego obiektu — oba liczniki są zatem 1. Kiedy posyłamy p przez wartość do funkcji **f**, musi zostać wykonana kopia obiektu p — ta kopia wewnątrz funkcji również odnosi się do obiektu 'A', tak więc licznik wynosi tu 2. Po wyjściu z funkcji, kopia jest usuwana ze stosu i licznik odwołań do 'A' znów ma wartość 1.
Teraz przypisujemy q do p: zmienna p odnosi się do obiektu 'A', licznik jest obniżany i osiąga wartość zero, tak więc 'A' jest usuwane. Teraz zarówno p jak i q odnoszą się do obiektu 'B', więc licznik wynosi 2. Po wyjściu z funkcji **main** najpierw usuwane jest ze stosu p — licznik spada do 1, a następnie q — teraz licznik osiąga zero i obiekt 'B' jest usuwany.

Dla s-wskaźników można, podobnie jak dla u-wskaźników, definiować własne deletery. Inaczej niż dla u-wskaźników, typ deletera nie jest częścią typu wskaźnika — po prostu przesyłamy deleter jako dodatkowy argument do konstruktora. Przed wersją C++17 nawet jeśli zarządzanym zasobem była tablica, domyślnie stosowany był deleter *nie* wywołujący **delete[]** jak powinien, tylko **delete** — trzeba zatem było definiować i przesyłać własne deletery, jak w przykładzie poniżej:

P171: ***shareddelete.cpp*** Deletery dla s-wskaźników tablicowych

```
#include <iostream>
#include <memory>
using std::shared_ptr;

template< typename T >
struct arrdel {
    void operator ()(T const *p) { delete[] p; }
};

int main() {
    shared_ptr<int> sp(new int(1));

      // pointer to int[] array - custom deleter
    shared_ptr<int> p1(new int[10], arrdel<int>());
      // ... or lambda
    shared_ptr<int> p2(new int[10'000'000],
                       [](int *p) { delete[] p; });
```

```
18        // ... or the one from the library
19      shared_ptr<int> p3(new int[3]{1, 2, 3},
20                         std::default_delete<int[]>());
21      std::cout << p3.get()[2] << " " << *p3 << std::endl; ①
22
23        // since c++17 this will work
24      shared_ptr<int[]> p4(new int[3]{4, 5, 6});
25      std::cout << p4[2] << std::endl;
26  }
```

Nie był również dla takich wskaźników określony operator indeksowania (**[ ]**): dlatego nie można go było użyć w linii① powyższego programu. Jednak od wersji C++17 standardu s-wskaźniki tablicowe można tworzyć tak jak u-wskaźniki i używają one właściwego deletera domyślnie, bez potrzeby definiowania ich przez użytkownika. Program skompilowany kompilatorem wspierającym C++17, drukuje

```
3 1
6
```

Jak u-wskaźniki, tak i s-wskaźniki mogą być używane w kontekście logicznym (**false** jeśli są puste, **true** w przeciwnym przypadku). Zdefiniowana jest też dla nich metoda **get** oraz przeciążone są operatory ***** i ->.

— 20 —

Jeszcze o konwersjach

Wielokrotnie wspominaliśmy o przekształceniach typu, czyli konwersjach. W rozdziale 10.1 (str. 147) omówiliśmy te, które dokonywane są automatycznie pomiędzy typami wbudowanymi: teraz zajmiemy się konwersjami pomiędzy typami zdefiniowanymi w programie oraz konwersjami jawnymi, które sami narzucamy za pomocą odpowiedniej składni.

Podrozdziały:

20.1 Konwersje od i do typu definiowanego

Jeśli zdefiniowaliśmy klasę, a więc nowy typ danych (zmiennych), to często chcielibyśmy określić, w jaki sposób ma być dokonywana konwersja od obiektów innych typów *do* obiektów zdefiniowanej przez nas klasy. Z drugiej strony, chcielibyśmy czasem zdefiniować konwersje odwrotne: *od* obiektów naszej klasy do obiektów innych typów (wbudowanych, bibliotecznych lub też zdefiniowanych przez nas).

20.1.1 Konwersja *do* typu definiowanego

Przypuśćmy, że klasa **A** ma konstruktor, który może być wywołany z jednym argumentem typu **B**. A zatem albo jest to konstruktor jednoparametrowy z parametrem typu **B**, albo pierwszy parametr ma taki typ, a pozostałe parametry mają wartości domyślne. Typ **B** może być typem wbudowanym (**double**, **int**, ...), albo typem przez nas zdefiniowanym.

Załóżmy teraz, że użyjemy obiektu klasy **B** w kontekście, w którym wymagany jest obiekt typu **A**. Jeśli opisany wyżej konstruktor w klasie **A** istnieje, to dokonana zostanie konwersja **B** → **A** poprzez utworzenie nowego obiektu klasy **A** z dostarczeniem danego obiektu klasy **B** do konstruktora jako jedynego argumentu.

Na przykład, jeśli istnieje funkcja o parametrze typu **A**

```
void fun(A a) { ... }
```

a jej wywołanie ma postać `fun(4.25)`, to mamy do czynienia z niezgodnością typów, bo argument jest typu **double**. Błędu jednak *nie* będzie, jeśli klasa **A** ma konstruktor, który można wywołać z jednym argumentem typu **double**. Jeśli bowiem taki **konstruktor konwertujący** jest, to zostanie utworzony obiekt klasy **A** z użyciem tego konstruktora (do którego zostanie przesłana jako argument wartość 4.25). Tak utworzony obiekt zostanie następnie przesłany do funkcji **fun**.

A co będzie, jeśli tę samą funkcję wywołamy z argumentem całkowitym, na przykład `'fun(4)'`? Takie wywołanie *też* będzie wtedy prawidłowe! Nie ma, co prawda, bezpośredniej konwersji **int** → **A**, bo nie ma w klasie **A** konstruktora pobierającego jeden argument typu **int**. Istnieje jednak standardowa konwersja **int** → **double**, a zdefiniowaną mamy, poprzez odpowiedni konstruktor, konwersję **double** → **A**. Zatem za pomocą takiej dwustopniowej konstrukcji można skonwertować wartość całkowitą do obiektu klasy **A**: najpierw utworzona zostanie tymczasowa zmienna typu **double**, a z niej obiekt klasy **A**. W sekwencji kilku konwersji prowadzących do celu takich konwersji pośrednich może być nawet więcej. Ważne jest jednak, że

> tylko jedna z konwersji w tej sekwencji może być konwersją definiowaną przez użytkownika; pozostałe muszą być konwersjami standardowymi.

Gdybyśmy bowiem dopuścili dowolne sekwencje konwersji, to możliwości byłoby tak wiele, że trudno byłoby w nich wszystkich zorientować się samemu programiście; w szczególności, trudno byłoby nawet przewidzieć, jakie typy takimi sekwencjami konwersji można połączyć, a jakie nie.

Zresztą, możliwość użycia w sekwencji nawet jednej konwersji zdefiniowanej przez użytkownika może być niewygodna. Jest tak wtedy, gdy potrzebujemy w naszej klasie konstruktora jednoparametrowego, ale wcale nie chcemy, aby był on wykorzystywany niejawnie do konwersji. Takie konwersje mogą się bowiem pojawić w najmniej spodziewanych miejscach, gdzie po prostu nie przewidzieliśmy, że będą przez kompilator wygenerowane. Dlatego istnieje specjalne słowo kluczowe, **explicit**, które może być użyte jako modyfikator konstruktora. Jeśli go użyjemy, to konstruktor taki *nie* będzie nigdy wykorzystany jako konstruktor konwertujący, a tylko wtedy, gdy służy swemu właściwemu celowi, to znaczy gdy jawnie tworzymy obiekty klasy.

Przyjrzyjmy się programowi:

P172: ***convto.cpp*** Konwersja do klasy definiowanej

```
1 #include <iostream>
2 using namespace std;
3
4 struct Point {
5     int x, y;
6     Point(int x = 0, int y = 0) : x(x), y(y) { }
7 };
```

```
8
9  struct Segment {
10     Point A, B;
11     // explicit
12     Segment(Point A = Point(), Point B = Point())
13         : A(A), B(B)
14     { }
15 };
16
17 void showPoint(Point A) {
18     cout << "Point[" << A.x << "," << A.y << "]";
19 }
20
21 void showSegment(Segment AB) {
22     cout << "Segment: ";
23     showPoint(AB.A);
24     cout << "--";
25     showPoint(AB.B);
26     cout << endl;
27 }
28
29 int main() {
30     int k = 7;
31     showPoint(k);
32
33     cout << endl;
34
35     Point A(1,1);
36     showSegment(A);
37     // showSegment(k);
38 }
```

Zdefiniowaliśmy tu dwie klasy: **Point** i **Segment**. Oba posiadają konstruktory, które mogą być wywołane z jednym argumentem: klasa **Point** z argumentem typu **int**, a klasa **Segment** z argumentem typu **Point** (oba konstruktory są wieloparametrowe, ale dzięki zastosowaniu parametrów domyślnych mogą być wywołane z jednym argumentem — patrz rozdz. 11.4 na str. 163.

Następnie zdefiniowane są dwie funkcje, **showPoint** i **showSegment**, których parametry są typu **Point** i **Segment**.

Przyjrzyjmy się teraz funkcji **main**. W linii 31 wywołujemy funkcję **showPoint** z argumentem typu **int**. Funkcji o takiej nazwie i takim typie parametru *nie ma*. Jest taka funkcja, tyle że z parametrem typu **Point**. Kompilator sprawdzi zatem, czy istnieje konwersja **int** → **Point**, to znaczy, czy istnieje konstruktor w klasie **Point**, który można wywołać z jednym argumentem typu **int**. Taki konstruktor istnieje, zatem konwersja zostanie dokonana: obiekt klasy **Point** na podstawie wartości całkowitej zostanie utworzony i wysłany do funkcji **showPoint**, jak o tym świadczy pierwsza

linia wydruku:

```
Point[7,0]
Segment: Point[1,1]--Point[0,0]
```

W linii 35 tworzymy obiekt A klasy **Point** i posyłamy go do funkcji **showSegment**. Znów dokonana musi być konwersja, aby to wywołanie mogło być prawidłowe. Funkcja oczekuje argumentu typu **Segment**, zatem obiekt tej klasy zostanie utworzony z obiektu A za pomocą wywołania konstruktora klasy **Segment** z wartością A jako argumentem. Świadczy o tym druga linia wydruku.

Zauważmy, że wywołanie z wykomentowanej linii 37 byłoby nieprawidłowe. Wymagałoby konwersji dwustopniowej: najpierw **int** → **Point**, potem **Point** → **Segment**. A zatem użyte musiałyby być dwie konwersje definiowane w programie: to jest jednak niemożliwe.

Spróbujmy teraz uaktywnić wykomentowaną linię 11. Konstruktor klasy **Segment** jest teraz zdefiniowany z modyfikatorem **explicit**. Nie może zatem pełnić roli konstruktora konwertującego **Point** → **Segment**. Wywołanie z linii 36 staje się teraz nieprawidłowe, bo wymaga właśnie takiej konwersji. Taki program, z „odkomentowaną" linią 11, jest wobec tego błędny:

```
cpp> g++ -Wall -pedantic-errors convto.cpp
convto.cpp: In function `int main()':
convto.cpp:36: error: conversion from `Point' to
       non-scalar type `Segment' requested
```

Jako drugi przykład rozpatrzmy klasę **Modulo**, której użyliśmy już w programach ***modsev.cpp*** (str. 388) i ***modsev1.cpp*** (str. 396). Teraz zauważmy, jak konstruktor konwertujący ułatwi nam przeciążenie operatora dodawania liczb typu **Modulo**.

P173: ***modcon.cpp*** Konwersje ułatwiające przeciążenia operatorów

```
1  #include <iostream>
2  using namespace std;
3
4  class Modulo {
5      int numb;
6  public:
7      static int modul;
8
9      Modulo() : numb(0) { }
10
11     Modulo(int numb) : numb(numb%modul) { }
12
13     friend Modulo operator+(Modulo,Modulo);
14     friend ostream& operator<<(ostream&,const Modulo&);
15 };
16 int Modulo::modul = 7;
```

```
17
18 Modulo operator+(Modulo m, Modulo n) {
19     return Modulo(m.numb + n.numb);
20 }
21
22 ostream& operator<<(ostream& str, const Modulo& m) {
23     return str << m.numb;
24 }
25
26 int main() {
27
28     Modulo m(5), n(6), k;
29
30     k = m + n;
31     cout << "m + n (mod " << Modulo::modul
32          << ") = "         << k << endl;
33
34     k = m + 6;
35     cout << "m + 6 (mod " << Modulo::modul
36          << ") = "         << k << endl;
37
38     k = 6 + m;
39     cout << "6 + m (mod " << Modulo::modul
40          << ") = "         << k << endl;
41 }
```

Tym razem operator dodawania obiektów klasy **Modulo** definiujemy poprzez zaprzyjaźnioną funkcję globalną, a nie jako metodę. Zauważmy, że zdefiniowaliśmy tylko jedną taką funkcję (linie 18-20): z oboma parametrami typu **Modulo**. Jak widać w funkcji **main**, używamy jej jednak do dodawania dwóch obiektów klasy **Modulo** (linia 30), do dodawania liczby typu **int** do obiektu klasy **Modulo** (linia 34), jak i do dodawania obiektu klasy **Modulo** do liczby (linia 38). We wszystkich przypadkach przeciążenie działa prawidłowo:

```
m + n (mod 7) = 4
m + 6 (mod 7) = 4
6 + m (mod 7) = 4
```

Ta ostatnia forma dodawania, *liczba+obiekt*, nie byłaby możliwa do zrealizowania przez przeciążenie operatora dodawania za pomocą metody, bo po lewej stronie mamy tu liczbę, a nie obiekt klasy. Dlaczego mogliśmy zdefiniować tylko jedną formę funkcji przeciążającej operator dodawania, tę z oboma parametrami typu **Modulo**? Było to możliwe dzięki konstruktorowi z linii 11, który jest konstruktorem konwertującym **int** → **Modulo**. Za jego pomocą argument całkowity zostanie przekonwertowany automatycznie do typu **Modulo** tam, gdzie konieczność takiej konwersji będzie wynikała z kontekstu, i to niezależnie od tego, czy liczba występuje po lewej, czy po prawej stronie operatora '+'.

20.1.2 Konwersja *od* typu definiowanego

Opisanym sposobem możemy przekształcać obiekty pewnej klasy (w szczególności typu wbudowanego) na obiekty definiowanej przez nas klasy. Można również „nauczyć" kompilator operacji odwrotnej: konwertowania obiektów definiowanej przez nas klasy na obiekty innego typu (w szczególności typu wbudowanego). Jest to jedyne wyjście, gdy klasa docelowa, czyli ta, *do* której ma nastąpić konwersja, jest dla nas niedostępna i nie możemy w niej dodefiniować konstruktora konwertującego (bo, na przykład, typem docelowym jest typ wbudowany w ogóle nie będący klasą, albo klasa docelowa pochodzi z biblioteki, której nie możemy lub nie chcemy modyfikować).

W takiej sytuacji w definiowanej przez nas klasie definiujemy **metodę konwertującą** (ang. *conversion method*). Jest to bezparametrowa metoda o nazwie '**operator Typ**', gdzie **Typ** jest nazwą typu (klasy) docelowego — może to być typ wbudowany, jak **int** czy **double**, a może też być to typ przez nas zdefiniowany.

Dla metod konwertujących, wyjątkowo, *nie* podaje się typu zwracanego, ale nie oznacza to, że metoda jest bezrezultatowa. Wartość zwracana musi mieć typ określony nazwą metody; musi zatem zawierać instrukcję **return** zwracającą wartość tego typu. Ponieważ jest to metoda, zawsze będzie działać na rzecz konkretnego obiektu: jej zadaniem jest „wyprodukowanie" obiektu odpowiedniego typu (do którego następuje konwersja) na podstawie obiektu, na rzecz którego działa. Oczywiście, nie powinna zmieniać obiektu źródłowego, a więc powinna być zadeklarowana jako **const**. Można też, podobnie jak konstruktory konwertujące, deklarować takie metody jako **explicit**: zapobiega to przypadkowym, niezamierzonym konwersjom, ale też powoduje konieczność stosowania konwersji jawnych tam, gdzie ich chcemy.

Rozważmy przykład, podobny do tego z programu *convto.cpp* (str. 440):

P174: *convfrom.cpp* Konwersja od klasy definiowanej

```
#include <iostream>
#include <cmath>
using std::cout; using std::endl;

struct Point {
    double x, y;
    Point(double x = 0, double y = 0) : x(x), y(y) { }
    operator double() const {          ①
        return std::sqrt(x*x+y*y);
    }
};

struct Segment {
    Point A, B;
    Segment(Point A = Point(), Point B = Point())
        : A(A), B(B)
    { }
    operator Point() const {           ②
        return Point( (A.x+B.x)/2, (A.y+B.y)/2 );
```

```
20      }
21  };
22
23  void showPoint(Point A) {
24      cout << "Point[" << A.x << "," << A.y << "]";
25  }
26
27  void showSegment(Segment AB) {
28      cout << "Segment: ";
29      showPoint(AB.A);
30      cout << "--";
31      showPoint(AB.B);
32      cout << endl;
33  }
34
35  void showDouble(double d) {
36      cout << "Double " << d;
37  }
38
39  int main() {
40      Point A(3,4);
41      showPoint(A);                          ③
42      cout << endl;
43      showDouble(A);                         ④
44
45      cout << endl;
46
47      Segment BC(Point(1,1),Point(3,3));
48      showSegment(BC);
49      showPoint(BC);                         ⑤
50
51      cout << endl;
52  }
```

Do klasy **Point** dodaliśmy tu metodę konwertującą **operator double()** (①). Zwraca ona odległość punktu od początku układu współrzędnych w postaci wartości typu **double** (w tym przypadku jest to wartość typu wbudowanego). Podobnie, do klasy **Segment** dodana została metoda konwertująca do klasy **Point** (②). Zwraca ona obiekt klasy **Point** reprezentujący geometryczny środek odcinka. Dodana też została funkcja **showDouble**, która wypisuje przekazaną przez argument liczbę typu **double**.

W linii ③ wywołujemy funkcję **showPoint** z argumentem typu **Point**, a więc typu zadeklarowanego parametru funkcji. Zaraz potem wywołujemy z tym samym argumentem funkcję **showDouble** (④). Funkcja ta spodziewa się argumentu typu **double**. Zatem potrzebna jest konwersja **Point** → **double**. Ponieważ taką konwersję zdefiniowaliśmy, wywołanie jest prawidłowe: przed przekazaniem do funkcji obiekt A klasy **Point** zostanie skonwertowany do typu **double**, o czym świadczy druga linia wydruku:

```
Point[3,4]
Double 5
Segment: Point[1,1]--Point[3,3]
Point[2,2]
```

Podobnie będzie w linii ⑤. Do funkcji **showPoint**, która ma parametr typu **Point**, przesyłamy obiekt klasy **Segment**. Zostanie zatem użyta metoda konwertująca klasy **Segment** (②) definiująca konwersję **Segment** → **Point**.

Zauważmy, że w drugim przypadku ten sam cel moglibyśmy osiągnąć definiując konstruktor przyjmujący jeden argument typu **Segment** w klasie **Point**. Nie było to jednak możliwe w przypadku poprzednim, gdyż nie da się zdefiniować konstruktora konwertującego w klasie **double** (przede wszystkim dlatego, że takiej klasy nie ma!).

Zauważmy na koniec, że metody konwertujące są dziedziczone i mogą być wirtualne — sens tego omówimy w jednym z następnych rozdziałów.

20.2 Konwersje jawne

W czystym C, tak jak w Javie, możemy zażądać jawnie konwersji od wartości jednego typu do wartości innego typu za pomocą **rzutowania**. Ma ono postać

```
(Typ) wyrazenie
```

gdzie **Typ** jest nazwą typu, a wyrazenie jest wyrażeniem o p-wartości innego typu. Wynikiem jest p-wartość typu **Typ**, reprezentująca wartość wyrażenia wyrazenie. W Javie tego typu rzutowania są bezpieczne: albo na etapie kompilacji, albo, jeśli to niemożliwe, na etapie wykonania zostanie sprawdzone, czy takie rzutowanie ma sens. W C taka forma rzutowania jest mniej bezpieczna; będzie ono wykonane „siłowo”, czasem zupełnie bezsensownie. Dlatego lepiej jest używać nowych operatorów, wprowadzonych w C++, które wykonują te konwersje w sposób bardziej kontrolowany. Wszystkie one mają postać

```
rodzaj_cast<Typ>(wyrazenie)
```

gdzie zamiast ’rodzaj’ należy wstawić static, dynamic, const lub reinterpret. Wynikiem będzie p-wartość typu **Typ** utworzona na podstawie wartości wyrażenia wyrazenie.

20.2.1 Konwersje uzmienniające

Konwersja uzmienniająca ma postać

```
const_cast<Typ>(wyrazenie)
```

Wyrażenie wyrazenie musi tu być tego samego typu co **Typ**, tylko z modyfikatorem **const** lub **volatile**. A zatem tego rodzaju konwersja usuwa „ustaloność” (lub „ulotność”) i może służyć *tylko* do tego celu. Z drugiej strony, tego samego efektu *nie*

można uzyskać za pomocą konwersji przy użyciu **static_cast**, **dynamic_cast** lub **reinterpret_cast**: kompilator uznałby taką konwersję **const Typ** → **Typ** za nielegalną. Rozpatrzmy przykład:

P175: *concast.cpp* Usuwanie stałości

```
1  #include <iostream>
2  using namespace std;
3
4  void changeFirst(char* str, char c) {
5      str[0]=c;
6  }
7
8  int main() {
9      const char name[] = "Jenny";
10     cout << name << endl;
11
12     // name[0]='K';
13
14     changeFirst(const_cast<char*>(name),'K');
15
16     // changeFirst(name,'K');
17
18     cout << name << endl;
19 }
```

Funkcja **changeFirst** zmienia pierwszą literę przekazanego jej C-napisu. W programie głównym tworzymy *ustalony* napis name o zawartości "Jenny" (linia 9). Próba jego zmiany w wykomentowanej linii 12 skończyłaby się przerwaniem kompilacji. W linii 14 wysyłamy ten napis do funkcji **changeFirst**, konwertując argument tak, aby usunąć atrybut stałości. Jak widać z wydruku

```
Jenny
Kenny
```

po tej operacji zmiana ustalonego napisu powiodła się. Zauważmy też, że bez konwersji, a więc tak jak w wykomentowanej linii 16, wywołać tej funkcji nie byłoby można, bo name jest C-napisem ustalonym, a funkcja **changeFirst** nie „obiecuje", poprzez deklarację typu parametru jako **const**, że napisu przekazanego jako argument nie zmieni (czego zresztą obiecać nie może, bo właśnie ten napis zmienia).

Jeśli deklarujemy zmienne ustalone, to robimy to właśnie po to, aby ich nie można było zmieniać. Zatem użycie konwersji uzmienniającej świadczy o jakiejś niekonsekwencji w programie. Powinno być zatem stosowane tylko w wyjątkowych wypadkach.

20.2.2 Konwersje statyczne

Konwersja statyczna ma postać

```
static_cast<Typ>(wyrazenie)
```

i dokonuje jawnego przekształcenia typu, sprawdzając, czy jest to przekształcenie dopuszczalne. Sprawdzenie odbywa się podczas kompilacji. Często użycie tego operatora jest właściwie zbędne, bo konwersja i tak zostanie dokonana. Jeśli jest to jednak konwersja, w której może wystąpić utrata informacji, to kompilator zwykle ostrzega nas przed jej użyciem. Stosując jawną konwersję statyczną, unikamy tego rodzaju ostrzeżeń kompilatora. Na przykład kompilacja prawidłowego fragmentu kodu

```
double x = 4;
int i = x;
```

spowoduje wysłanie ostrzeżeń kompilatora

```
d.cpp:6: warning: initialization to `int' from `double'
d.cpp:6: warning: argument to `int' from `double'
```

których możemy uniknąć jawnie dokonując konwersji:

```
double x = 4;
int i = static_cast<int>(x);
```

Częstym zastosowaniem rzutowania statycznego jest rzutowanie od typu **void*** do typu **Typ*** (konwersja w drugą stronę jest zawsze bezpieczną konwersją standardową, która nie wymaga sprawdzania, więc nie musi być jawna). Takie konwersje stosuje się także do rzutowaia w dół wskaźników typu „wskaźnik do obiektu klasy bazowej" do typu „wskaźnik do obiektu klasy pochodnej" (dla typów niepolimorficznych, o czym powiemy w dalszej części).

20.2.3 Konwersje dynamiczne

Konwersja dynamiczna ma postać

```
dynamic_cast<Typ>(wyrazenie)
```

Konwersje dynamiczne stosuje się, gdy prawidłowość przekształcenia nie może być sprawdzona na etapie kompilacji, bo zależy od typu obiektu klasy polimorficznej. Typ ten jest znany dopiero w czasie wykonania i wtedy ma miejsce sprawdzenie poprawności. Tego rodzaju konwersje używane są wyłącznie w odniesieniu do klas polimorficznych i tylko dla typów wskaźnikowych i referencyjnych. Ponieważ o polimorfizmie jeszcze nie mówiliśmy, pozostawimy dalsze szczegóły do rozdz. 25.2 (str. 557).

20.2.4 Konwersje wymuszane

„Najsilniejszą" formą konwersji jest **konwersja wymuszana**. Ma ona postać

```
reinterpret_cast<Typ>(wyrazenie)
```

Użycie takiej konwersji oznacza, że rezygnujemy ze sprawdzania jej poprawności w czasie kompilacji i wykonania i, co za tym idzie, ponosimy pełną odpowiedzialność za jej skutki. Stosuje się ją, gdy wiemy z góry, że ani w czasie kompilacji, ani w czasie wykonania nie będzie możliwe określenie jej sensowności. W ten sposób można, na przykład, dokonać konwersji **char*** → **int*** lub **klasaA*** → **klasaB***, gdzie klasy **klasaA** i **klasaB** są zupełnie niezależne. Takie konwersje nie są bezpieczne, a ich rezultat może zależeć od używanej platformy czy kompilatora.

Przyjrzyjmy się na przykład poniższemu programowi:

P176: ***dyncast.cpp*** Jawne konwersje wymuszane w C++

```
#include <iostream>
#include <cstring>
#include <fstream>
using namespace std;

class Person {
    char nam[30];
    int  age;
public:
    Person(const char* n, int a) : age(a) {
        strcpy(nam,n);
    }

    void info() {
        cout << nam << " (" << age << ")" << endl;
    }
};

int main() {
    const size_t size = sizeof(Person);

    Person john("John Brown",40);
    Person mary("Mary Wiles",26);

    ofstream out("person.ob");
    out.write(reinterpret_cast<char*>(&john),size);
    out.write(                (char*) &mary ,size);
    out.close();

    char* buff1 = new char[size];
    char* buff2 = new char[size];
    ifstream in("person.ob");
    in.read(buff1,size);
    in.read(buff2,size);
```

```
35      in.close();
36
37      Person* p1 = reinterpret_cast<Person*>(buff1);
38      Person* p2 =                  (Person*) buff2 ;
39
40      p1->info();
41      p2->info();
42
43      delete [] buff1;
44      delete [] buff2;
45  }
```

Zdefiniowaliśmy tu (linie 22 i 23) dwa obiekty klasy **Person**, zawierającej jedną składową tablicową i jedną typu **int**. W liniach 26 i 27 zapisujemy te obiekty w formie binarnej do pliku. Metoda **write** (patrz rozdz. 16.4.2 na stronie 341) ma pierwszy parametr typu **const char***. Tymczasem chcemy zapisać reprezentację binarną obiektu john klasy **Person**: obiekt ten znajduje się pod adresem &john i ma długość **sizeof**(Person). Typem &john jest **Person***, zatem dokonujemy konwersji tego argumentu do typu **char*** za pomocą rzutowania z użyciem **reinterpret_cast**. W linii 27 robimy to samo za pomocą rzutowania w stylu C, aby pokazać, te dwie formy mogą tu być użyte zamiennie.

Dwa obiekty klasy **Person** zapisane na dysk odczytujemy następnie do dwóch tablic znakowych buff1 i buff2 (linie 33 i 34). Po wczytaniu są to po prostu tablice znaków (bajtów): ani w czasie kompilacji, ani w czasie wykonania system nie ma możliwości sprawdzenia, czy zawarte w nich ciągi bajtów rzeczywiście są reprezentacją obiektów klasy **Person**. Ale my wiemy, że powinno tak być, bo sami przed chwilą te ciągi bajtów zapisaliśmy. Wymuszamy zatem (linie 37 i 38) konwersję zmiennych buff do typu **Person*** — znów na dwa sposoby: raz za pomocą **reinterpret_cast** i raz za pomocą rzutowania w stylu C. Wydruk z linii 40 i 41

```
John Brown (40)
Mary Wiles (26)
```

przekonuje nas, że konwersja się udała. Pamiętać jednak trzeba, że karkołomne konwersje wymuszane nie zawsze dają rezultaty zgodne z oczekiwaniem. Co gorsza, rezultaty te mogą zależeć od użytego kompilatora i architektury komputera: skoro świadomie zrezygnowaliśmy z kontroli typów, język nie daje nam tu żadnych gwarancji. Tworzona jest wartość, która ma wzorzec bitowy taki jak wartość konwertowana, ale przypisany jest jej inny typ: sensowność tego nie jest ani zapewniana, ani sprawdzana.

— 21 —

Dziedziczenie i polimorfizm

Podobnie jak w Javie i innych językach obiektowych, klasy mogą dziedziczyć własności po innych klasach — nazywamy je wtedy klasami **pochodnymi** (ang. *derived class, subclass*), a klasy, z których dziedziczą, ich klasami **bazowymi** (ang. *base class, superclass*). Klasy, które nie dziedziczą z żadnej innej klasy, nazywamy **pierwotnymi**. W odróżnieniu od Javy i C#, klasy w C++ mogą dziedziczyć z wielu klas bazowych — pierwotnych lub pochodnych: jest to **wielodziedziczenie** , zwane też dziedziczeniem **wielobazowym** (którego lepiej unikać, prowadzi bowiem do kodu zawiłego i trudnego do modyfikacji). W tym rozdziale zapoznamy się z podstawowymi własnościami dziedziczenia w języku C++.

Podrozdziały:

21.1 Podstawy dziedziczenia

Definiując klasę pochodną definiujemy typ danych rozszerzający typ określany przez klasę bazową, a więc tę, z której klasa dziedziczy. Obiekty klasy pochodnej będą zawierać te składowe, które zawierają obiekty klasy bazowej, i, choć niekoniecznie, dodatkowe składowe, których nie było w klasie bazowej. Klasa pochodna może też dodawać nowe metody lub zmieniać implementację metod odziedziczonych ze swojej klasy bazowej.

Zatem klasa bazowa jest bardziej ogólna, modeluje pewien fragment rzeczywistości na wyższym poziomie abstrakcji. Klasa pochodna jest bardziej szczegółowa, mniej abstrakcyjna. Tak więc, na przykład, pojęcie *mebel* jest bardziej abstrakcyjne, zaś krzesło bardziej konkretne: zatem klasa opisująca krzesła dziedziczyłaby z klasy opisującej meble: **Mebel** ← **Krzeslo**. Mogłaby dodać, na przykład, składową opisującą ilość nóg, której w bardziej ogólnej klasie **Mebel** nie było, bo nie każdy mebel ma nogi.

Zauważmy tu, że zaznaczając dziedziczenie za pomocą strzałek, jak to zrobiliśmy powyżej, rysujemy te strzałki w kierunku od klasy pochodnej do klasy bazowej.

Składnia deklaracji/definicji klas pochodnych jest następująca:

```
class A {
    // ...
};

class B : public A {
    // ...
};

class C : B {
    // ...
};
```

Klasy bazowe dla danej klasy deklarujemy na **liście dziedziczenia** umieszczonej po nazwie klasy i dwukropku, a przed definicją (ciałem) klasy. Klas bazowych może być kilka: ich nazwy umieszczamy wtedy na liście oddzielając je przecinkami. Powyższy zapis oznacza, że

- klasa **A** jest klasą pierwotną; nie dziedziczy z żadnej innej klasy (nie ma w C++ wspólnego „przodka" w rodzaju klasy **Object** z Javy);
- klasa **B** dziedziczy z **A**;
- klasa **C** dziedziczy z **B**, a więc pośrednio również z klasy **A**.

W definicji klasy **B** występuje specyfikator dostępu **public**. Prócz tego specyfikatora, mogą w tym miejscu wystąpić również specyfikatory **private** lub **protected**. Określają one górną granicę dostępności tych składowych klasy pochodnej, które odziedziczone zostały z klasy bazowej (patrz rozdz. 14.2 na stronie 264). Oznacza to, że składowe określone w klasie bazowej będą w klasie pochodnej miały dostępność taką samą jak w klasie bazowej lub węższą, jeśli ich dostępność w klasie bazowej była szersza niż ta zadeklarowana w klasie pochodnej. Odziedziczone składowe prywatne w ogóle nie są w klasie pochodnej bezpośrednio dostępne. Mogą jednak być dostępne za pomocą odziedziczonych nieprywatnych metod z klasy bazowej — takie metody znajdują się w zakresie klasy bazowej, więc do jej składowych prywatnych oczywiście mają dostęp.

Składowe prywatne odziedziczone z klasy bazowej, nawet jeśli nie są widoczne w klasie pochodnej, fizycznie wchodzą w skład obiektów klasy pochodnej.

Reasumując:

- Specyfikator **public** oznacza, że wszystkie składowe publiczne w klasie bazowej będą publiczne i w klasie pochodnej, a składowe chronione (**protected**) będą chronione i w klasie dziedziczącej (składowe prywatne, jak powiedzieliśmy, nie są widoczne);
- Specyfikator **protected** oznacza, że wszystkie składowe publiczne w klasie bazowej będą *chronione* (**protected**) w klasie pochodnej, tak jak i składowe chronione z klasy bazowej.

- Specyfikator **private** oznacza, że wszystkie składowe publiczne i chronione z klasy bazowej stają się prywatnymi w klasie pochodnej.

Przypomnijmy tu, że

> Atrybut **protected** oznacza, że dane pole czy metoda będą dostępne (tak jakby były publiczne) w klasach pochodnych danej klasy, a zachowują się jak prywatne dla wszystkich innych klas i funkcji.

Brak specyfikatora na liście dziedziczenia, jak w definicji klasy **C** z powyższego przykładu, jest równoważny z określeniem specyfikatora **private**.

Zauważmy, że w powyższym przykładzie dziedziczenie z klasy **C** miałoby już niewielki sens, bo w obiektach ewentualnej klasy dziedziczącej żadne składowe z klas bazowych (w tym przypadku **A**, **B** i **C**) nie byłyby w ogóle widoczne.

Jeśli w klasie pochodnej zawęziliśmy dostępność pól/metod specyfikatorem **protected** lub **private** w definicji klasy (po dwukropku, a przed ciałem klasy), to można tę dostępność *przywrócić* indywidualnie dla wybranych pól/metod, podając ich identyfikatory w postaci kwalifikowanych nazw w odpowiednich sekcjach. Podkreślmy jeszcze raz: *nazwy*, a nie pełne deklaracje. Kwalifikowane, czyli wraz z nazwą klasy bazowej.

W poniższym przykładzie w klasie **A** składowe x, y i z są publiczne, a składowe (tutaj będące metodami) **fff**, **ggg** i **hhh** są chronione. Składowa k jest prywatna i nie będzie bezpośrednnio widoczna w klasie pochodnej.

```
class A {
    double k;

public:
    int x, y, z

protected:
    double fff(int);
    double ggg(int);
    double hhh(int);
    // ...
};
```

Niech teraz klasa **B** będzie zdefiniowana tak:

```
1   class B : private A {
2   public:
3       A::x;
4       A::y;
5
6   protected:
7       A::fff;
8       // ...
9   };
```

W klasie **B** dostępność dziedziczonych składowych klasy **A** zawężona jest do poziomu **private** (linia 1; ten sam efekt można było zapewnić nie podając specyfikatora dostępności w ogóle, gdyż dostępność **private** jest domyślna). Następnie jednak przywracana jest dostępność publiczna dla składowych `x` i `y` (linie 2-4). Zauważmy, że *nie* przywracamy takiej dostępności składowej `z`, tak więc w klasie **B** stanie się ona prywatna. Zauważmy też, że w liniach 3 i 4 wymienieliśmy tylko kwalifikowane nazwy, a nie deklaracje — nie podaliśmy na przykład typu pól.

W liniach 6-7 przywróciliśmy dostępność chronioną dla składowej **fff**. Tu również podaliśmy tylko kwalifikowaną nazwę, a nie deklarację funkcji — nie ma tu listy parametrów czy określenia typu wartości zwracanej. Metody **ggg** i **hhh**, którym dostępności chronionej nie przywróciliśmy, stają się w klasie **B** prywatne.

Prócz odziedziczonych składowych w klasie pochodnej można definiować własne pola i metody. Tak więc obiekt klasy pochodnej nigdy nie będzie mniejszy niż obiekt klasy bazowej: jak wspomnieliśmy, zawiera bowiem zawsze kompletny podobiekt klasy bazowej i często dodatkowe składowe, których w klasie bazowej nie było.

W klasie pochodnej można definiować składowe o tych samych nazwach co składowe klasy bazowej. Mówimy wtedy o przesłanianiu pól i metod (przedefiniowywaniu, nadpisywaniu, przekrywaniu, przykrywaniu ...; ang. *overriding*). Przesłaniania pól lepiej nie stosować, bo prowadzi to do chaosu trudnego do opanowania. Natomiast przesłanianie metod jest fundamentalnym narzędziem programowania obiektowego.

Metody/pola z klasy bazowej (na przykład klasy **A**) mają zakres tejże klasy bazowej; na przykład mają dostęp do składowych prywatnych tej klasy. Natomiast zakres klasy pochodnej jest zawarty w zakresie klasy podstawowej. Znaczy to, że jeśli te pola i metody *nie* są prywatne i *nie* zostały w klasie pochodnej przedefiniowane (przesłonięte), to w klasie pochodnej (na przykład **B**) są również widoczne bezpośrednio (a więc nie trzeba stosować dla nich nazw kwalifikowanych). Jeśli natomiast w klasie pochodnej zostały przesłonięte, to zakresem takich pól/metod będzie klasa pochodna **B**: w tej klasie dostępne są bezpośrednio „wersje" przedefiniowane. Oczywiście składowe prywatne z klasy bazowej nie są dostępne w zakresie klasy **B**. Natomiast do składowych nieprywatnych z klasy **A**, przesłoniętych w klasie pochodnej **B**, można się wciąż odwołać z zakresu klasy **B** poprzez jawną kwalifikację nazwy za pomocą operatora zakresu klasowego, a więc, w naszym przypadku, poprzedzając nazwę wskazaniem zakresu '`A::`'.

Rozpatrzmy przykład:

P177: ***inher.cpp*** Widoczność przesłoniętych składowych

```
1  #include <iostream>
2  using namespace std;
3
4  class A {
5  public:
6      int fun(int x) { return x*x; }
7  };
8
9  class B : private A {
10     int fun(int x, int y) {
```

```
11          return A::fun(x) + y*y;
12      }
13  public:
14      int pub(int x, int y) { return fun(x,y); }
15  };
16
17  int main() {
18      A a;
19      B b;
20      cout << "a.fun(3)    = " << a.fun(3)    << endl;
21      cout << "b.pub(3,4) = " << b.pub(3,4) << endl;
22   // cout << "b.fun(3,4) = " << b.fun(3,4) << endl;
23  }
```

W klasie **A** funkcja **fun** jest publiczna. Klasa **B** dziedziczy z klasy **A**, ale ogranicza dostępność składowych odziedziczonych do **private**. W klasie **B** funkcja **fun** jest przesłaniana; ponadto klasa ta definiuje publiczną funkcję **pub**. Wewnątrz funkcji **pub** użyta jest nazwa **fun**. Czego ona dotyczy? Ponieważ funkcja **pub** jest składową klasy **B**, więc w jej zakresie nazwa **fun** odnosi się do wersji zdefiniowanej w tejże klasie, czyli do funkcji zdefiniowanej w liniach 10-12. W definicji tej funkcji wywołana ma być funkcja **fun**, ale w wersji z klasy bazowej. Ponieważ funkcja **fun** w klasie bazowej jest publiczna, można ją wywołać z zakresu klasy pochodnej pod warunkiem, że użyjemy nazwy kwalifikowanej (linia 11). Gdybyśmy nie kwalifikowali nazwy **fun**, to użyta zostałaby wersja z klasy **B**, co doprowadziłoby w naszym przypadku do nieskończonej rekursji; dzięki kwalifikacji natomiast wywołana zostanie właściwa funkcja:

```
a.fun(3)    = 9
b.pub(3,4) = 25
```

Zauważmy, że na rzecz obiektu klasy **A** możemy wywołać funkcję **fun** (linia 20), bo w tej klasie funkcja ta, zdefiniowana w linii 6, jest publiczna. Natomiast na rzecz obiektu klasy **B** (wykomentowana linia 22) tego zrobić nie możemy, bo w klasie **B** funkcja **fun** jest prywatna. Możemy natomiast wywołać publiczną funkcję **pub**, a ta, jako metoda klasy **B**, ma w swoim zakresie prywatną wersję funkcji **fun** zdefiniowaną w tej klasie.

Jak już mówiliśmy, obiekt klasy pochodnej jest obiektem klasy bazowej, uzupełnionym ewentualnie przez dodatkowe składowe. W tym sensie może być w wielu sytuacjach traktowany jak gdyby był obiektem klasy bazowej (tak jak w Javie i innych językach obiektowych).

Na przykład wskaźnik lub referencja do obiektu klasy pochodnej może być użyty tam, gdzie oczekiwany jest wskaźnik (referencja) do obiektu klasy bazowej — wskazywanym obiektem będzie wówczas podobiekt klasy bazowej zawarty w obiekcie klasy pochodnej. Taka konwersja, zwana rzutowaniem **w górę** (ang. *upcasting*) jest standardowa i może być wykonywana niejawnie.

Niejawne rzutowanie w górę zachodzi tylko jeśli klasa pochodna dziedziczy z klasy bazowej publicznie (ze specyfikatorem **public**). W innych przypadkach takiej konwersji trzeba zażądać jawnie za pomocą operatora **static_cast** lub **dynamic_cast**.

Mechanizm, o którym wspomnieliśmy, ma dla programowania obiektowego fundamentalne znaczenie i w dalszym ciągu znajdziemy wiele przykładów na jego zastosowanie. Dzięki niemu:

- możemy przekazywać referencję do obiektu klasy pochodnej do konstruktora klasy bazowej, na przykład, aby utworzyć podobiekt klasy bazowej w czasie konstruowania obiektu klasy pochodnej;
- jeśli funkcja zwraca wskaźnik (referencję) do obiektu klasy bazowej, to w instrukcji **return** można zwrócić wskaźnik (referencję) do obiektu klasy pochodnej;
- dla operatorów, w których parametrem jest referencja do obiektu klasy bazowej, nastąpi ich wywołanie, gdy jako argument (obiekt po prawej stronie operatora) pojawi się obiekt klasy pochodnej. Dla operatorów zdefiniowanych jako metody nie dotyczy to jednak argumentu domyślnego **this** (wskaźnika do obiektu, na rzecz którego następuje wywołanie, czyli tego po lewej stronie operatora). Znaczy to, że jeśli w klasie **A** przeciążyliśmy operator dodawania, a `b` jest obiektem klasy **B** dziedziczącej z **A**, to opracowanie wyrażenia '`b+a`' *nie* spowoduje wywołania metody przeciążającej dodawanie z klasy **A**, bo wywołanie jest na rzecz obiektu klasy pochodnej. Natomiast '`a+b`' zadziała, bo `a` jest typu **A**, a dla argumentu `b` niejawna konwersja **B&** → **A&** zajdzie.

Konwersja w drugą stronę, od wskaźnika/referencji do obiektu klasy bazowej do wskaźnika/referencji do obiektu klasy pochodnej musi jednak być zawsze jawna; na przykład (dla klas polimorficznych, o czym za chwilę) za pomocą operatora konwersji dynamicznej (**dynamic_cast**), jak o tym wspominaliśmy w rozdziale o konwersjach (rozdz. 20.2.3, str. 448). Taki typ konwersji nazywamy rzutowaniem w **dół** (ang. *downcasting*). Nastąpi wtedy dynamicznie, czyli w trakcie wykonania programu, sprawdzenie poprawności takiego rzutowania.

Zauważmy, że mówimy tu o wskaźnikach i referencjach, a nie o konwersji samych obiektów. Na przykład, jeśli parametrem funkcji jest *obiekt* klasy bazowej **A** (przekazywany przez wartość, a nie przez wskaźnik lub referencję), to jako argumentu *nie powinniśmy* używać obiektu klasy pochodnej **B**. Wynika to choćby z faktu, że obiekty klasy pochodnej i bazowej zajmują na stosie różną liczbę bajtów, a więc aby takie wywołanie zrealizować obiekt musiałby zostać „przycięty” (do zawartego w nim podobiektu typu **A**), co czasem daje oczekiwane rezultaty, a czasem zupełnie nieoczekiwane...

Często zachodzi potrzeba definiowania wskaźników lub referencji do podobiektu klasy bazowej zawartego w obiekcie klasy pochodnej lub odwrotnie. Niech **A** będzie klasą bazową, a **B** klasą pochodną. Wtedy:

- jeśli pb jest wskaźnikiem typu **B*** na obiekt b klasy **B**, to `(A*)pb` lub `static_cast<A*>(pb)` jest wskaźnikiem do podobiektu klasy **A** zawartego w obiekcie b. Zamiast pb można oczywiście użyć równoważnie &b;
- podobnie, `(A&)b` lub `static_cast<A&>(b)` jest referencją do tego podobiektu;
- jeśli pa jest wskaźnikiem typu **A*** do obiektu klasy **B**, to `(B*)pa` lub `dynamic_cast<B*>(pa)` jest wskaźnikiem do obiektu typu **B**. Aby użyć rzutowania dynamicznego, klasa **A** powinna być polimorficzna, a więc zawierać przynajmniej jedną metodę wirtualną (wystarczy, że jest to na przykład destruktor — patrz dalsza część rozdziału);
- podobnie `(B&)(*pa)` lub `dynamic_cast<B&>(*pa)` jest referencją do tego obiektu.

Operowanie tymi konwersjami jest przydatne i konieczne zarówno przy korzystaniu z polimorfizmu, jak i na przykład w konstruktorach i przypisaniach definiowanych w klasach dziedziczących z innych klas.

Rozpatrzmy przykład ilustrujący różne tego rodzaju rzutowania.

P178: *cast.cpp* Konwersje wiążące obiekt i podobiekt

```
#include <iostream>
using namespace std;

struct A {
    int x;
    int y;
    A() : x(1), y(2) {}
};

struct B : public A {
    int x;  // ????
};

int main() {
    B b, *pb = &b;
    b.x = 11;
    b.y = 12;

    cout << "b.x=" << b.x         << " b.y="
         << b.y    << " b.A::x=" << b.A::x
         << " b.A::y=" << b.A::y << endl;

    cout << "\n      pb->x=" <<    pb ->x   << endl;
    cout <<   "((A*)pb)->x=" << ((A*)pb)->x << endl;
    cout <<   "         b.x=" <<       b.x   << endl;
    cout <<   "  ((A&)b).x=" << ((A&)b).x   << endl;
```

```
27      cout <<    "((A*)&b)->x=" << ((A*)&b)->x << endl;
28
29      A* pa = new B;
30      ((B&)*pa).x = 11;
31
32      cout << "\n     (*pa).x=" << (*pa).x      << endl;
33      cout <<    "((B&)*pa).x=" << ((B&)*pa).x << endl;
34      cout <<    "       pa->x=" <<        pa->x << endl;
35      cout <<    "((B*)pa)->x=" << ((B*)pa)->x << endl;
36
37      cout << "\nsizeof(b) = " << sizeof b << endl;
38      int* t = (int*) &b;
39      cout << t[0] << " " << t[1] << " " << t[2] << endl;
40  }
```

Klasa **A** ma składowe x i y. Dziedzicząca klasa **B** definiuje pole o nazwie x (linia 11), więc zasłania składową x dziedziczoną z **A** (co, jak wspomnieliśmy, nie jest zalecane!). W programie głównym tworzymy obiekt klasy **B**. W liniach 16-17 inicjujemy składowe tego obiektu. Zauważmy, że prócz x i y w skład obiektu wchodzi też przesłonięta składowa x odziedziczona z **A**. Tak więc w zakresie klasy **B** nazwa x oznacza składową (o wartości 11) zdefiniowaną w tej klasie, a nazwa kwalifikowana A::x oznacza składową x (o wartości 1) odziedziczoną z klasy **A**. Tak więc b.x wynosi 11, ale b.A::x wynosi 1. Widać to z wydruku

```
b: x=11 y=12 b.A::x=1 b.A::y=12

      pb->x=11
((A*)pb)->x=1
        b.x=11
  ((A&)b).x=1
((A*)&b)->x=1

    (*pa).x=1
((B&)*pa).x=11
      pa->x=1
((B*)pa)->x=11

sizeof(b) = 12
1 12 11
```

Z wydruku widzimy też, że jeśli pb zrzutujemy do typu **A***, to wartością `((A*)pb)->x` jest 1, czyli wyłuskiwana jest wartość x z podobiektu klasy **A**. Podobnie `(A&)b` jest referencją do tego podobiektu.

Ponieważ składowa y *nie* została przesłonięta, więc w zakresie klasy **B** nazwy y i A::y ozaczają tę samą zmienną.

W linii 29 tworzymy obiekt klasy **B**, ale wskaźnik do niego zapisujemy w zmiennej pa typu **A***. Na wydruku widzimy teraz efekt rzutowania w dół: pa jest wskaźnikiem

typu **A*** wskazującym obiekt klasy **B**. Jej typem jest **A***, a zatem `pa->x` odnosi się do składowej `x` w podobiekcie klasy **A** zawartym w obiekcie `b`. Po zrzutowaniu do typu wskaźnikowego **B***, wskazywaną składową o nazwie `x` jest jednak składowa o tej nazwie z zakresu **B** (linia 35).

W liniach 37-39 sprawdzamy jaki jest rozmiar obiektu typu **B**. Wynosi on 12 bajtów, co odpowiada trzem liczbom typu **int** — są to `x` i `y` z podobiektu klasy **A** oraz `x` dodane w klasie **B** (UWAGA: ten fragment *może* być zależny od architektury komputera i użytego kompilatora!). Traktując adres obiektu jak adres tablicy liczb całkowitych (linia 38), "możemy się przekonać, że rzeczywiście obiekt zawiera liczby 1, 12 (`x` i `y` z podobiektu typu **A**) oraz 11 (składowa `x` dodana w klasie **B**).

Bardzo ważną właściwością dziedziczenia jest to, że

Nie są dziedziczone konstruktory i destruktor.

Z drugiej strony, właśnie konstruktory i destruktory pełnią ważną rolę, szczególnie dla klas zawierających pola wskaźnikowe, kiedy logiczna zawartość obiektów nie wchodzi fizycznie w ich skład. Zatem problemy związane z konstrukcją obiektów klas pochodnych i ich destrukcją omówimy teraz osobno.

21.2 Konstruktory i destruktory klas pochodnych

Jak wspomnieliśmy, konstruktory nie są dziedziczone. Jeśli w klasie pochodnej nie zdefiniowaliśmy konstruktora, to zostanie użyty konstruktor domyślny. Aby jednak powstał obiekt klasy pochodnej, musi być *najpierw* utworzony podobiekt klasy nadrzędnej wchodzący w jego skład. On również zostanie utworzony za pomocą konstruktora domyślnego, który zatem musi istnieć!

Podobiekt klasy bazowej jest tworzony jeszcze przed wykonaniem konstruktora klasy pochodnej.

Co w takim razie zrobić, aby do konstrukcji podobiektu klasy bazowej zawartego w tworzonym właśnie obiekcie klasy pochodnej użyć konstruktora innego niż domyślny? Wówczas musimy w klasie pochodnej zdefiniować konstruktor, a wywołanie właściwego konstruktora dla podobiektu klasy bazowej *musi* nastąpić poprzez listę inicjalizacyjną — wewnątrz konstruktora byłoby już za późno (patrz rozdz. 15.3.2 na stronie 304). Na liście tej umieszczamy jawne wywołanie konstruktora dla podobiektu. Tak więc, jeśli klasą bazową jest klasa **A** i chcemy wywołać jej konstruktor, aby „zagospodarował" podobiekt tej klasy dziedziczony w klasie **B**, to na liście inicjalizacyjnej konstruktora klasy pochodnej **B** umieszczamy wywołanie `A(...)`, gdzie w miejsce kropek wstawiamy oczywiście argumenty dla wywoływanego konstruktora. Wywołuje się tylko konstruktory *bezpośredniej* klasy bazowej („ojca", ale nie „dziadka"; oczywiście konstruktor ojca poprzez swoją listę inicjalizacyjną może wywołać konstruktor swojego ojca...).

Na przykład w poniższym fragmencie definiujemy klasę bazową **Point**. Nie ma ona w ogóle konstruktora domyślnego.

P179: *pix.cpp* Wywołanie konstruktora klasy bazowej

```
1  struct Point {
2      int x;
3      int y;
4      Point(int x, int y)
5          : x(x), y(y)
6      { }
7  };
8
9  struct Pixel: public Point {
10     int color;
11     Pixel(int x, int y, int color)
12         : Point(x,y), color(color)
13     { }
14 };
```

Klasa **Pixel** dziedziczy z klasy **Point**. Ponieważ klasa **Point** nie miała konstruktora domyślnego, w klasie **Pixel** *musimy* przynajmniej jeden konstruktor zdefiniować i na jego liście inicjalizacyjnej wywołać jawnie konstruktor klasy **Point** z odpowiednimi argumentami, co robimy w linii 12 powyższego fragmentu.

Zauważmy, że *nie* wolno na liście inicjalizacyjnej konstruktora klasy pochodnej wymieniać nazw pól z klasy bazowej. W powyższym przykładzie na liście inicjalizacyjnej konstruktora klasy **Pixel** nie można umieścić wyrażenia `x(x)`, bo składowa `x` pochodzi z klasy bazowej. Wolno natomiast, za pomocą wyrażenia `color(color)`, zainicjować składową color, bo jest ona zadeklarowana w klasie pochodnej **Pixel**, a nie było jej w klasie bazowej. Składowe dziedziczone z klasy bazowej mogą być więc zainicjowane, ale tylko za pomocą jawnego wywołania konstruktora klasy bazowej z listy inicjalizacyjnej konstruktora klasy pochodnej.

Pewien problem powstaje przy definiowaniu konstruktora kopiującego. Pisząc konstruktor kopiujący w klasie pochodnej, musimy zastanowić się, w jaki sposób skonstruowany będzie podobiekt klasy bazowej zawarty w tworzonym obiekcie klasy pochodnej. Jeśli z listy inicjalizacyjnej konstruktora kopiującego klasy pochodnej jawnie nie wywołamy konstruktora kopiującego klasy bazowej, to do utworzenia podobiektu klasy bazowej użyty będzie konstruktor *domyślny* klasy bazowej (który wobec tego musi istnieć).

Jeśli jednak chcemy, aby do utworzenia podobiektu użyty został inny niż domyślny konstruktor klasy bazowej, to musimy jawnie go wywołać z listy inicjalizacyjnej. Jakiego jednak argumentu użyć przy tym wywoływaniu? Odpowiedź jest prosta i wynika z tego, o czym mówiliśmy w poprzednim podrozdziale: ponieważ typem parametru konstruktora kopiującego klasy bazowej **A** jest **A&** (lub, częściej, **const A&**), więc wystarczy „wysłać” referencję do tego obiektu klasy pochodnej **B**, który jest argumentem wywołania konstruktora z klasy **B** (czyli jest obiektem-wzorcem). Tak jak bowiem mówiliśmy, jeśli typem parametru funkcji jest wskaźnik lub referencja do obiektu klasy

bazowej, to dopuszczalnym argumentem wywołania jest wartość wskaźnikowa lub referencyjna odnosząca się do obiektu klasy pochodnej.

W poniższym programie, dla uproszczenia, w ogóle nie ma pól wskaźnikowych, ale definiujemy konstruktory kopiujące (i domyślne):

P180: ***cop.cpp*** Konstruktory kopiujące

```
#include <iostream>
using namespace std;

class A {
    int a;
public:
    A(const A& aa) {
        a = aa.a;
        cout << "Copy-ctor A, a = " << a << endl;
    }

    A(int aa = 0) {
        a = aa;
        cout << "Def-ctor  A, a = " << a << endl;
    }

    void showA() { cout << "a = " << a; }
};

class B: public A {
    int b;
public:
    B(const B& bb)
        : A(bb)
    {
        b = bb.b;
        cout << "Copy-ctor B, b = " << b << endl;
    }

    B(int bb = 1)
        : A(1)
    {
        b = bb;
        cout << "Def-ctor  B, b = " << b << endl;
    }

    void showB() {
        showA();
        cout << ", b = " << b << endl;
    }
```

```
41 };
42
43 int main() {
44     B b1(2);
45     b1.showB();
46
47     B b2(b1);
48     b2.showB();
49 }
```

W liniach 23-28 definiujemy konstruktor kopiujący dla klasy pochodnej **B**. Na liście inicjalizacyjnej wywołujemy konstruktor z klasy **A**, a jako argumentu używamy obiektu klasy **B**. Ponieważ **B** jest klasą pochodną, więc takie wywołanie „pasuje" do konstruktora kopiującego klasy **A**. Wynikiem jest

```
Def-ctor  A, a = 1
Def-ctor  B, b = 2
a = 1, b = 2
Copy-ctor A, a = 1
Copy-ctor B, b = 2
a = 1, b = 2
```

Jawne wywołanie konstruktora kopiującego z linii 24 można usunąć. Wtedy, zgodnie z tym co mówiliśmy, do skonstruowania podobiektu klasy **A** zostanie użyty konstruktor domyślny tej klasy; po wykomentowaniu linii 24 wydruk programu będzie zatem

```
Def-ctor  A, a = 1
Def-ctor  B, b = 2
a = 1, b = 2
Def-ctor  A, a = 0
Copy-ctor B, b = 2
a = 0, b = 2
```

Patrząc na ten i poprzedni wydruk, widzimy, że

> Konstruktory wywoływane są w kolejności „od góry": najpierw dla podobiektów klas bazowych, potem dla danej klasy. Jeśli klasa dziedziczy z kilku klas, to konstruktory klas bazowych są wywoływane w kolejności ich wystąpienia na liście dziedziczenia.

Tak będzie również dla bardziej rozbudowanej hierarchii dziedziczenia. Jeśli, na przykład, klasa **C** dziedziczy z **B**, która z kolei dziedziczy z **A**, to podczas tworzenia obiektu klasy **C** najpierw zostanie utworzony obiekt **A**, następnie obiekt klasy **B** zawierający jako podobiekt utworzony już obiekt **A**, a dopiero na końcu obiekt klasy **C** zawierający jako podobiekt utworzony obiekt klasy **B**.

Z kolei, w trakcie konstrukcji obiektów poszczególnych klas najpierw tworzone są obiekty będące składowymi klasy (przez wywołanie ich konstruktorów, jeśli są to składowe obiektowe). Tworzone one są w kolejności ich zadeklarowania. Potem dopiero wykonywane jest ciało samego konstruktora klasy.

Jak wspomnieliśmy, destruktory również nie są dziedziczone. Jeśli są zdefiniowane, to

destruktory wywoływane są w kolejności odwrotnej do konstruktorów.

A zatem, podczas niszczenia obiektu najpierw wywoływany jest jego destruktor (oczywiście, jeśli jest zdefiniowany). Następnie usuwane są obiekty będące składowymi tego obiektu nie odziedziczonymi z klas bazowych. Jeśli są to składowe obiektowe, to oczywiście nastąpi wywołanie dla nich destruktorów, jeśli były zdefiniowane. Następnie wywoływany jest destruktor dla podobiektu bezpośredniej klasy bazowej, potem usuwane są obiekty będące niedziedziczonymi składowymi tego podobiektu i tak dalej. Ilustruje to poniższy program:

P181: ***condes.cpp*** Kolejność wywoływania konstruktorów i destruktorów

```
#include <iostream>
using namespace std;

struct K {
    char k;
    K(char kk = 'k') {
        k = kk;
        cout << "Ctor K\n";
    }

    ~K() {
        cout << "Dtor K\n";
    }
};

struct A {
    char a;
    A(char aa = 'a') {
        a = aa;
        cout << "Ctor A\n";
    }

    ~A() {
        cout << "Dtor A\n";
    }
};

```

```
28 struct B: public A {
29     char b;
30     K    k;
31     B(char bb = 'b') : A(bb) {
32         b = bb;
33         cout << "Ctor B\n";
34     }
35
36     ~B() {
37         cout << "Dtor B\n";
38     }
39 };
40
41 struct C: public B {
42     char c;
43     C(char cc = 'c') : B(cc) {
44         c = cc;
45         cout << "Ctor C\n";
46     }
47
48     ~C() {
49         cout << "Dtor C\n";
50     }
51 };
52
53 int main() {
54     C c;
55 }
```

Mamy tu hierarchię dziedziczenia **A** ← **B** ← **C** (jak zwykle strzałka wskazuje od klasy pochodnej do klasy bazowej). Klasa **B** ma też pole obiektowe typu **K**. W funkcji **main** tworzymy obiekt klasy **C**, który po wyjściu z funkcji jest niszczony. A oto wydruk z tego programu:

```
Ctor A
Ctor K
Ctor B
Ctor C
Dtor C
Dtor B
Dtor K
Dtor A
```

Zgodnie z tym co mówiliśmy, najpierw tworzony jest podobiekt klasy **A**, a co za tym idzie wywoływany jest konstruktor tej klasy. Następnie tworzony jest obiekt klasy **B**. Obiekt ten ma składową obiektową typu **K**. Widzimy, że najpierw konstruowana jest ta składowa, a co za tym idzie wywoływany jest konstruktor klasy **K**, a dopiero

potem wywoływany jest konstruktor klasy **B**. Dopiero na samym końcu wywoływany jest konstruktor klasy **C**.

Kolejność wywoływania destruktorów jest dokładnie odwrotna. W szczególności podczas niszczenia podobiektu klasy **B** najpierw wywoływany jest destruktor tej klasy, a dopiero potem niszczona jest jego składowa obiektowa klasy **K**.

21.3 Operator przypisania dla klas pochodnych

Dużo trudności sprawia czasem właściwe zdefiniowanie (przeciążenie) operatora przypisania dla klasy pochodnej.

Jeżeli w klasie bazowej jest zdefiniowany nieprywatny operator przypisania, a w klasie pochodnej nie jest, to do przypisania odziedziczonego podobiektu klasy bazowej zostanie użyty operator przypisania zdefiniowany w klasie bazowej. Natomiast dla części „własnej" obiektu klasy pochodnej zostanie wtedy użyte przypisanie dostarczone przez system (a więc „pole po polu"; prawdopodobnie nieprawidłowo, jeśli są w klasie pola wskaźnikowe). Rozpatrzmy na przykład poniższy program, w którym definiujemy prostą klasę **A** i klasę dziedziczącą **B**:

P182: *inhas.cpp* Dziedziczenie operatora przypisania

```
#include <iostream>
using namespace std;

struct A {
    char a;
    A(char aa = 'a') {
        a = aa;
    }

    A& operator=(const A& aa) {
        a = aa.a;
        cout << "A::operator=()\n";
        return *this;
    }
};

struct B: public A {
    char b;
    B(char bb = 'b') : A(bb) {
        b = bb;
    }
};

int main() {
    B b1(1),b2(2);
    b1 = b2;
```

```
27 }
```

Program ten drukuje

```
A::operator=()
```

co świadczy o tym, że podczas wykonywania przypisania dla obiektów klasy pochodnej **B** została automatycznie wywołana metoda **operator=()** z klasy nadrzędnej **A**. Ta odziedziczona z klasy bazowej metoda **operator=()** nie może oczywiście wiedzieć o składowych klasy pochodnej, których nie było w klasie bazowej.

Przeanalizujmy zatem sytuację, gdy operator przypisania został w klasie bazowej zdefiniowany, ale dla części „własnej" również chcemy przedefiniować przypisanie. Zatem w klasie pochodnej musimy również przeciążyć operator '='. Metoda przeciążająca operator przypisania w klasie pochodnej powinna uwzględniać składowe klasy pochodnej nieobecne w klasie bazowej oraz musi *jawnie* wywołać ten operator z klasy bazowej, by „zajął się" podobiektem z niej odziedziczonym: ponieważ zostanie wtedy uruchomiona wersja przypisania zdefiniowana w klasie pochodnej, więc dla części odziedziczonej metoda **operator=()** z klasy bazowej nie zostanie teraz wywołana „sama z siebie".

Zauważmy, że można to zrobić przez jawne, „po nazwie", wywołanie metody przeciążającej operator przypisania z klasy bazowej. Można też osiągnąć ten sam cel przez proste przypisanie do *this, jeśli tylko podpowiemy kompilatorowi, żeby traktował wskazywany obiekt jako obiekt klasy bazowej (obiektem tym jest podobiekt klasy bazowej zawarty w obiekcie klasy pochodnej). Wtedy bowiem, zgodnie z regułami przeciążania operatorów, na rzecz tego podobiektu wywołana zostanie funkcja **operator=()** zdefiniowana w klasie bazowej.

Brzmi to zawile, ale jest całkiem proste. Załóżmy, jak zwykle, że klasa **B** dziedziczy z klasy **A**. Załóżmy też, że w klasie **A** operator przypisania został przeciążony, czyli jest w niej zdefiniowana metoda **operator=()**. Parametrem tej metody jest referencja (zwykle z modyfikatorem **const**) do obiektu klasy **A**, ale, jak wiemy, można taką metodę wywołać z argumentem typu pochodnego, którym to argumentem będzie referencja do przypisywanego obiektu klasy pochodnej (czyli tego, który występuje po prawej stronie przypisania). Na marginesie zauważmy, że cały mechanizm nie zadziałałby, gdybyśmy w klasie **A**, najzupełniej legalnie, zdefiniowali metodę **operator=()** z parametrem typu **A** lub **const A** zamiast **A&** lub **const A&**. Tak więc, przedefiniowując operator przypisania w klasie pochodnej **B**, moglibyśmy napisać (zakładając, że definicja ta jest poza klasą):

```
B& B::operator=(const B& b) {
    this->A::operator=(b);
    // ...
    return *this;
}
```

W ten sposób, na rzecz *this, a więc obiektu klasy pochodnej **B**, który pojawił się po lewej stronie przypisania, wywołujemy jawnie metodę **operator=()** z klasy

bazowej kwalifikując nazwę za pomocą operatora zakresu. Posyłamy jako argument przez referencję obiekt b, w więc ten, który pojawił się po prawej stronie przypisania. Ten sam efekt można uzyskać też tak:

```
B& B::operator=(const B& b) {
    (A&)(*this) = b;
    // ...
    return *this;
}
```

W tej metodzie referencja do obiektu *this klasy **B** została zrzutowana w górę do typu **A&**: tak więc przypisanie z trzeciej linii tego przykładu spowoduje automatycznie wywołanie metody **operator=()** z klasy bazowej **A**, podobnie jak w przypadku poprzednim. Oczywiście, zamiast rzutowania w stylu C, czyli za pomocą `(A&)` możemy użyć operatora rzutowania w stylu C++, czyli trzecią linię powyższego przykładu zastąpić przez

```
static_cast<A&>(*this) = b;
```

Wreszcie, równoważnie, można użyć rzutowania wskaźników i następnie wyłuskania (dereferencji) obiektu:

```
B& B::operator=(const B& b) {
    *((A*)this) = b;
    // ...
    return *this;
}
```

choć wygląda to chyba najmniej czytelnie.

W poniższym programie zademonstrowane są przeciążenia operatora przypisania, konstruktory, w tym kopiujące, i destruktory dla dwóch klas: **Osoba** i dziedziczącej z niej klasy **Pracownik**. W obu klasach istnieją pola wskaźnikowe, a więc prawidłowe zdefiniowanie konstruktorów kopiujących, destruktorów i operatorów przypisania jest niezbędne.

P183: *inh.cpp* Klasy z polami wskaźnikowymi: dziedziczenie

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  class Osoba {
6      char* nazwis;
7  public:
8      Osoba()
9          : nazwis(strcpy(new char[14], "Nazw.Nieznane"))
10     {
11         cout << "Konstr. domyslny Osoba: "
```

```
12              << nazwis << endl;
13      }
14
15      Osoba(const char* n)
16          : nazwis(strcpy(new char[strlen(n)+1], n))
17      {
18          cout << "Konstr. char* Osoba: " << nazwis << endl;
19      }
20
21      Osoba(const Osoba& os)
22          : nazwis(strcpy(new char[strlen(os.nazwis)+1],
23                                                  os.nazwis))
24      {
25          cout << "Konstr. kopiujacy Osoba: "
26               << nazwis << endl;
27      }
28
29      Osoba& operator=(const Osoba& os) {
30          if ( this != &os ) {
31              delete [] nazwis;
32              nazwis = strcpy(new char[strlen(os.nazwis)+1],
33                                                      os.nazwis);
34              cout << "Przypisanie Osoba: "
35                   << nazwis << endl;
36          }
37          return *this;
38      }
39
40      ~Osoba() {
41          cout << "Usuwamy Osoba: " << nazwis << endl;
42          delete [] nazwis;
43      }
44
45      const char* getNazwisko() const { return nazwis; }
46  };
47
48  class Pracownik : public Osoba {
49      char* funkcja;
50  public:
51      Pracownik()
52          : funkcja(strcpy(new char[14], "Stan.Nieznane"))
53      {
54          cout << "Konstruktor domyslny Pracownik: "
55               << funkcja << endl;
56      }
57
```

```
58      Pracownik(const char* s, const char* n)
59          : Osoba(n),funkcja(strcpy(new char[strlen(s)+1], s))
60      {
61          cout << "Konstruktor char* char* Pracownik: "
62               << funkcja << endl;
63      }
64
65      Pracownik(const Pracownik& prac)
66          : Osoba(prac), funkcja(strcpy(new
67                char[strlen(prac.funkcja)+1],prac.funkcja))
68      {
69          cout << "Konstruktor kopiujacy Pracownik: "
70               << funkcja << endl;
71      }
72
73      Pracownik& operator=(const Pracownik& prac) {
74          if ( this != &prac ) {
75              (Osoba&)(*this) = prac;
76              delete [] funkcja;
77              funkcja = strcpy(new
78                  char[strlen(prac.funkcja)+1],
79                  prac.funkcja);
80              cout << "Przypisanie Pracownik: "
81                   << funkcja << endl;
82          }
83          return *this;
84      }
85
86      ~Pracownik() {
87          cout << "Usuwamy Pracownik: " << funkcja << endl;
88          delete [] funkcja;
89      }
90
91      const char* getFunkcja() const { return funkcja; }
92  };
93
94  int main() {
95      cout << "\nMain: Tworzymy obiekt nem" << endl;
96      Pracownik nem;
97      cout << "Main: obiekt nemo utworzony: "
98           << nem.getFunkcja() << " "
99           << nem.getNazwisko()   << endl;
100
101     cout << "\nMain: Tworzymy obiekt mal" << endl;
102     Pracownik mal("Szef", "Malinowski");
103     cout << "Main: obiekt mal utworzony: "
```

```
104          << mal.getFunkcja() << " "
105          << mal.getNazwisko()   << endl;
106
107     cout << "\nMain: Kopiujemy mal -> kop" << endl;
108     Pracownik kop(mal);
109     cout << "Main: obiekt kop utworzony: "
110          << kop.getFunkcja() << " "
111          << kop.getNazwisko()   << endl;
112
113     cout << "\nMain: Przypisujemy nem = kop" << endl;
114     nem = kop;
115     cout << "Main: nem = kop przypisane: "
116          << nem.getFunkcja()  << " "
117          << nem.getNazwisko() << endl << endl;
118 }
```

Zauważmy w liniach 59 i 66 jawne wywołania konstruktora klasy bazowej **Osoba** z listy inicjalizacyjnej konstruktorów klasy pochodnej **Pracownik**. W linii 75 z kolei następuje, w ciele metody przeciążającej operator przypisania, jawne wywołanie analogicznej metody z klasy bazowej. Wydruk tego programu

```
Main: Tworzymy obiekt nem
Konstr. domyslny Osoba: Nazw.Nieznane
Konstruktor domyslny Pracownik: Stan.Nieznane
Main: obiekt nemo utworzony: Stan.Nieznane Nazw.Nieznane

Main: Tworzymy obiekt mal
Konstr. char* Osoba: Malinowski
Konstruktor char* char* Pracownik: Szef
Main: obiekt mal utworzony: Szef Malinowski

Main: Kopiujemy mal -> kop
Konstr. kopiujacy Osoba: Malinowski
Konstruktor kopiujacy Pracownik: Szef
Main: obiekt kop utworzony: Szef Malinowski

Main: Przypisujemy nem = kop
Przypisanie Osoba: Malinowski
Przypisanie Pracownik: Szef
Main: nem = kop przypisane: Szef Malinowski

Usuwamy Pracownik: Szef
Usuwamy Osoba: Malinowski
Usuwamy Pracownik: Szef
Usuwamy Osoba: Malinowski
Usuwamy Pracownik: Szef
Usuwamy Osoba: Malinowski
```

demonstruje kolejność wywoływania konstruktorów i metod przeciążających operator przypisania. Po zakończeniu programu wszystkie trzy obiekty są usuwane: co prawda wszystkie przechowują takie same dane, ale są to trzy niezależne obiekty, o czym świadczy fakt, że wszystkie wywołania destruktorów powiodły się. Widzimy, że przy konstrukcji obiektów klasy pochodnej **Pracownik** najpierw tworzone są podobiekty klasy **Osoba**. Natomiast podczas destrukcji najpierw wywoływany jest destruktor klasy **Pracownik**, a potem destruktor klasy bazowej **Osoba**.

21.4 Metody wirtualne i polimorfizm

W klasie pochodnej można zdefiniować metodę o sygnaturze i typie zwracanym (patrz rozdz. 11.2, str. 156) takich samych jak dla pewnej metody z klasy bazowej. Nie jest to przeciążenie, tylko **przesłonięcie**: sygnatury są bowiem te same (a funkcje przeciążane mają tę samą nazwę, ale *różne* sygnatury).

Załóżmy następującą sytuację:

```
class A {
    // ...
    void fun() { ... }
    // ...
};

class B : public A {
    // ...
    void fun() { ... }
    // ...
};
```

Zdefiniujmy teraz

```
A a, *pa  = new A, *pab = new B,
     &raa = a,     &rab = *pab;
```

A zatem

- a jest obiektem klasy **A**;
- pa jest wskaźnikiem typu **A*** do obiektu klasy **A**. Mówimy, że typem *statycznym* obiektu wskazywanego przez pa jest **A** i dynamicznym również **A**;
- pab jest wskaźnikiem typu **A*** do obiektu klasy **B**. Typem statycznym obiektu wskazywanego przez pab jest **A**, ale dynamicznym jego typ prawdziwy, czyli **B**;
- raa jest referencją typu **A&** do obiektu klasy **A**. Typem statycznym obiektu, do którego referencją jest raa jest typ **A**; dynamicznym również **A**;
- rab jest referencją typu **A&** do obiektu klasy **B**. Typ statyczny obiektu, do którego referencją jest rab to **A**, typ dynamiczny to **B**.

Przypomnijmy, że

> Typ *statyczny* obiektu wskazywanego przez wskaźnik lub referencję określony jest przez deklarację tego wskaźnika (referencji). Typ *dynamiczny* to rzeczywisty typ obiektu, do którego odnosi się ten wskaźnik lub to referencja. Typ dynamiczny obiektu może być zazwyczaj określony dopiero podczas wykonania programu; typ statyczny jest znany już w czasie kompilacji.

Przypuśćmy teraz, że wywołujemy funkcję **fun** za pomocą zmiennych a, pa, pab, raa i rab i pytamy, która z metod: czy ta z klasy **A**, czy ta z klasy **B**, zostanie wywołana. Otóż dla wszystkich wywołań decyduje tu typ *statyczny*; wywołania

```
a.fun(); pa->fun(); pab->fun(); raa.fun(); rab.fun();
```

wszystkie spowodują wywołanie funkcji **fun** z klasy **A**, mimo że w trzecim i piątym przypadku obiekt, na rzecz którego nastąpi wywołanie, jest w rzeczywistości obiektem klasy pochodnej **B**, a w klasie tej metoda **fun** została przedefiniowana, przesłaniając wersję odziedziczoną z klasy bazowej.

Dla znających Pythona czy Javę może to być zaskoczenie. Tam bowiem, jak w większości języków obiektowych, decyduje typ dynamiczny: jeśli obiekt, na rzecz którego wywołujemy metodę, jest klasy pochodnej względem tej, która jest typem statycznym wskaźnika (referencji) do tego obiektu, to wywołana będzie wersja tej metody pochodząca z klasy pochodnej (jeśli została tam przedefiniowana). Mówimy wtedy, że metody są **wirtualne**. A zatem w Javie wszystkie metody (prócz finalnych i prywatnych) *są* wirtualne. Klasy w których istnieją metody wirtualne, nazywamy klasami **polimorficznymi**, bo wywołanie ich poprzez wskaźnik (referencję) pewnego typu zależy od typu obiektu na który ten wskaźnik wskazuje, ma zatem „wiele kształtów". A zatem w Javie klasy, prócz finalnych, *są* polimorficzne.

Trzeba jednak zdawać sobie sprawę, że ceną za polimorfizm jest pewna utrata wydajności. Dla wywołań na rzecz obiektów klas niepolimorficznych odpowiednia metoda jest wybierana już w czasie kompilacji na podstawie typu statycznego. Mówimy, że następuje wtedy **wczesne wiązanie** (ang. *early binding*).

Typ dynamiczny obiektu wskazywanego przez wskaźnik lub referencję może być natomiast określony dopiero w czasie wykonania. Kompilator, napotkawszy wywołanie metody z klasy polimorficznej, nie może umieścić w pliku wykonywalnym kodu odpowiadającego wywołaniu konkretnej funkcji. Zamiast tego umieszczany jest tam kod sprawdzający prawdziwy typ obiektu i wybierający odpowiednią metodę. Mówimy, że następuje wtedy **późne wiązanie** (ang. *late binding*). Tak więc każde wywołanie metody wirtualnej powoduje narzut czasowy w trakcie wykonania. Wybranie odpowiedniej metody wymaga też dostępu do informacji o różnych wersjach metody w klasach dziedziczących. Informacja ta jest zwykle umieszczana w specjalnej tablicy, której adres jest przechowywany w *każdym* obiekcie klasy polimorficznej. Obiekt taki musi być zatem większy niż obiekt analogicznej klasy niepolimorficznej — polimorfizm powoduje zatem również narzut pamięciowy.

W C++, przede wszystkim właśnie ze względu na wydajność, podejście do polimorfizmu jest nieco inne niż w większości innych języków obiektowych. Jako programiści mamy mianowicie możliwość wyboru: czy chcemy, aby definiowana klasa była polimorficzna, czy też z polimorfizmu rezygnujemy na rzecz podniesienia wydajności. Domyślnie nowo definiowane klasy *nie są* polimorficzne, a zatem definiowane w nich metody *nie są* wirtualne. Jeśli w programie następuje wywołanie, poprzez wskaźnik lub referencję, dowolnej metody na rzecz obiektu klasy niepolimorficznej, kompilator umieszcza od razu wywołanie konkretnej metody w kodzie wynikowym. Kieruje się przy tym wyłącznie typem zadeklarowanym (statycznym) wskaźnika lub referencji.

Aby definiowana klasa była polimorficzna, wystarczy jeśli choć jedna metoda tej klasy będzie wirtualna. W szczególności *może* to być destruktor (ale *nie* konstruktor — ten wirtualny nie może być nigdy).

Deklaracja metody jako wirtualnej musi mieć miejsce w klasie bazowej. Jeśli metoda została zadeklarowana w klasie bazowej jako wirtualna, to wersje przesłaniające tę metodę we wszystkich klasach pochodnych (nie tylko „synach", ale i „wnukach", „prawnukach",...) są też wirtualne. Ponowne deklarowanie ich w klasach pochodnych jako wirtualnych jest dopuszczalne i zalecane, bo zwiększa czytelność kodu, ale w zasadzie zbędne.

Metodę deklarujemy jako wirtualną przez dodanie modyfikatora **virtual** w jej deklaracji. W klasach pochodnych, jak powiedzieliśmy, powtarzać tego nie musimy; na przykład:

```
class A {
    // ...
    virtual double fun(int,int);
    // ...
};

class B : public A {
    // ...
    double fun(int,int);
    // ...
};
```

Zdefiniujmy jak przedtem:

```
A a, *pa  = new A, *pab = new B,
     &raa = a,     &rab = *pab;
```

Teraz klasy są polimorficzne, metoda **fun** jest wirtualna, a więc zadziała mechanizm późnego wiązania. Wywołania

```
a.fun(), pa->fun(), pab->fun(), raa.fun(), rab.fun()
```

spowodują teraz w trzecim i piątym przypadku wywołanie metody **fun** z klasy **B**, gdyż:

- wywołanie jest poprzez wskaźnik lub referencję (typu bazowego: odpowiednio **A*** i **A&**);
- prawdziwym typem obiektu, na rzecz którego następuje wywołanie, jest typ pochodny **B**;
- metoda **fun** jest wirtualna;
- metoda **fun** została przedefiniowana (nadpisana) w klasie **B**.

Oczywiście, nic by się złego nie stało, gdybyśmy w klasie **B** nie przedefiniowali metody **fun**. Wywołana zostałaby tak czy owak wersja widoczna w klasie **B**; gdyby **fun** nie została przesłonięta, to w klasie **B** widoczną wersją metody **fun** byłaby ta odziedziczona z klasy **A**. Nawet zresztą gdy metodę przedefiniowaliśmy, możemy jawnie „wyłączyć" polimorfizm: wywołania

```
pab->A::fun();    rab.A::fun();
```

spowodują wywołanie wersji funkcji **fun** z klasy bazowej **A** nawet jeśli wersja przesłaniająca w klasie **B** istnieje i mimo że obiekt, na rzecz którego następuje wywołanie, jest typu **B**, a wywołanie jest poprzez wskaźnik lub referencję. Tak więc jawna kwalifikacja nazwy metody (za pomocą nazwy klasy) powoduje, że normalny mechanizm polimorfizmu nie jest używany — takie wywołanie będzie „wcześnie wiązane". Skoro tak, to możliwe jest też wywołanie

```
b.A::fun()
```

bezpośrednio na rzecz obiektu klasy **B** (nie poprzez wskaźnik lub referencję). Odwrotna sytuacja nie jest oczywiście możliwa nigdy: nie można, nawet używając nazw kwalifikowanych, wywołać metody z klasy **B** poprzez nazwę *obiektu* klasy bazowej **A** (a nie wskaźnika lub referencji):

```
a.B::fun() // Źle!!
```

byłoby nielegalne.

Przesłaniając w klasie pochodnej metodę dziedziczoną z klasy bazowej możemy zawęzić jej dostępność (ale nie rozszerzyć — odwrotnie niż w Javie!).

Jaka zatem będzie dostępność wirtualnej metody wywoływanej poprzez wskaźnik typu **A*** do obiektu klasy **B**, jeśli w klasie pochodnej **B** dostępność tej metody zawęziliśmy? Otóż będzie ona taka, jak w klasie, do której odnosi się wskaźnik lub referencja (typ statyczny), a *nie* taka jak w klasie obiektu (typ dynamiczny). Jeśli metoda jest publiczna w klasie bazowej, to odpowiednie wersje tej metody z klas pochodnych będą dostępne poprzez wskaźnik lub referencja do obiektu klasy bazowej, nawet jeśli w klasach pochodnych ta sama składowa jest prywatna! Rozpatrzmy przykład:

P184: *figur.cpp* Dostępność funkcji wirtualnych

```
#include <iostream>
using namespace std;

class Figura {
protected:
    int  height;
public:
    Figura(int height = 0) : height(height)
    { }

    virtual void what() const {
        cout << "Figura: h=" << height <<endl;
    }
};

class Prostokat : public Figura {
private:
    int  base;
    void what() const {
        cout << "Prostokat: (h,b)=(" << height
             << "," << base << ")\n";
    }
public:
    Prostokat(int height = 0, int base = 0)
        : Figura(height), base(base)
    { }
};

int main() {
    Figura    *f = new Prostokat(4,5)  , &rf = *f;
    Prostokat *p = new Prostokat(40,50);

      // what w Prostokat private, ale w Figura nie!
    f->what();                          // Prostokat
    rf.what();                          // Prostokat

      // p->what(); nie, bo what prywatne w Prostokat
      // Ale ponizsze legalne!
    ((Figura*)p)->what();               // Prostokat
    ((Figura&)*p).what();               // Prostokat

      // OK: wersja publiczna z klasy bazowej Figura
    p->Figura::what();                  // Figura
}
```

Wirtualna funkcja **what** jest w klasie **Figura** zadeklarowana jako **public** (linia 11). W klasie **Prostokat** metoda ta jest przesłaniana, a wersja przesłaniająca jest **private** (linia 19). Zmienne f i rf są typu **Figura*** i **Figura&**, ale obiektem przez nie wskazywanym jest obiekt klasy **Prostokat** (linia 30). Tak więc typem *dynamicznym* wskazywanego obiektu jest **Prostokat**, ale statycznym **Figura**. Przyjrzyjmy się wywołaniom z linii 34 i 35. Wywołujemy tam metodę **what** z klasy obiektu, czyli z klasy **Prostokat**, bo metoda jest wirtualna i brany jest pod uwagę typ dynamiczny. W tej klasie metoda **what** jest prywatna, ale mimo to wywołanie się powiedzie: typ statyczny obieku do którego odnoszą się zmienne f i rf to **Figura**, a tam metoda ta była publiczna.

```
Prostokat: (h,b)=(4,5)
Prostokat: (h,b)=(4,5)
Prostokat: (h,b)=(40,50)
Prostokat: (h,b)=(40,50)
Figura: h=40
```

Natomiast zakomentowane wywołanie z linii 37 nie powiodłoby się. Tam bowiem typem statycznym obiektu wskazywanego przez p jest obiekt klasy **Prostokat**, a w tej klasie metoda **what** jest prywatna.

Zwróćmy uwagę na wywołania z linii 39 i 40. Zmienna p jest co prawda wskaźnikiem do obiektu typu pochodnego (i na taki obiekt wskazuje), ale przed wywołaniem rzutujemy wartość tego wskaźnika na typ **Figura***, a zatem zmieniamy typ statyczny wskazywanego obiektu na **Figura**, w której to klasie **what** jest metodą publiczną — analogiczny mechanizm zastosowaliśmy dla referencji rf. Zatem oba te wywołania powiodą się.

Ostatnia linia wydruku jest rezultatem instrukcji z linii 43 programu. Obiektem wskazywanym przez zmienną wskaźnikową p jest co prawda obiekt klasy **Prostokat**, ale wywołaliśmy jawnie, poprzez kwalifikację nazwą zakresu, metodę **what** z klasy bazowej **Figura**. Wywołanie zatem nie jest polimorficzne. Było możliwe, gdyż w klasie **Figura** metoda **what** jest publiczna.

Przekonajmy się jeszcze, że polimorfizm rzeczywiście kosztuje, a zatem nie powinien być stosowany bez potrzeby. W przykładzie poniżej definiujemy trzy bardzo podobne klasy:

P185: ***polysiz.cpp*** Wzrost rozmiaru obiektów klas polimorficznych

```
1 #include <iostream>
2 using namespace std;
3
4 class A {
5     int i;
6 public:
7     A() : i(0)
8     { }
9 };
10
```

```
11 class B {
12     int i;
13 public:
14     B() : i(0)
15     { }
16     ~B()
17     { }
18 };
19
20 class C {
21     int i;
22 public:
23     C() : i(0)
24     { }
25     virtual ~C()
26     { }
27 };
28
29 int main() {
30    cout << "sizeof(A): " << sizeof(A) << endl;
31    cout << "sizeof(B): " << sizeof(B) << endl;
32    cout << "sizeof(C): " << sizeof(C) << endl;
33 }
```

Klasa **B** jest identyczna jak klasa **A**, tyle że definiuje destruktor (zresztą w tej klasie niepotrzebny). Destruktor ten nie jest wirtualny, a więc klasa nie jest polimorficzna. Natomiast klasa **C** jest identyczna jak **B**, ale jej destruktor jest wirtualny. Wystarczy to, aby klasa ta była polimorficzna (choć, póki co, żadna inna klasa z niej nie dziedziczy). W liniach 30-32 drukujemy rozmiary obiektów wszystkich trzech klas:

```
sizeof(A): 4
sizeof(B): 4
sizeof(C): 8
```

Widzimy, że dopóki klasa nie jest polimorficzna, rozmiar obiektu jest taki, jak wynika z rozmiaru pól. Samo dodanie metody (w tym przypadku destruktora) nie zwiększa rozmiaru obiektów. Natomiast dodanie polimorfizmu, jak dla klasy **C**, powoduje wzrost rozmiaru obiektu, w naszym przykładzie o cztery bajty. W przypadku tej prostej klasy oznacza to zatem wzrost o 100 %.

Często się zdarza, że pola klasy są głównie typu wskaźnikowego; obiekty takich klas zwykle nie są duże. Narzut spowodowany polimorfizmem, a więc obecnością dodatkowej informacji (o tzw. tablicy funkcji wirtualnych) w każdym obiekcie klasy może być więc dość spory.

21.5 Klasy abstrakcyjne

W C++ (podobnie jak w Javie) można definiować **klasy abstrakcyjne**. Klasami takimi będą klasy, w których pewne metody w ogóle nie są zdefiniowane, a tylko zadeklarowane. Takie metody powinny być oczywiście wirtualne — w dziedziczących klasach muszą być dostarczone konkretne implementacje tych metod, aby można było tworzyć ich obiekty. Obiektów samej klasy abstrakcyjnej tworzyć nie można, bo nie jest to klasa do końca zdefiniowana. Zazwyczaj służy tylko jako definicja interfejsu, czyli zbioru metod jakie chcemy implementować na różne sposoby w klasach dziedziczących.

Można natomiast tworzyć obiekty klas pochodnych (nieabstrakcyjnych), w których metody wirtualne zadeklarowane ale nie zdefiniowane w klasie bazowej zostały przesłonięte konkretnymi implementacjami (klasy takie stają się wtedy **konkretne**). Co bardzo ważne, do takich obiektów można się odnosić poprzez wskaźniki i referencje o typie statycznym abstrakcyjnej klasy bazowej.

Metodę wirtualną można zadeklarować jako **czysto wirtualną** pisząc po nawiasie kończącym listę argumentów '`=0`', na przykład:

```
virtual void fun(int i) = 0;
```

W ten sposób informujemy kompilator, że definicji tak zadeklarowanej metody może w ogóle nie być, a zatem cała klasa, w której tę metodę zadeklarowano, będzie abstrakcyjna.

W zasadzie, choć rzadko się to robi, metodę czysto wirtualną (albo *zerową*) *można* w klasie abstrakcyjnej zdefiniować, ale klasa *pozostaje przy tym abstrakcyjna* i nie można tworzyć jej obiektów. Tak czy owak, w klasach dziedziczących trzeba tę metodę przedefiniować, aby uczynić te klasy klasami konkretnymi, których obiekty będzie można tworzyć. Do wersji tej metody zdefiniowanej w abstrakcyjnej klasie bazowej odwołać się wtedy można poprzez jawną specyfikację zakresu ('`Klasa::fun()`').

Rozpatrzmy przykład:

P186: ***virtu.cpp*** Funkcje czysto wirtualne

```
#include <iostream>
#include <cmath>     // atan
using std::ostream; using std::cout; using std::endl;

class Figura {
protected:
    static const double PI;
public:
    virtual double getPole() const       = 0;
    virtual double getObwod() const      = 0;
    virtual void   info(ostream&) const  = 0;
    static  double totalPole(Figura* tab[], int size) {
        double suma = 0;
        for (int i = 0; i < size; ++i)
```

```
15              suma += tab[i]->getPole();
16          return suma;
17      }
18      static  Figura* maxObwod(Figura* tab[], int size) {
19          int ind = 0;
20          for (int i = 0; i < size; ++i)
21              if (tab[i]->getObwod() >
22                      tab[ind]->getObwod())
23                  ind = i;
24          return tab[ind];
25      }
26  };
27  const double Figura::PI = 4*atan(1.);
28  void Figura::info(ostream& str) const {
29      str << "Figura: ";
30  }
31
32  class Kolo : public Figura {
33      double promien;
34  public:
35      Kolo(double r) : promien(r){ }
36      double getPole()  const { return PI*promien*promien; }
37      double getObwod() const { return 2*PI*promien; }
38      void   info(ostream& str) const {
39          Figura::info(str);
40          str << "kolo o promieniu  " << promien;
41      }
42  };
43
44  class Kwadrat : public Figura {
45      double bok;
46  public:
47      Kwadrat(double s) : bok(s) { }
48      double getPole() const { return bok*bok; }
49      double getObwod() const { return 4*bok; }
50      void   info(ostream& str) const {
51          Figura::info(str);
52          str << "kwadrat o boku    " << bok;
53      }
54  };
55
56  int main() {
57      Figura* tab[] = { new Kolo(1.), new Kwadrat(1.),
58                        new Kolo(2.), new Kwadrat(3.)
59                      };
60      int size = sizeof(tab)/sizeof(tab[0]);
```

```
61      for (int i = 0; i < size; ++i) {
62          tab[i]->info(cout);
63          cout << endl;
64      }
65      Figura* maxobw = Figura::maxObwod(tab,size);
66      cout << "Suma pol: " << Figura::totalPole(tab,size)
67           << "\nFigura o najwiekszym obwodzie: ";
68      maxobw->info(cout);
69      cout <<  "\n ma obwod "
70           << maxobw->getObwod() << endl;
71      for (size_t i=0; i < std::size(tab); ++i) delete tab[i];
72  }
```

Klasa **Figura** jest tu abstrakcyjna, bo zawiera metody czysto wirtualne; trudno byłoby rozsądnie zaimplementować metody takie jak **getPole** czy **getObwod** dla figur geometrycznych „w ogóle". Dopiero dla konkretnych figur, jak kwadrat czy koło, można takie wielkości obliczać. Dlatego dopiero w klasach dziedziczących z klasy **Figura**, a opisujących konkretne figury, definiujemy implementacje metod wirtualnych. Taka hierarchia klas ma wiele zalet. Spójrzmy na funkcje statyczne z klasy **Figura**. Ich parametrem jest tablica wskaźników do figur. Jakich figur? Wskazywane obiekty nie będą obiektami typu **Figura** — takich się nie da utworzyć, bo klasa ta jest abstrakcyjna. Ale będą na pewno obiektami klas dziedziczących z **Figura**, w których *na pewno* będą zaimplementowane wszystkie metody. Tak więc wywoływanie tych metod poprzez wskaźniki (lub referencje) na rzecz obiektów wskazywanych powiedzie się, niezależnie od ich konkretnego typu. Zauważmy, że nie muszą to być obiekty tego samego konkretnego typu — w naszym przykładzie w tablicy są wskaźniki zarówno do kół jak i kwadratów. Co więcej, możemy dodać inne klasy dziedziczące z **Figura**, na przykład opisujące trójkąty, a sama klasa bazowa, w szczególności funkcje statyczne **totalPole** i **maxObwod**, nie ulegną żadnej zmianie! Tak więc abstrakcyjna klasa bazowa definiuje **interfejs** (czyli „sposób użycia") dla całej rodziny klas, nawet takich, które dopiero będą napisane.

Zauważmy też, że metoda **info** jest zadeklarowana jako czysto wirtualna, chociaż *jest* zaimplementowana (linie 28-30). Pozostaje jednak czysto wirtualna — wszystkie konkretne klasy dziedziczące *muszą* ją zaimplemetować. Do implementacji z abstrakcyjnej klasy bazowej mogą się jednak odwołać poprzez jej nazwę kwalifikowaną (linie 39 i 51).

Rozpatrzmy jeszcze jeden przykład klasy czysto abstrakcyjnej: klasa ta definiuje interfejs do tworzenia stosów i operowaniu na nich.

P187: *stack.cpp* Interfejs stosu z metodą fabrykującą

```
1  #include <iostream>
2  using namespace std;
3
4  class STACK
5  {
6  public:
```

```
7       virtual void push(int) = 0;
8       virtual int pop()      = 0;
9       virtual bool empty() const = 0;
10      static STACK* getInstance(int);
11      virtual ~STACK() { }
12  };
13
14  class ListStack: public STACK {
15
16      struct Node {
17          int   data;
18          Node* next;
19          Node(int data, Node* next)
20              : data(data), next(next)
21          { }
22      };
23
24      Node* head;
25
26      ListStack() {
27          head = nullptr;
28          cerr << "Tworzenie ListStack" << endl;
29      }
30
31      ListStack(const ListStack&) { }
32      void operator=(ListStack&) { }
33
34  public:
35      friend STACK* STACK::getInstance(int);
36
37      int pop() {
38          int   data = head->data;
39          Node* temp = head->next;
40          delete head;
41          head = temp;
42          return data;
43      }
44
45      void push(int data) {
46          head = new Node(data, head);
47      }
48
49      bool empty() const {
50          return head == nullptr;
51      }
52
```

```
53      ~ListStack() {
54          cerr << "Usuwanie ListStack" << endl;
55          while (head) {
56              Node* node = head;
57              head = head->next;
58              cerr << " usuwanie wezla" << node->data <<endl;
59              delete node;
60          }
61      }
62  };
63
64  class ArrayStack : public STACK {
65
66      int  top;
67      int* arr;
68      enum {MAX_SIZE = 100};
69
70      ArrayStack() {
71          top = 0;
72          arr = new int[MAX_SIZE];
73          cerr << "Tworzenie ArrayStack" << endl;
74      }
75
76      ArrayStack(const ArrayStack&) { }
77      void operator=(ArrayStack&) { }
78
79  public:
80      friend STACK* STACK::getInstance(int);
81
82      void push(int data) {
83          arr[top++] = data;
84      }
85
86      int pop() {
87          return arr[--top];
88      }
89
90      bool empty() const {
91          return top == 0;
92      }
93
94      ~ArrayStack() {
95          cerr << "Usuwanie ArrayStack z " << top
96               << " elementami wciaz na stosie" << endl;
97          delete [] arr;
98      }
```

```
99 };
100
101 STACK* STACK::getInstance(int size) {
102     if (size > 100)
103         return new ListStack();
104     else
105         return new ArrayStack();
106 }
107
108 int main() {
109
110     STACK* stack;
111
112     stack = STACK::getInstance(120);
113     stack->push(1);
114     stack->push(2);
115     stack->push(3);
116     stack->push(4);
117     cerr << "pop " << stack->pop() << endl;
118     cerr << "pop " << stack->pop() << endl;
119     delete stack;
120
121     stack = STACK::getInstance(50);
122     stack->push(1);
123     stack->push(2);
124     stack->push(3);
125     stack->push(4);
126     cerr << "pop " << stack->pop() << endl;
127     cerr << "pop " << stack->pop() << endl;
128     delete stack;
129 }
```

Klasa **STACK** zawiera deklaracje typowych metod do operowania na stosach (w tym przypadku stosach liczb całkowitych), oraz definicję funkcji statycznej **getInstance**, która zwraca wskaźnik do stosu. Żadnej implementacji metod niestatycznych nie ma; są one implementowane w dwóch klasach dziedziczących: **ListStack** i **ArrayStack**. W pierwszej z nich stos implementowany jest za pomocą listy jednokierunkowej, a w drugiej przez tablicę. W obu tych klasach konstruktor jest prywatny. Jedynym sposobem na utworzenie obiektów tych klas jest użycie statycznej funkcji „fabrykującej"**getInstance** z klasy bazowej — może ona to zrobić, gdyż została zaprzyjaźniona z obiema klasami konkretnymi (linie 35 i 80). Funkcja ta tworzy obiekt klasy **ListStack** albo **ArrayStack**, w zależności od rądanego rozmiaru stosu: dla małych stosów wybiera implementację za pomocą tablicy, a dla dużych za pomocą listy (linie 101-106). Zauważmy, że ponieważ w funkcji tej tworzone są obiekty klas **ListStack** i **ArrayStack**, jej definicja musiała zostać umieszczona *po* definicji klas dziedziczących, gdy kompilator już „wie", że żadna z nich nie pozostała klasą abstrakcyjną

(i zna rozmiar tworzonych obiektów).

Funkcja **main** jest klientem klasy **STACK**. Tworzone są tu dwa obiekty o typie statycznym **STACK**; typ dynamiczny jest w każdym przypadku inny, jak widzimy z wydruku

```
Tworzenie ListStack
pop 4
pop 3
Usuwanie ListStack
 usuwanie wezla2
 usuwanie wezla1
Tworzenie ArrayStack
pop 4
pop 3
Usuwanie ArrayStack z 2 elementami wciaz na stosie
```

Zauważmy, że oba stosy używane są w analogiczny sposób; w zasadzie klient nie wie, jakiej konkretnej klasy są zwrócone obiekty. Co więcej, nie musi nawet wiedzieć, że są tak naprawdę różnych klas.

Konstruktor kopiujący i operator przypisania zostały w obu klasach zdefiniowane jako prywatne (linie 31-32 i 76-77). Zapobiega to kopiowaniu i przypisywaniu obiektów, co nie miałoby sensu, bo obiekty klas **ListStack** i **ArrayStack** są zupełnie różne (mają inne pola, a nawet rozmiary).

21.6 Wirtualne destruktory

Aby zapewnić właściwe usuwanie obiektów, należy deklarować destruktor jako funkcję wirtualną, jeśli tylko mamy zamiar korzystać z publicznego dziedziczenia z danej klasy. Wywołując wtedy destruktor (poprzez operator **delete**) obiektu klasy pochodnej wskazywanego przez wskaźnik do klasy bazowej, wywołamy naprawdę destruktor dla całego obiektu. Dzięki polimorfizmowi wywołany będzie bowiem wtedy destruktor z klasy pochodnej, a destruktor podobiektu klasy bazowej będzie potem wywołany i tak, według normalnych zasad kolejności wywoływania destruktorów. Jak bowiem mówiliśmy, przy usuwaniu obiektu klasy pochodnej najpierw wykonywany jest destruktor dla części „własnej", a potem automatycznie wywoływany jest destruktor dla podobiektu klasy bazowej. Gdyby destruktor nie był wirtualny, to ponieważ typem statycznym jest wskaźnik do obiektu klasy bazowej, wywołany byłby od razu destruktor z tej klasy, który jednak nie wie na przykład o istnieniu składowych czy zasobów dodanych w klasie pochodnej.

W poniższym przykładzie obiekt klasy pochodnej **Pelne** jest wskazywany przez wskaźnik osoba typu **Nazwisko***, a więc wskaźnik do obiektu klasy bazowej **Nazwisko**.

P188: ***wirdes.cpp*** Wirtualny destruktor

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

```
3
4  class Nazwisko {
5      char* nazwis;
6  public:
7      Nazwisko(const char* n)
8          : nazwis(strcpy(new char[strlen(n)+1], n))
9      {
10         cout << "Ctor Nazwisko: " << nazwis << endl;
11     }
12
13     virtual
14     ~Nazwisko() {
15         cout << "Dtor Nazwisko: " << nazwis << endl;
16         delete [] nazwis;
17     }
18 };
19
20 class Pelne : public Nazwisko {
21     char* imie;
22 public:
23     Pelne(const char* i, const char* n)
24         : Nazwisko(n),
25           imie(strcpy(new char[strlen(i)+1], i))
26     {
27         cout << "Ctor Pelne, Imie: " << imie << endl;
28     }
29
30     ~Pelne() {
31         cout << "Dtor Pelne, Imie: " << imie << endl;
32         delete [] imie;
33     }
34 };
35
36 int main() {
37     Nazwisko* osoba = new Pelne("Jan", "Malinowski");
38     delete osoba;
39 }
```

Dzięki wirtualności destruktora, podczas usuwania obiektu (linia 38) będzie wywołany najpierw destruktor z rzeczywistej klasy obiektu, a więc z klasy **Pelne**. Zwolni on pamięć zajmowaną przez imię, a następnie, automatycznie, zadziała destruktor dla podobiektu klasy bazowej **Nazwisko**, który zwolni pamięć zajmowaną przez nazwisko:

```
Ctor Nazwisko: Malinowski
Ctor Pelne, Imie: Jan
Dtor Pelne, Imie: Jan
Dtor Nazwisko: Malinowski
```

Gdyby destruktor w klasie bazowej **Nazwisko** *nie* był zadeklarowany jako wirtualny (po wykomentowaniu linii 13), to wywołany byłby tylko destruktor z klasy określanej przez statyczny typ wskaźnika, a więc z klasy **Nazwisko**:

```
Ctor Nazwisko: Malinowski
Ctor Pelne, Imie: Jan
Dtor Nazwisko: Malinowski
```

i, jak widać, imię w ogóle nie zostałoby usunięte!

Zauważmy, że mamy tu do czynienia z pewną niekonsekwencją: destruktor w klasie pochodnej, **~Full**, nadpisuje destruktor z klasy bazowej, **~Name**, choć ich nazwy są inne. Pod tym względem destruktor jest wyjątkowy: w innych przypadkach metoda nadpisująca musi oczywiście mieć tę samą nazwę co odpowiednia metoda w klasie bazowej.

21.7 Wielodziedziczenie

Jak już wspominaliśmy, w C++ klasa może dziedziczyć z wielu klas bazowych. Jest to narzędzie pozwalające na tworzenie skomplikowanych hierarchii dziedziczenia i daje spore możliwości programistyczne. Z drugiej jednak strony, zbytnie skomplikowanie struktury klas dziedziczących prowadzi wtedy do trudnego do opanowania i modyfikowania kodu. W zasadzie wielodziedziczenia należy zatem unikać, jeśli nie jest niezbędne. Poniżej podamy tylko podstawowe informacje na temat dziedziczenia wielobazowego — szczegółów należy szukać w bardziej zaawansowanych podręcznikach.

Definiując klasę dziedziczącą z wielu klas wymieniamy je na liście dziedziczenia po kolei, oddzielając przecinkami. Dla każdej z nich z osobna podajemy specyfikator dostępu (**private**, **protected** lub **public**). Opuszczenie tego specyfikatora jest równoważne podaniu specyfikatora **private** dla klas, a **public** dla struktur. Wszystkie klasy występujące na liście dziedziczenia muszą być kompilatorowi znane, — nie wystarczy deklaracja zapowiadająca.

Tak więc

```
class C : public A, B {
    // ...
};
```

deklaruje klasę **C** dziedziczącą publicznie z klasy **A** oraz prywatnie z klasy **B**. W zasadzie dopuszczalne jest, aby obie klasy bazowe zawierały składniki o tej samej nazwie: w klasie pochodnej trzeba się wtedy do takich składników odnosić poprzez nazwę kwalifikowaną nazwą klasy (z czterokropkiem). Jednak trzeba pamiętać, że użycie kwalifikowanych nazw wyłącza polimorfizm.

Obiekty klasy pochodnej będą zawierać podobiekty wszystkich klas bazowych. Podczas tworzenia obiektów klasy pochodnej można więc jawnie wywoływać konstruktory dla dziedziczonych podobiektów; w przeciwnym przypadku użyte będą konstrukory domyślne. Oczywiście, jak zwykle, konstruktory klas bazowych mogą być wywołane tylko poprzez użycie listy inicjalizacyjnej.

W poniższym przykładzie definiujemy abstrakcyjną klasę **DoDruku** opisującą funkcjonalność „bycia drukowalnym”. Klasa ma jedno pole określające strumień wyjściowy oraz jedną czysto wirtualną metodę **druk**. Mamy również standardową klasę **Osoba**, dziedziczącą z **DoDruku**, a co za tym idzie definiującą metodę **druk**. Zauważmy, że na liście inicjalizacyjnej konstruktora tej klasy wywołujemy jawnie konstruktor **DoDruku** (linia 21) podając jako argument odpowiedni strumień (domyślnie jest to cout). Definiujemy też klasę abstrakcyjną **Figura** i dwie dziedziczące z niej klasy **Kolo** i **Kwadrat**. Obie implementują metodę **druk**, ale tylko **Kolo** dziedziczy również z **DoDruku**, a zatem na liście inicjalizacyjnej jej konstruktora pojawiają się jawne wywołania dwóch klas bazowych (linia 43).

P189: ***multbas.cpp*** Wielodziedziczenie

```
1  #include <iostream>
2  #include <string>
3  #include <cmath>
4  using namespace std;
5
6  class DoDruku {
7  protected:
8      ostream& str;
9  public:
10     DoDruku(ostream& str)
11         : str(str)
12     { }
13     virtual void druk() const = 0;
14 };
15
16 class Osoba : public DoDruku {
17     string imie;
18     int    ur;
19 public:
20     Osoba(string i, int u, ostream& str = cout)
21         : DoDruku(str), imie(i), ur(u)
22     { }
23
24     void druk() const {
25         str << imie + " (" << ur << ")" << endl;
26     }
27 };
28
29 class Figura {
30 protected:
31     static const double PI;
32     string name;
33 public:
34     virtual double getPole() const = 0;
```

```
35      Figura(string name) : name(name) { }
36  };
37  const double Figura::PI = 4*atan(1.);
38
39  class Kolo : public Figura, public DoDruku {
40      double prom;
41  public:
42      Kolo(string n, double r, ostream& str = cout)
43          : Figura(n), DoDruku(str), prom(r)
44      { }
45      double getPole() const { return PI*prom*prom; }
46      void   druk() const {
47          str << "kolo " << name << " o promieniu "
48              << prom << " i polu " << getPole() << endl;
49      }
50  };
51
52  class Kwadrat : public Figura {
53      double side;
54  public:
55      Kwadrat(string n, double s)
56          : Figura(n),side(s)
57      { }
58      double getPole() const { return side*side; }
59      void druk() const {
60          cout << "kwadrat " << name << " o boku "
61               << side << " i polu " << getPole() << endl;
62      }
63  };
64
65  void drukTable(DoDruku* tab[], int size) {
66      for (int i = 0; i < size; ++i)
67          tab[i]->druk();
68  }
69
70  int main() {
71      Kolo    ci1("pierwsze",2,cout),
72             *pci2 = new Kolo("drugie",3);
73      Kwadrat sq1("pierwszy",4),
74             *psq2 = new Kwadrat("drugi",5);
75      Osoba   ps1("Jim",1972),
76             *pps2 = new Osoba("Tom",1978,cout);
77
78      DoDruku* tab[] = {&ci1, &ps1, pci2, pps2};
79
80      cout << "** Drukowanie obiektow typu DoDruku" << endl;
```

```
81      drukTable(tab,4);
82
83      cout << "** Drukowanie kwadratow" << endl;
84      sq1.druk();
85      psq2->druk();
86
87      delete pci2;
88      delete psq2;
89      delete pps2;
90  }
```

W liniach 65-68 definiujemy wolną (globalną) funkcję drukującą obiekty typu **DoDruku**. Zauważmy, że w tablicy wskaźników przesyłanej do tej funkcji podczas jej wywołania (linia 81) mogliśmy umieścić adresy obiektów typu **Osoba** i **Kolo**, ale nie **Kwadrat**, chociaż ta klasa też definiuje metodę **druk**. Nie dziedziczy jednak z **DoDruk**, a więc wskaźnik do obiektu tej klasy nie może mieć typu statycznego **DoDruku***. Program drukuje

```
** Drukowanie obiektow typu DoDruku
kolo pierwsze o promieniu 2 i polu 12.5664
Jim (1972)
kolo drugie o promieniu 3 i polu 28.2743
Tom (1978)
** Drukowanie kwadratow
kwadrat pierwszy o boku 4 i polu 16
kwadrat drugi o boku 5 i polu 25
```

Charakterystyczna w powyższym przykładzie była klasa abstrakcyjna **DoDruku**. Deklaruje ona jedną czysto wirtualną metodę określającą pewną funkcjonalność klas, często zupełnie niezwiązanym, z niej dziedziczących: wszystkie na pewno zawierać będą metodę **druk** pozwalającą na drukowanie informacji o obiekcie. Takie proste klasy, zwykle abstrakcyjne, deklarujące pewną ogólną funkcjonalność, nazywamy klasami **domieszkowymi** (ang. *mix-in class*); odpowiadają one z grubsza interfejsom z Javy. O ile skomplikowane wielodziedziczenie, ogólnie rzecz biorąc, nie jest zalecane, jeśli tylko można go uniknąć, to dodawanie do listy dziedziczenia prostych klas domieszkowych jest stosunkowo bezpieczne i często bardzo wygodne.

— 22 —

Wyjątki

Tak jak Java,Python i wiele innych języków obiektowych, język C++ umożliwia obsługę **wyjątków** (ang. *exception*), czyli sytuacji, gdy powstaje w czasie wykonania programu błąd uniemożliwiający dalszy jego przebieg (na przykład dzielenie przez zero czy wywołanie metody za pomocą pustego wskaźnika). Normalnie, powstanie takiej sytuacji powoduje przerwanie programu, zwykle z mało czytelnym komunikatem o naturze błędu. Obsługa wyjątków pozwala na zdefiniowanie przez programistę akcji, które należy podjąć po powstaniu sytuacji wyjątkowej — w szczególności programista może świadomie podjąć decyzję o ich całkowitym zignorowaniu.

W profesjonalnych programach właściwa obsługa sytuacji wyjątkowych może stanowić znaczną część całego kodu; żaden bank nie chciałby bowiem, aby jego klientom ukazywał się nagle na ekranie komunikat *„segmentation fault, core dumped”* lub *„skontaktuj się ze sprzedawcą”*.

Właściwa i pełna obsługa sytuacji wyjątkowych to zadanie trudne. Jedną z przyczyn jest fakt, że język C++ był zaprojektowany tak, aby był zgodny z językiem C, gdzie obsługi wyjątków nie ma.

Podrozdziały:

22.1 Zgłaszanie wyjątków

W dowolnym miejscu programu może być **zgłoszony** (wysłany) wyjątek (ang. *throwing* lub *raising an exception*). Często nawet nie wiemy, że funkcja biblioteczna, którą wywołujemy, może to zrobić. Lepiej jednak o tym wiedzieć. Autor takiej funkcji bibliotecznej nie znał naszego programu, więc nie mógł wiedzieć, jak obsłużyć daną sytuację wyjątkową. Aby nie przerywać programu, zgłasza więc wyjątek, a użytkownik korzystający z tej funkcji wiedząc, że taki wyjątek może zostać przez funkcję zgłoszony, sam definiuje sposób jego obsługi. Widzimy zatem, że mechanizm obsługi wyjątków daje możliwość pewnego rodzaju komunikacji między różnymi fragmentami kodu, być może pochodzącymi z różnych modułów i napisanych przez różnych autorów.

W funkcjach, które sami piszemy, możemy określić, kiedy i jaki wyjątek zostanie zgłoszony. Wyjątek może być opisany obiektem dowolnego typu, klasowego lub

wbudowanego. Zazwyczaj tworzy się specjalne klasy, których obiekty będą opisywać wyjątki. Równie dobrze jednak wyjątek może być opisany po prostu liczbą lub napisem.

Wyjątek jest zgłaszany za pomocą operatora **throw**:

```
throw excpt;
```

gdzie excpt może być obiektem dowolnego typu, również wbudowanego, jak **int** czy **double**. W momencie zgłoszenia wyjątku normalny przebieg programu jest przerywany i poszukiwana jest procedura obsługi danego wyjątku (czyli odpowiednia fraza **catch**, o czym powiemy w następnym podrozdziale). Jeśli taka procedura zostanie znaleziona, to wyjątek uważa się za obsłużony i wykonywana jest treść procedury. Nie ma automatycznego powrotu do miejsca zgłoszenia wyjątku!

Jeśli taka procedura nie zostanie znaleziona w funkcji, w której ta sytuacja wyjątkowa miała miejsce, to przepływ sterowania opuszcza kod funkcji, a ramka stosu związana z jej wywołaniem jest usuwana (stos jest „zwijany"). Oznacza to, między innymi, że zmienne lokalne zdefiniowane w tej funkcji są bezpowrotnie tracone. Dla usuwanych lokalnych zmiennych obiektowych wywoływane są, co bardzo ważne, ich destruktory. Jeśli funkcja była rezultatowa, to wartość zwracana jest nieokreślona i wobec tego bezużyteczna. Na tej samej zasadzie poszukiwanie procedury obsługi jest następnie kontynuowane w funkcji wywołującej. Jeśli i tam nie zostanie znaleziona, to i ta funkcja przerywa swoje działanie i jej ramka wywołania na stosie jest też zwijana. W ten sposób mamy dwie możliwości:

- Odpowiednia procedura obsługi zostanie w końcu znaleziona. Wtedy sterowanie przechodzi do wykonania tej procedury, a następnie, jeśli program nie został w tej procedurze przerwany, jest kontynuowany od miejsca w programie za procedurą obsługi. Pamiętajmy, że *nie ma* powrotu do miejsca w programie, w którym nastąpiło zgłoszenie wyjątku (mówimy, że mechanizm obsługi wyjątków jest w C++ **niewznawialny**);

- Proces poszukiwania procedury obsługi dochodzi do funkcji **main** i tam jej również nie znajduje. W tej sytuacji następuje wyjście z programu poprzez wywołanie funkcji **terminate**, (z nagłówka ***exception***) która wywołuje z kolei funkcję **abort** kończącą program. Użytkownik może, za pomocą funkcji **set_terminate**, ustalić inną, przez siebie napisaną funkcję (bezrezultatową i bezparametrową) do pełnienia roli funkcji **terminate**. Tak czy owak, nie może z niej być powrotu: program musi się skończyć, ewentualnie po wykonaniu pewnych czynności porządkujących lub informacyjnych określonych w funkcji zastępującej **terminate**.

W poniższym programie podstawiamy funkcję **termin** zamiast domyślnej **terminate** (linia 18):

P190: ***term.cpp*** Nieobsłużone wyjątki

```
1 #include <iostream>
2 #include <cmath>      // sqrt
```

```
3 #include <cstdlib>    // exit
4 #include <exception>
5 using namespace std;
6
7 void termin() {
8     cout << "termin: exit(7)" << endl;
9     exit(7);
10 }
11
12 double Sqrt(double x) {
13     if (x < 0) throw "x < 0";
14     return sqrt(x);
15 }
16
17 int main() {
18     set_terminate(&termin);
19
20     double z, x;
21
22     x =  16;
23     z = Sqrt(x);
24     cout << "Sqrt(" << x << ")=" << z << endl;
25
26     x = -16;
27     z = Sqrt(x);
28     cout << "Sqrt(" << x << ")=" << z << endl;
29 }
```

Z kolei w funkcji **Sqrt** zgłaszamy wyjątek, jeśli przekazany argument jest ujemny. Wyjątku tego nie przechwytujemy (czyli nie obsługujemy). Zatem wywołana zostanie funkcja **termin**. Nie ma ona prawa powrócić do funkcji wywołującej, ale powoduje, za pomocą wywołania funkcji **exit** (z pliku nagłówkowego ***cstdlib***), zakończenie programu z tak zwanym statusem powrotu równym 7. Status ten jest odbierany przez powłokę systemu operacyjnego i może być w jakimś celu wykorzystany. W Linuksowych powłokach, jak ***tcsh*** czy ***bash***, status powrotu ostatnio wykonanego programu staje się wartością wbudowanej zmiennej powłoki o nazwie '$?' i wobec tego jest znany tuż po zakończeniu programu, co często wykorzystuje się w skryptach powłoki

```
cpp> g++ -pedantic-errors -Wall -o term term.cpp
cpp> ./term
Sqrt(16)=4
termin: exit(7)
cpp> echo $?
7
```

W tym przykładzie każde powstanie wyjątku kończy się przerwaniem programu (tyle że w sposób cywilizowany). Zwykle jednak nie o to nam chodzi: chcielibyśmy prze-

chwytywać wyjątki i sami decydować, czy i jak program ma być kontynuowany.

22.2 Wychwytywanie wyjątków

Do definiowania procedur obsługi wyjątków służą tzw. frazy **catch**. Mają one postać definicji funkcji, której parametr jest takiego typu, jakiego typu będzie obsługiwany przez tę frazę wyjątek. Każda fraza **catch** „zajmuje się" tylko wyjątkami określonego przez swój parametr typu. Każda też obsługuje wyłącznie wyjątki powstałe podczas wykonywania jednej instrukcji złożonej bezpośrednio poprzedzającej tę frazę — ta instrukcja złożona musi być poprzedzona słowem kluczowym **try**:

```
1    try {
2        // sekwencja instrukcji
3    }
4    catch(Typ t) {
5        // obsluga
6    }
7    // ...
```

W powyższym schematycznym przykładzie wyjątek pewnego typu **Typ** może zostać zgłoszony podczas wykonywania instrukcji złożonej oznaczonej słowem kluczowym **try**. Cała instrukcja złożona jest, jak zwykle, ujęta w nawiasy klamrowe i może zawierać dowolną liczbę instrukcji prostych i złożonych; mogą tam w szczególności występować wywołania funkcji. Jeśli wyjątek zostanie zgłoszony podczas wykonywania takiej funkcji, a nie został w niej obsłużony, to nastąpi powrót z niej i wyjątek będzie dalej obsługiwany tak, jakby został wysłany w miejscu, gdzie funkcja była wywołana.

Jeśli w trakcie wykonywania instrukcji złożonej z linii 1-3 wyjątek typu **Typ** (lub typu pochodnego od **Typ**) zostanie wysłany, to sterowanie przejdzie natychmiast do wykonywania instrukcji zawartych we frazie **catch**. Wewnątrz tej frazy dostępny będzie obiekt opisujący wyjątek (ten, który użyty został w odpowiedniej instrukcji **throw**). Składnia frazy **catch** przypomina składnię definicji funkcji. Również argument jest odbierany podobnie jak dla zwykłych funkcji. W szczególności można, i zazwyczaj tak się właśnie robi, odbierać go poprzez referencję. Możemy wtedy bowiem korzystać z dobrodziejstw polimorfizmu. Jeśli odbieramy ten argument przez wartość, to, jak dla funkcji, otrzymujemy *kopię* argumentu.

Czasem wystarczy nam informacja, że wyjątek danego typu został wysłany, a sam jego obiekt nie jest nam do niczego potrzebny i nie jest używany wewnątrz frazy **catch**. Wtedy w nagłówku frazy można pozostawić tylko określenie typu, a nazwę parametru pominąć.

Jeżeli wewnątrz procedury obsługi zawartej we frazie **catch** nie zostanie wysłany nowy wyjątek, nie nastąpi wykonanie instrukcji **return** lub zakończenie programu, to po jej wykonaniu sterowanie przejdzie do instrukcji następującej po frazie **catch** (linia 7 w naszym przykładzie).

Oczywiście do powstania sytuacji wyjątkowej podczas wykonywania bloku **try** nie musi dojść. Jeśli do niej nie dojdzie, to po zakończeniu wykonywania tego bloku

sterowanie przejdzie *za* frazę **catch** (czyli znów do linii 7 w naszym przykładzie); sama fraza **catch** zostanie wtedy całkowicie pominięta.

Rozpatrzmy następujący program:

P191: ***excpt.cpp*** Wyjątki jako obiekty klas polimorficznych

```
1  #include <iostream>
2  #include <string>
3  #include <sstream>    // ostringstream
4  #include <cmath>      // sqrt, log, atan
5  using namespace std;
6
7  struct Blad {
8      virtual string opis() = 0;
9  };
10
11 class Ujemna : public Blad {
12     double x;
13 public:
14     Ujemna(double x) : x(x) { }
15     string opis() {
16         ostringstream strum;
17         strum << "Ujemny argument: x = " << x;
18         return strum.str();
19     }
20 };
21
22 class PozaZakresem : public Blad {
23     double x, min, max;
24 public:
25     PozaZakresem(double x, double mi, double ma)
26         : x(x), min(mi), max(ma)
27     { }
28     string opis() {
29         ostringstream strum;
30         strum << "Argument x = " << x << "\n     "
31               << "    poza zakresem [" << min << ","
32               << max << "]";
33         return strum.str();
34     }
35 };
36
37 double fun(double x) {
38     if (x < 0 || x > 2) throw PozaZakresem(x,0,3);
39     return sqrt(x*(3-x));
40 }
41
```

```
42 double logPI(double x) {
43     static double LPI = log(4*atan(1.));
44     if (x <= 0) throw Ujemna(x);
45     return log(x)/LPI;
46 }
47
48 int main() {
49     double x = 4*atan(1.);
50
51     cout << "x = " << x << endl;
52     try {
53         double z1 = logPI(x);
54         cout << "  z1 = " << z1 << endl;
55         double z2 = fun(x);
56         cout << "  z2 = " << z2 << endl;
57     }
58     catch(Blad& blad) {
59         cerr << "  Blad: " << blad.opis() << endl;
60     }
61 }
```

Definiujemy w tym programie klasę abstrakcyjną **Blad** i dwie klasy z niej dziedziczące: **Ujemne** i **PozaZakresem**. Obie implementują wirtualną funkcję **opis** odziedziczoną z klasy bazowej.

W liniach 37-46 definiujemy dwie funkcje globalne. Każda z nich może zgłosić wyjątek: funkcja **fun** wyjątek klasy **PozaZakresem** jeśli argumentem jest liczbą spoza zakresu $[0, 3]$, a funkcja **logPI** wyjątek klasy **Ujemne** jeśli argument jest ujemny (funkcja oblicza logarytm argumentu przy podstawie π). W funkcji **main** wywołujemy w bloku **try** obie te funkcje; ewentualny błąd wychwytuje fraza **catch** zdefiniowana w liniach 58-60. Zauważmy, że przechwytuje ona wyjątki typu **Blad** *przez referencję*, a więc przechwyci wyjątki wszystkich klas pochodnych od **Blad** i będzie mogła korzystać z polimorfizmu. Tutaj tak właśnie jest, bo klasy są polimorficzne i funkcja **opis** jest wirtualna. Wywołanie tej metody w linii 59 spowoduje zatem wywołanie tej metody z prawdziwej klasy przekazanego poprzez argument obiektu,

```
x = 3.14159
  z1 = 1
  Blad: Argument x = 3.14159
        poza zakresem [0,3]
```

a więc w naszym przypadku z klasy **PozaZakresem**, bo wyjątek został zgłoszony w funkcji **fun**.

22.3 Hierarchie wyjątków

Za blokiem **try** można umieścić kilka następujących po sobie bloków **catch**. Jeśli w takiej sytuacji nastąpi zgłoszenie wyjątku, to podczas poszukiwania odpowiedniej procedury obsługi będzie sprawdzany typ wyjątków deklarowany w nagłówkach przez kolejne bloki **catch**. Jeśli (po ewentualnej konwersji niejawnej, jak w poprzednim przykładzie) znaleziony zostanie odpowiedni typ, to ta właśnie fraza **catch** zostanie użyta i na tym obsługa tego wyjątku zakończy się: pozostałe bloki **catch** zostaną już zignorowane, nawet jeśli jest wśród nich taki o bliższym dopasowaniu typu parametru z typem wyjątku. Konwersje w górę są wykonywane tylko dla argumentów obiektowych. Na przykład program

P192: *hier.cpp* Konwersje wyjątków

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      try {
6          throw 7;
7      }
8      catch(double) { cout << "double" << endl; }
9      catch(int   ) { cout << "int   " << endl; }
10 }
```

wydrukuje '**int**', chociaż gdyby traktować frazę **catch** jak funkcję, byłoby możliwe wywołanie funkcji oczekującej wartości typu **double**, jeśli argument jest typu **int**. Gdyby w linii 6 wysłanym obiektem była liczba '7.' (z kropką), to wybraną frazą **catch** byłaby ta z parametrem **double**.

Dla wyjątków typu obiektowego konwersja w górę *będzie* jednak zastosowana

P193: *hierob.cpp* Wyjątki typu obiektowego

```
1  #include <iostream>
2  using namespace std;
3
4  class A { };
5  class B : public A { };
6
7  int main() {
8      try {
9          throw B();
10     }
11     catch(A) { cout << "A" << endl; }
12     catch(B) { cout << "B" << endl; }
13 }
```

jak to widać z wydruku tego programu

```
cpp> g++ -pedantic-errors -Wall -o hierob hierob.cpp
hierob.cpp: In function `int main()':
hierob.cpp:12: warning: exception of type `B' will be
                    caught by earlier handler for `A'
cpp> ./hierob
A
```

Kompilator wypisał tu ostrzeżenie, bo w tym przykładzie druga fraza **catch** w ogóle nie jest osiągalna. To samo obowiązuje dla wyjątków wskaźnikowych; program

P194: *hier1.cpp* Wyjątki typu obiektowego

```
#include <iostream>
using namespace std;

struct A {
    const char* info() { return "A*"; }
};

struct B : A {
    const char* info() { return "B*"; }
};

int main()
{
    try {
        throw new B;
    }
    catch(A* a) { cout << a->info() << endl; }
    catch(B* b) { cout << b->info() << endl; }
}
```

drukuje

```
cpp> g++ -pedantic-errors -Wall -o hier1 hier1.cpp
hier1.cpp: In function `int main()':
hier1.cpp:18: warning: exception of type `B*' will be
                    caught by earlier handler for `A*'
cpp> ./hier1
A*
```

Tak więc, jeśli klasa **B** dziedziczy z **A**, wyjątek jest typu **B** (lub **B***), a fraza **catch** deklaruje typ **A** (lub **A*** lub **A&**), to wyjątek zostanie wychwycony przez tę frazę, jeśli:

- **A** i **B** są tym samym typem;
- **B** jest klasą dziedziczącą *publicznie* z **A** i *tylko* z **A** (**B** może dziedziczyć też z innych klas, ale nie publicznie);

- oba typy są wskaźnikowe, a typy wskazywane spełniają jeden z powyższych warunków;
- **catch** deklaruje **A&**, a typ **B** wyjątku spełnia któryś z wymienionych warunków.

Oczywiście, typy wyjątków obsługiwnae przez kolejne frazy **catch** nie muszą mieć ze sobą nic wspólnego: w każdym przypadku podczas poszukiwania procedury obsługi sprawdzane są kolejno aż do znalezienia typu odpowiadającego typowi wyjątku.

Czasem na samym końcu listy fraz **catch** umieszcza się taką frazę z trzema kropkami zamiast parametru

```
catch(...) {
    // ...
}
```

Oznacza ona *„wyłap każdy wyjątek, niezależnie od jego typu”*.

22.4 Wyjątki w konstruktorach i destruktorach

Sytuacja wyjątkowa może zdarzyć się podczas wykonywania konstruktora lub destruktora. Taka sytuacja jest szczególnie trudna do właściwej obsługi. Powiedzmy zatem o kilku sprawach, o których trzeba wtedy pamiętać.

Jeśli wyjątek został zgłoszony podczas konstrukcji obiektu, to obiekt ten nie powstanie, jego destruktor nie zostanie wywołany, a wszystkie do tej pory utworzone składowe zostaną usunięte. Mogą to być już utworzone składowe obiektowe: dla nich destruktory zostaną wywołane. Oczywiście powstanie kłopot, jeśli są w klasie składowe wskaźnikowe, a same obiekty, na które one wskazują, zostały w konstruktorze zaalokowane na stercie lub odnoszą się do zasobów systemowych, jak np. plików. Tego typu obiekty są zwykle usuwane (zwalniane) w destruktorze, ale on nie zadziała. W ten sposób, w razie wystąpienia sytuacji wyjątkowej, nieudany obiekt zostanie co prawda usunięty, ale zasoby (pamięć, otwarte pliki) nie zostaną zwolnione. Można temu zaradzić „opakowując” tego rodzaju składowe wskaźnikowe tak, aby uczynić z nich obiekty, dla których w razie niepowodzenia wywołany zostanie destruktor zwalniający zasoby. Rozpatrzmy przykład:

P195: *zasob.cpp* Zwalnianie zasobów w sytuacjach wyjątkowych

```
1 #include <iostream>
2 #include <cstring>
3 #include <cstdio>  // FILE, fopen, fclose
4 using namespace std;
5
6 class A {
7     struct nazw {
8         char* n;
9         nazw(const char* n)
```

```
10              : n(strcpy(new char[strlen(n)+1],n))
11          { }
12          ~nazw() {
13              cerr << "dtor nazw: " << n << endl;
14              delete [] n;
15          }
16      };
17
18      nazw Nazwisko;
19      FILE*    plik;
20  public:
21      A(const char* n, const char* p)
22          : Nazwisko(n)
23      {
24          plik = fopen(p,"r");
25          // ...
26  //      throw 1;
27          // ...
28      }
29
30      // inne pola i metody
31
32      ~A() {
33          cerr << "dtor A" << endl;
34          if (plik) fclose(plik);
35      }
36  };
37
38  int main() {
39      try {
40          A a("Kowalski","zasob.cpp");
41      } catch(...) {
42          cerr << "Nie udalo sie skonstruowac obiektu\n";
43      }
44  }
```

Klasa **A** zawiera składową opisującą nazwisko w postaci C-napisu. Sam napis alokowany jest dynamicznie, ale wskaźnik do niego nie jest bezpośrednio składową klasy **A**. Zamiast tego składową tej klasy jest *obiekt* pomocniczej, „opakowującej" struktury **nazw**, który dopiero zawiera, jako swoją składową `n`, wskaźnik do napisu. Ta pomocnicza struktura definiuje destruktor usuwający napis ze sterty.

Prócz nazwiska, klasa **A** zawiera pole wskaźnikowe wskazujące obiekt typu **FILE** (jest to standardowy typ w czystym C opisujący pliki).

Załóżmy, że linia 26 (**throw** `1`) jest wykomentowana. Konstruktor klasy **A** inicjuje składowe opisujące nazwisko i kończy się prawidłowo. Żaden wyjątek nie został zgłoszony. Po wyjściu sterowania z ciała bloku **try** obiekt klasy **A**, jako obiekt lokalny dla

tego bloku, jest usuwany i wywoływany jest jego destruktor zamykający plik. Następnie usuwane są obiekty składowe i wywoływane są ich destruktory, a więc w naszym przypadku usunięty będzie obiekt Nazwisko, a w jego destruktorze zwolniona zostanie pamięć na nazwisko. Wydruk programu

```
dtor A
dtor nazw: Kowalski
```

świadczy o tym, że obiekt a został prawidłowo usunięty.

Spróbujmy teraz uaktywnić linię 26, która powoduje powstanie sytuacji wyjątkowej w trakcie wykonywania konstruktora. Teraz wydruk z programu to

```
dtor nazw: Kowalski
Nie udalo sie skonstruowac obiektu
```

Po powstaniu wyjątku destruktor klasy **A** dla powstającego obiektu *nie* został wywołany. Tak więc plik, choć już otwarty, nie został zamknięty — przepadł tylko wskaźnik do niego. Natomiast napis zawierający nazwisko został prawidłowo usunięty! Stało się tak, bo powstanie wyjątku spowodowało wywołanie destruktorów dla już utworzonych składowych obiektowych, a więc dla składowej Nazwisko.

W przykładzie powyższym nie wyłapywaliśmy wyjątku powstającego podczas konstruowania obiektu w samym konstruktorze, ale pozwoliliśmy mu wyjść poza konstruktor, gdzie był przechwytywany w funkcji **main**. Inna jest sytuacja z wyjątkami, jakie mogą powstać w trakcie wykonania destruktora. Problem polega na tym, że destruktor, jak mówiliśmy, może zostać wywołany podczas zwijania stosu w poszukiwaniu procedury obsługi innego wyjątku. Powstanie dodatkowego nieobsłużonego wyjątku w destruktorze powodowałoby „podwójne" zwijanie stosu. Taka sytuacja nie jest w C++ możliwa; jeśli powstanie, program jest natychmiast kończony za pomocą funkcji **terminate**. Tak więc, jeśli jakikolwiek wyjątek może być zgłoszony podczas wykonywania destruktora, to należy go obsłużyć — przechwycić odpowiednią frazą **catch** — *wewnątrz* tego destruktora, nie dopuszczając do jego „ucieczki".

22.5 Specyfikacje wyjątków

Definiując funkcję można jawnie wyspecyfikować, czy w ogóle i jakiego typu wyjątki mogą zostać zgłoszone, a nieobsłużone podczas jej wykonywania.

Obecnie zaleca się stosowanie tylko dwóch form takich specyfikacji, jak pokazano poniżej:

```
int f(int i) noexcept;
int g(int i);
```

Deklarujemy tu, że funkcja **f** nie zgłasza wyjątku; jeśli się mimo to zdarzy, program natychmiast się zakończy przez wywołanie **terminate**. Funkcja **g** natomiast może zgłaszać dowolny wyjątek. Specyfikacja wyjątków należy do sygnatury funkcji i musi być powtórzona przy każdej deklaracji i (jednej) definicji.

Specyfikator **noexcept** może pobierać argument typu (konwertowalnego do) **bool**; dwie powyższe deklaracje są równoważne

```
int f(int i) noexcept(true);
int g(int i) noexcept(false);
```

noexcept może być też użyty jako operator; na przykład For example

```
void m(int) noexcept(noexcept(f(i)));
void n(int) noexcept(noexcept(g(i)));
```

znaczy, że funkcja **m** ma taką samą specyfikację jak **f** natomiast **n** taką samą jak **g** (oczywiście, `f(i)` i `g(i)` *nie* oznaczają tu rzeczywistego wywołania funkcji).

Jeśli taka fraza została podana dla metody wirtualnej, to wszystkie wersje przesłaniające tę metodę w klasach pochodnych mogą być mniej restrykcyjne, ale nigdy bardziej.

22.6 Ponawianie wyjątku

Często zdarza się, że przechwytując wyjątek chcemy wykonać pewne czynności (na przykład zwolnić zasoby), ale dalej nie wiemy, czy i jak kontynuować program. Przekazujemy wtedy ten sam wyjątek dalej, po częściowym obsłużeniu, tak aby mógł zostać obsłużony do końca na przykład w funkcji wywołującej. Oczywiście, jak mówiliśmy, nie może być sytuacji takiej, aby jednocześnie były obsługiwane dwa wyjątki. Ale wyjątek uważa się za już obsłużony, jeśli sterowanie weszło do frazy **catch**. Wewnątrz tego bloku można zatem zgłosić następny wyjątek, w szczególności powtórzyć zgłoszenie tego samego wyjątku który spowodował wejście do tego bloku **catch**. Wystarczy w tym celu, wewnątrz bloku, użyć instrukcji **throw** bez żadnego argumentu. Na przykład rozważmy taki schemat:

```
void fun() {
    // otwieramy gniazda
    try {
        // korzystamy z gniazd
    }
    catch(...)  // przechwytujemy wszystkie wyjatki
    {
        // zamykamy gniazda
        throw; // ponawiamy ten sam wyjatek
    }
    // zamykamy gniazda
}
```

Wewnątrz funkcji otwieramy połączenia internetowe. Korzystamy z nich, ale przewidujemy, że może być zgłoszony wyjątek. Zatem wyłapujemy go, zamykamy połączenia i wznawiamy ten sam wyjątek, aby funkcja wywołująca mogła go przechwycić

i zdecydować, co robić dalej. Zauważmy, że funkcja wywołująca nie miałaby możliwości prawidłowego zamknięcia połączeń, jeśli zostały one ustanowione lokalnie w funkcji **fun**.

22.7 Standardowe wyjątki

Standardowa biblioteka C++ definiuje wiele typów wyjątków, które są zgłaszane przez funkcje pochodzące z tej biblioteki (funkcje biblioteczne czystego C nie zgłaszają wyjątków). Wszystkie te typy wyprowadzone są z typu **exception**. Klasa ta deklaruje wirtualną metodę **what**, dostarczającą, w postaci C-napisu, informację o wyjątku. W klasach pochodnych od **exception** należy zatem zapewnić sensowną implementację tej metody.

Do najważniejszych wyjątków standardowych, które często przychodzi nam obsługiwać, należą:

bad_alloc (nagłówek ***new***) — Zgłaszany jest przez operator **new** w razie niemożności przydzielenia pamięci na tworzony obiekt.

bad_cast (nagłówek ***typeinfo***) — Zgłaszany jest przez operator **dynamic_cast**, , jeśli nie udała się żądana konwersja.

bad_typeid (nagłówek ***typeinfo***) — Zgłaszany jest przez operator **typeid**, jeśli wskaźnik będący jego argumentem jest pusty.

bad_exception (nagłówek ***exception***) — Zgłaszany jest, jeśli w funkcji wystąpi jakikolwiek nieobsłużony wyjątek nie wymieniony w specyfikacji wyjątków tej funkcji, pod warunkiem, że na liście specyfikacji znajduje się właśnie **bad_exception**. Pozwala to obsługiwać nieoczekiwane wyjątki o nieznanym zawczasu typie bez przerywania programu.

ios::failure (nagłówek ***ios***) — Zgłaszany jest, jeśli stan strumienia zmienił się w niepożądany sposób. Definicję tego niepożądanego stanu należy określić wywołując przedtem na rzecz obiektu strumienia metodę **exceptions**.

— 23 —

Moduły i przestrzenie nazw

Możliwość modularyzacji programu jest cechą wszystkich nowoczesnych języków. Polega ona na dostarczeniu programiście metod i narzędzi umożliwiających składanie dużych programów z oddzielnych fragmentów, które, z jednej strony, są od siebie oddzielone i mogą być rozwijane osobno, z drugiej zaś strony mogą być stosunkowo łatwo ze sobą łączone, pozwalając na szybkie tworzenie zaawansowanych aplikacji z gotowych „cegiełek". Inne mechanizmy pozwalają też uniknąć ewentualnych konfliktów nazw, które mogą się pojawić, szczególnie gdy moduły są rozwijane niezależnie przez różnych programistów.

Podrozdziały:

23.1 Moduły programu

Program w języku C++ może być fizycznie zapisany w wielu plikach. Jak wiemy z rozdz. 3 o dyrektywach preprocesora (str. 19), w pliku można umieścić polecenie włączenia innego pliku z kodem źródłowym. To, co zobaczy kompilator po przetworzeniu przez preprocesor, to tekst całości: dla kompilatora zatem będzie to jeden **moduł**, choć fizycznie zapisany jest w dwóch lub więcej plikach. Taki moduł zwany jest **jednostką translacji**. Z kolei cały program może składać się z wielu jednostek translacji, z których każda może być kompilowana osobno. Przy większych programach jest to bardzo istotne; zmiana wprowadzona w jednej jednostce powoduje konieczność ponownej kompilacji tej jednostki, ale nie zawsze całego programu.

Jednostki translacji mogą, i powinny, stanowić jednocześnie podstawę podziału programu na jednostki logiczne. Tak jak pewne powtarzalne pojedyncze zadania staramy się zapisać w postaci oddzielnych funkcji, tak zespół funkcji i klas dotyczących pewnego wycinka ogólnego zadania realizowanego przez cały program można zebrać w jednej jednostce translacji. Upraszcza to pisanie, analizowanie i pielęgnację kodu, szczególnie gdy przybiera on znaczne rozmiary i jest pisany czy modyfikowany przez wielu programistów.

Każda jednostka translacji kompilowana jest niezależnie, być może w innym czasie i na innym komputerze. Ponieważ w C++ sprawdzane są typy zmiennych i poprawność wywołań funkcji, wynika z tego, że każda funkcja, która jest w danej jednostce translacji używana (wywoływana), musi być w tej jednostce zadeklarowana. Natomiast definicja funkcji powinna być tylko jedna, umieszczona w jednej tylko jednostce translacji (nie dotyczy to funkcji rozwijanych, których *definicja* musi być widoczna

w każdej jednostce translacji w której są używane). Oczywiście wszystkie deklaracje i definicja funkcji muszą być zgodne.

Po połączeniu przez linker (program łączący), wszystkie funkcje z różnych jednostek translacji „widzą" się nawzajem bez dodatkowych zabiegów. Zatem nazwy funkcji globalnych należą do „uwspólnionego" zakresu złożonego z zakresów globalnych wszystkich modułów (mówimy, że są *eksportowane*). Wielu programistów umieszcza jednak słowo kluczowe **extern** przed deklaracją funkcji w jednostce translacji, w której nie ma definicji tej funkcji. Jest to pamiątka po czystym C, w C++ dopuszczalna, ale zbędna.

Wyjątkowo, funkcje globalne zdefiniowane ze specyfikatorem **static** *nie* są eksportowane (włączane do „uwspólnionego" zakresu globalnego); są widoczne tylko dla funkcji z tego samego modułu.

Inaczej rzecz się ma ze zmiennymi zadeklarowanymi w zasięgu globalnym. Tu uwspólnienia nie ma: zmienna globalna x z jednej jednostki translacji jest widoczna tylko wewnątrz tej jednostki; inny moduł może bezkonfliktowo zdefiniować zmienną globalną o tej samej nazwie i będą to dwie oddzielne zmienne, każda widoczna tylko w swoim module. Jeśli taką zmienną chcemy eksportować, to należy ją zdefiniować w jednym tylko module, a w pozostałych jednostkach translacji, w których będziemy z niej korzystać, zadeklarować ją jako zmienną zewnętrzną za pomocą specyfikatora **extern** (patrz rozdz. 7.3.2, str. 88). Jeśli natomiast, na odwrót, chcemy zdefiniować zmienną globalną i zagwarantować, że *nie* będzie ona dostępna w innych modułach, nawet jeśli, przypadkowo, będzie w nich zadeklarowana zmienna zewnętrzna (**extern**) o tej samej nazwie, to definiujemy ją z modyfikatorem **static** (patrz rozdz. 7.3.1, str. 85).

23.1.1 Pliki nagłówkowe i implementacyjne

Pisząc większy program, musimy godzić ze sobą dwa wymagania. Z jednej strony, program powinien być łatwy do zrozumienia, rozwijania i modyfikowania dla autorów programu. Z drugiej strony, pamiętać trzeba o wygodzie użytkownika — zapewnić trzeba przejrzysty interfejs pozwalający na efektywne korzystanie z programu i ewentualne jego rozwijanie bez konieczności wnikania w gąszcz szczegółów implementacyjnych. Tym celom służy podział programu na pliki o różnym charakterze.

Wiele jednostek kompilacyjnych może korzystać z tych samych funkcji, klas, szablonów, przestrzeni nazw, wyliczeń... Ich deklaracje muszą więc być dokładnie takie same. Moglibyśmy je oczywiście powtarzać we wszystkich modułach. Byłoby to jednak proszeniem się o kłopoty. Jakąkolwiek poprawkę czy zmianę trzeba by wtedy wprowadzać do każdego pliku, gdzie deklaracje te występują. Zamiast tego można zebrać je do jednego pliku i w tych modułach, gdzie są potrzebne i powinny być znane, włączać je za pomocą dyrektywy `#include` (rozdz. 3.2, str. 20). W plikach takich *nie* umieszczamy definicji funkcji czy metod klas, tylko ich deklaracje (z wyjątkiem funkcji rozwijanych, które powinny być tam umieszczone wraz z definicją). Pliki te stanowią właśnie interfejs, z którego odczytać można nazwy, typ, przeznaczenie i „instrukcje obsługi" deklarowanych obiektów. Dobrym zwyczajem jest wprowadzanie do takich plików precyzyjnych komentarzy. Pliki takie nazywamy **plikami nagłówko-**

wymi i tradycyjnie mają one rozszerzenie ***.h***. Są zwykle niewielkie, bo nie zawierają definicji.

Definicje zadeklarowanych w pliku nagłówkowym obiektów zbieramy z kolei w innym pliku, **pliku implementacyjnym**. Tu nie ma ogólnie przyjętej konwencji co do jego rozszerzenia, ale często stosuje się rozszerzenie ***.cxx*** lub ***.C***, lub po prostu ***.cpp***. Do tego pliku również włączamy za pomocą dyrektywy `#include` plik nagłówkowy z deklaracjami definiowanych funkcji czy metod. W ten sposób mamy gwarancję, że definicje będą spójne z deklaracjami (a więc z interfejsem), bo ewentualne niezgodności będą wtedy wychwycone i zgłoszone przez kompilator.

Przy bardziej skomplikowanej strukturze programu istnieje niebezpieczeństwo, że wskutek zagnieżdżenia dyrektyw `#include` ten sam plik nagłówkowy zostanie do tej samej jednostki translacji włączony więcej niż raz. Czasem nie ma w tym niczego złego, czasem może stwarzać problemy. Aby się przed tym uchronić, można stosować „sztuczkę" z użyciem dyrektywy `#ifndef` opisaną w rozdz. 3.2.

Mając już pliki nagłówkowy i implementacyjny, w których zebraliśmy zarówno interfejs, jak i implementację pewnej funkcjonalności (na przykład stosu czy drzewa poszukiwań binarnych), możemy ich użyć w wielu różnych aplikacjach, które tej funkcjonalności wymagają. Wystarczy wtedy

- w pliku źródłowym aplikacji włączyć plik nagłówkowy;
- zadbać o to, by w czasie łączenia (linkowania) programu był dostępny *skompilowany* kod plików implementacyjnych (kod źródłowy nie jest wtedy potrzebny).

Rozpatrzmy prosty przykład. Przy wielu okazjach przydaje się możliwość sortowania tablicy czy drukowania zawartości tablicy. Piszemy zatem moduł złożony z dwóch plików. W pliku nagłówkowym ***sortint.h*** deklarujemy funkcje do tego służące. Zamieszczamy komentarze mówiące, jak te funkcje stosować i do czego służą

P196: ***sortint.h*** Plik nagłówkowy

```
#ifndef _SORTINT_H
#define _SORTINT_H

// komentarze...

void sort(int[],int);
void pisztab(const int[],int);
#endif
```

W osobnym pliku, ***sortintImpl.cpp***, implementujemy te dwie funkcje.

P197: ***sortintImpl.cpp*** Plik implementacyjny

```
#include <iostream>
#include "sortint.h"    // wlaczamy naglowek
                        // z deklaracjami
using namespace std;

```

```
6  // implementacja funkcji sort
7  void sort(int a[], int size) {
8      int i, indmin = 0;
9      for (i = 1; i < size; ++i)
10         if (a[i] < a[indmin]) indmin = i;
11     if (indmin != 0) {
12         int p = a[0];
13         a[0] = a[indmin];
14         a[indmin] = p;
15     }
16
17     for (i = 2; i < size; ++i) {
18         int j = i, v = a[i];
19         while (v < a[j-1]) {
20             a[j] = a[j-1];
21             j--;
22         }
23         if (i != j ) a[j] = v;
24     }
25 }
26
27 // implementacja funkcji pisztab
28 void pisztab(const int t[], int size) {
29     cout << "[ ";
30     for (int i = 0; i < size; ++i)
31         cout << t[i] << " ";
32     cout << "]" << endl;
33 }
```

W tym pliku koniecznie włączamy za pomocą `#include` plik nagłówkowy. Nie jest on tu co prawda potrzebny, bo definicja jest jednocześnie deklaracją, ale chodzi o to, aby kompilator mógł sprawdzić zgodność definicji z interfejsem, jaki jest znany użytkownikowi, który ma dostęp tylko do pliku nagłówkowego. Plik implementacyjny możemy teraz skompilować z opcją '`-c`', aby utworzyć plik wynikowy, czyli kod binarny, ale jeszcze nie połączony (zlinkowany) do pliku wykonywalnego. Takie pliki mają zwykle rozszerzenie ***.o***.

```
cpp> ls                                                    (1)
sortintImpl.cpp  sortint.h                                 (2)
cpp> g++ -pedantic-errors -Wall -c sortintImpl.cpp         (3)
cpp> ls                                                    (4)
sortintImpl.cpp  sortint.h  sortintImpl.o                  (5)
```

W linii 1 powyższej sesji wypisujemy listę plików rozpoczynających się od '`sortint`'. W linii 3 kompilujemy plik implementacyjny ***sortintImpl.cpp*** z opcją '`-c`'. Jak widać w linii 5, pojawił się rzeczywiście plik ***sortintImpl.o***. Oczywiście, aby kompilacja się udała, musiał być dostępny plik nagłówkowy ***sortint.h*** który jest przez preprocesor

dołączany do pliku ***sortintImpl.cpp*** dając w sumie jedną jednostkę translacyjną.

Przypuśćmy teraz, że piszemy aplikację ***sortintApp.cpp*** w której nasze funkcje chcemy wykorzystać. Aby móc je zastosować, włączamy w aplikacji, za pomocą dyrektywy `#include`, *tylko* plik nagłówkowy. Zawiera on deklaracje funkcji **sort** i **pisztab**, a to wystarczy kompilatorowi aby sprawdzić poprawność ich wywołań.

P198: ***sortintApp.cpp*** Aplikacja

```
#include "sortint.h"  // tylko plik naglowkowy!
#include <iostream>
using namespace std;

int main() {
    int tab[] = {9,7,2,6,4,5,6,2,7,9,2,9,5,2},
        size  = sizeof(tab)/sizeof(tab[0]);

    cout << "Tablica  oryginalna: ";
    pisztab(tab, size);

    sort(tab, size);

    cout << "Tablica posortowana: ";
    pisztab(tab, size);
}
```

Kompilujemy teraz, być może po roku od utworzenia pliku ***sortintImpl.o***, kod źródłowy naszej aplikacji ***sortintApp.cpp***. Podczas kompilacji dołączamy dla linkera plik ***sortintImpl.o*** — otrzymujemy plik wykonywalny ***sortintApp***, który po uruchomieniu drukuje wyniki programu. Zauważmy, że nie potrzebowaliśmy tu pliku źródłowego ***sortintImpl.cpp***.

```
cpp> ls
sortintApp.cpp  sortint.h  sortintImpl.o
cpp> g++ -o sortintApp sortintApp.cpp sortintImpl.o
cpp> ./sortintApp
Tablica  oryginalna: [ 9 7 2 6 4 5 6 2 7 9 2 9 5 2 ]
Tablica posortowana: [ 2 2 2 2 4 5 5 6 6 7 7 9 9 9 ]
```

Załóżmy, że po upływie następnego roku zorientowaliśmy się, że użyty tu algorytm sortowania przez wstawianie jest nieefektywny i należy zastąpić go algorytmem sortowania przez kopcowanie. Co musimy zrobić? Zmienić implementację tej funkcji w pliku ***sortintImpl.cpp***, skompilować go i plik wynikowy dostarczyć użytkownikowi aplikacji. Użytkownik musi jeszcze raz zlinkować program (jeśli zachował plik wynikowy '***.o***' swojej aplikacji, to nie musi jej nawet powtórnie kompilować). Samo łączenie (linkowanie) jest proste i szybkie. Interfejs opisany plikiem nagłówkowym nie zmienił się, a zatem nie trzeba zmieniać ani jednej linijki w aplikacji użytkownika. Skompilowany kod implementacyjny zamieszcza się często w bibliotekach, na przykład tzw. bibliotekach dzielonych, zwykle w plikach o rozszerzeniu ***.so*** (*shared object*)

pod Linuksem i **.dll** (*dynamic-link library*) w systemie Windows. Wtedy wystarczy „podmienić" pliki biblioteczne, a wszystkie korzystające z nich programy powinny działać bez żadnych dodatkowych zabiegów (tyle że lepiej, bo przecież poprawiliśmy implementacje biblioteki).

Na tej zasadzie zorganizowane jest środowisko C/C++. Programista dołącza w pisanej przez siebie aplikacji pliki nagłówkowe zawierające deklaracje potrzebnych mu narzędzi. Pliki te umieszczone są zwykle w katalogu **include** instalacji C/C++ na danej platformie. Są to najczęściej zwykłe pliki tekstowe, które można i warto przeglądać. Implementacja zadeklarowanych w plikach nagłówkowych obiektów (funkcji, klas...) jest zawarta, w binarnej, skompilowanej postaci, w plikach bibliotecznych — zwykle w katalogu **lib**.

23.2 Przestrzenie nazw

Aby ułatwić jeszcze bardziej tworzenie w C++ dużych, pisanych przez wielu programistów aplikacji, dodano do języka mechanizm **przestrzeni nazw** (ang. *name space*). Jest to narzędzie pozwalające logicznie grupować nazwy obiektów rozmaitego typu: zmiennych, funkcji, klas, wyliczeń, wzorców itd.

Przypuśćmy, że stworzyliśmy w programie zestaw klas i funkcji do obsługi graficznego interfejsu użytkownika (GUI), a ktoś inny napisał zestaw realizujący tzw. *business logic*. Możemy te dwa zestawy umieścić w osobnych przestrzeniach nazw za pomocą słowa kluczowego **namespace**, po którym, w nawiasach klamrowych, umieszczamy deklaracje i ewentualnie definicje:

```
namespace GUI {
    class Menu { /* ... */ };
    void show(Menu&);
    double calculate(double) {
        // ...
    }
    // ...
}

namespace Business {
    class Tax { /* ... */ };
    double calculate(double&);
    // ...
}
```

Zauważmy, że w obu przestrzeniach nazw występuje funkcja **calculate**. W jednej z nich została już zdefiniowana, w drugiej tylko zadeklarowana. Nie powoduje to żadnego konfliktu, bo obie funkcje należą do różnych przestrzeni nazw i nie mają ze sobą nic wspólnego, niezależnie od tego, czy ich sygnatura jest taka sama czy inna. Jeśli umieściliśmy nazwę jakiegoś obiektu (stałej, klasy, funkcji, wyliczenia...) w przestrzeni nazw, to pełną, kwalifikowaną nazwą tego obiektu staje się nazwa obiektu poprzedzona identyfikatorem przestrzeni nazw i czterokropkiem. Na przykład, aby

zdefiniować funkcję **calculate** zadeklarowaną w przestrzeni nazw **Business**, musimy użyć w definicji pełnej nazwy (chyba, że samą definicję umieścimy wewnątrz bloku **namespace**):

```
double Business::calculate(double& z) {
    // ...
}
```

a żeby zdefiniować poza zakresem przestrzeni nazw np. destruktor klasy **Menu** z przestrzeni nazw **GUI**:

```
GUI::Menu::~Menu() { /* ... */ }
```

Do istniejących przestrzeni nazw można dodawać nowe elementy. Na przykład tenże destruktor moglibyśmy zdefiniować tak

```
namespace GUI {
    Menu::~Menu() { /* ... */ }
    const double PI = 3.14;
}
```

Zauważmy, że nazwy klasy **Menu** nie musieliśmy teraz kwalifikować nazwą **GUI**, bo całą definicję umieściliśmy w zakresie tej przestrzeni poprzez użycie słowa kluczowego **namespace**. Przy okazji dodaliśmy do tej przestrzeni nazw nowy element w postaci nazwy stałej zmiennopozycyjnej PI.

Funkcje **calculate** z obu przestrzeni można wywoływać bezkonfliktowo:

```
double x, y, v, w;
// ...
x = GUI::calculate(v);
y = Business::calculate(w);
```

W ten sposób nazwy z różnych przestrzeni nazw są od siebie odseparowane. Jeśli na przykład różne fragmenty aplikacji lub bibliotek piszą różni autorzy, to każdy może stworzyć dla swoich klas czy funkcji oddzielną przestrzeń nazw i nie przejmować się ewentualnym konfliktem nazw; użytkownik jego klasy będzie musiał świadomie ją wybrać kwalifikując nazwę klasy nazwą przestrzeni nazw.

Z drugiej strony, jeśli jesteśmy pewni, że żadnego konfliktu nie będzie, to konieczność poprzedzania wszystkich identyfikatorów nazwą przestrzeni może być nieco uciążliwa. Można temu zaradzić poprzez **deklarację użycia**. Deklaracja taka, wyrażona za pomocą słowa kluczowego **using**, informuje kompilator, że pewna nazwa oznaczać będzie obiekt nią identyfikowany z pewnej konkretnej przestrzeni nazw, stając się jej synonimem (aliasem). Na przykład po

```
using GUI::calculate;
```

w zasięgu, w którym deklaracja ta została użyta, niekwalifikowana nazwa **calculate** będzie się odnosić do takiej nazwy z przestrzeni **GUI**, a zatem będzie synonimem

nazwy **GUI::calculate**. Oczywiście do **calculate** z przestrzeni **Business** w dalszym ciągu możemy się odnosić, ale tę nazwę będziemy musieli kwalifikować, czyli pisać jawnie **Business::calculate(x)**.

Można w końcu, co nie jest na ogół zalecane, włączyć do aktualnego zakresu *wszystkie* nazwy z pewnej przestrzeni nazw. Służy do tego **dyrektywa użycia**, wyrażona przez dwa słowa kluczowe **using namespace**. Na przykład po

```
using namespace GUI;
```

w aktualnym zasięgu można używać *wszystkich* nazw z przestrzeni **GUI** bez kwalifikowania ich nazwą tej przestrzeni.

Jeśli przestrzeń nazw, którą za pomocą dyrektywy użycia „otwieramy" jest duża i nam niezbyt dobrze znana, to taką dyrektywą można doprowadzić do konfliktów nazw, uniknięciu których przestrzenie nazw miały przecież służyć.

Rozpatrzmy jeszcze prosty przykład:

P199: *nmspc.cpp* Przestrzenie nazw

```
1  #include <iostream>
2
3  namespace A {
4      const int two = 2;
5      const int six = 6;
6      void write() { std::cout << "ns-A" << " "; }
7  }
8
9  namespace B {
10     void write() { std::cout << "ns-B" << " "; }
11 }
12
13 namespace C {
14     const int two = 22;
15     const int six = 66;
16 }
17
18 int main() {
19     using A::write;
20     using namespace C;
21
22     write();
23     B::write();
24     std::cout << six << std::endl;
25 }
```

Mamy tu trzy przestrzenie nazw. W programie głównym deklarujemy (linia 19), że niekwalifikowanej nazwy **write** będziemy w zakresie funkcji **main** używać jako synonimu nazwy **A::write**. Otwierając w linii 20 przestrzeń nazw **C** powodujemy, że

wszystkie nazwy z tej przestrzeni mogą być używane wewnątrz funkcji **main** bez kwalifikowania ich nazwą przestrzeni. Dlatego stała six użyta w linii 24 odnosi się do stałej tak nazwanej w przestrzeni **C**, a nie z przestrzeni **A**. Oczywiście funkcja **write** z przestrzeni **B** jest w dalszym ciągu dostępna, tyle że jej użycie wymaga nazwy kwalifikowanej (linia 23). Program drukuje '`ns-A ns-B 66`'.

Zauważmy, że nazwy cout i endl w tym programie kwalifikujemy nazwą przestrzeni nazw **std** (linie 6, 10 i 24). Włączyliśmy bowiem nagłówek ***iostream***, ale *nie* napisaliśmy zaraz potem sakramentalnego

```
using namespace std;
```

Otóż wszystkie udogodnienia biblioteki standardowej, do których mamy dostęp po włączeniu plików nagłówkowych (między innymi pliku ***iostream***), są umieszczane w przestrzeni nazw **std**. Wyjątkiem są funkcje **operator new** i **operator delete** oraz makra preprocesora. Normalnie, dla uproszczenia, w naszych przykładowych programach otwieraliśmy na samym początku całą tę przestrzeń nazw, właśnie za pomocą powyższej dyrektywy użycia. Dzięki temu nie musieliśmy kwalifikować jej nazwą takich nazw, jak cout czy endl. Teraz natomiast tej przestrzeni nie otworzyliśmy, więc nazwy z niej pochodzące musieliśmy kwalifikować.

Przestrzenie nazw to stosunkowo nowy element języka C++. W C tradycyjnie używa się również plików nagłówkowych, ale wszystkie nazwy w nich deklarowane są jednakowo dostępne bez żadnych kwalifikacji, bo nie ma tam mechanizmu przestrzeni nazw. Trzeba było zatem zapewnić możliwość kompilowania przez kompilatory C++ programów napisanych w C. Osiągnięto to poprzez umowę, że jeśli użyjemy tradycyjnej (pochodzącej z C) nazwy pliku nagłówkowego, to odpowiedni plik jest włączany i zadeklarowane w nim nazwy są dodawane do domyślnej przestrzeni nazw. Jeśli natomiast ten sam plik nagłówkowy włączymy pod „nową" nazwą, to nazwy w nim deklarowane są dodawane do przestrzeni nazw **std**. Przyjęto przy tym konwencję, że pliki nagłówkowe o tradycyjnej nazwie ***nazwa.h*** są w C++ nazywane ***cnazwa***. Tak więc dyrektywa preprocesora

```
#include <string.h>
```

jest równoważna

```
#include <cstring>
```

tyle że w pierwszym przypadku zawartość jest włączana do domyślnej przestrzeni nazw, zaś w drugim do przestrzeni **std**. To samo dotyczy następujących 18 plików nagłówkowych pochodzących z C:

Tablica 23.1: Pliki nagłówkowe z C w C++

C	C ++	C	C ++	C	C ++
assert.h	cassert	ctype.h	cctype	errno.h	cerrno
float.h	cfloat	iso646.h	ciso646	limits.h	climits
locale.h	clocale	math.h	cmath	setjmp.h	csetjmp

Tablica 23.1: Pliki nagłówkowe z C w C++

signal.h	csignal	stdarg.h	cstdarg	stddef.h	cstddef
stdio.h	cstdio	stdlib.h	cstdlib	string.h	cstring
time.h	ctime	wchar.h	cwchar	cwtype.h	cwctype

Plik ***iostream.h*** jest bardzo często używany zamiast ***iostream***. Zauważmy, że nie pochodzi on w ogóle z C, więc powyżej opisana zasada go nie dotyczy. Nie powinno się go nigdy używać, gdyż nie należy do standardu i w związku z tym może w ogóle nie istnieć! Zawsze należy stosować prawidłową nazwę ***iostream***.

Pozostałe 32 standardowe pliki nagłówkowe *nie* pochodzą z C i są charakterystyczne tylko dla C++:

Tablica 23.2: Pliki nagłówkowe C++

algorithm	iomanip	list	ostream	streambuf
bitset	ios	locale	queue	string
complex	iosfwd	map	set	typeinfo
deque	iostream	memory	sstream	utility
exception	istream	new	stack	valarray
fstream	iterator	numeric	stdexcept	vector
functional	limits			

Zakończmy ten rozdział inną wersją przykładu ***stack.cpp*** (str. 480). Teraz nazwy z modułu opisującego stos umieszczamy w przestrzeni nazw **mySTACKS**, co zapobiega konfliktom między nazwami zadeklarowanymi w tym module a nazwami pochodzącymi z innych przestrzeni nazw. Nagłówek deklarujący abstrakcyjną klasę **STACK** umieszczamy w pliku ***mySTACK.h***

P200: ***mySTACK.h*** Nagłówek klasy abstrakcyjnej

```
1  #ifndef mySTACK_H
2  #define mySTACK_H
3
4  namespace mySTACKS {
5      class STACK
6      {
7      public:
8          virtual void push(int) = 0;
9          virtual int pop()      = 0;
10         virtual bool empty()   = 0;
11         static STACK* getInstance(int);
12         virtual ~STACK() { }
13     };
14 }
15 #endif
```

Z kolei nagłówek dla implementacji umieszczamy w pliku ***mySTACKS.h*** — włącza

on ***mySTACK.h*** i dodaje do przestrzeni nazw **mySTACKS** klasy konkretne, które będą implementować klasę abstrakcyjną **STACK**:

P201: ***mySTACKS.h*** Nagłówek dla klas implementujących

```
#ifndef mySTACKS_H
#define mySTACKS_H

#include "mySTACK.h"

namespace mySTACKS {

    class ListStack: public STACK {
        struct Node {
            int   data;
            Node* next;
            Node(int data, Node* next);
        };
        Node* head;
        ListStack();
    public:
        friend STACK* STACK::getInstance(int);
        int pop();
        void push(int data);
        bool empty();
        ~ListStack();
    };

    class ArrayStack : public STACK {
        int  top;
        int* arr;
        ArrayStack();
    public:
        friend STACK* STACK::getInstance(int);
        void push(int data);
        int pop();
        bool empty();
        ~ArrayStack();
    };
}
#endif
```

Plik z implementacją, ***mySTACKSImpl.cpp***, włącza nagłówek ***mySTACKS.h*** (który z kolei włącza ***mySTACK.h***) i dostarcza implementacji klas konkretnych

P202: ***mySTACKSImpl.cpp*** Implementacja

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

```
3  using namespace mySTACKS;
4
5  // ListStack
6
7  ListStack::Node::Node(int data, Node* next)
8      : data(data), next(next)
9  { }
10
11 ListStack::ListStack() {
12     head = nullptr;
13     std::cerr << "Creating ListStack" << std::endl;
14 }
15
16 int ListStack::pop() {
17     int   data = head->data;
18     Node* temp = head->next;
19     delete head;
20     head = temp;
21     return data;
22 }
23
24 void ListStack::push(int data) {
25     head = new Node(data, head);
26 }
27
28 bool ListStack::empty() {
29     return head == nullptr;
30 }
31
32 ListStack::~ListStack() {
33     std::cerr << "Deleting ListStack" << std::endl;
34     while (head) {
35         Node* node = head;
36         head = head->next;
37         std::cerr << " node " << node->data << std::endl;
38         delete node;
39     }
40 }
41
42 // ArrayStack
43
44 ArrayStack::ArrayStack() {
45     top = 0;
46     arr = new int[100];
47     std::cerr << "Creating ArrayStack" << std::endl;
48 }
```

```
49
50 void ArrayStack::push(int data) {
51     arr[top++] = data;
52 }
53
54 int ArrayStack::pop() {
55     return arr[--top];
56 }
57
58 bool ArrayStack::empty() {
59     return top == 0;
60 }
61
62 ArrayStack::~ArrayStack() {
63     std::cerr << "Deleting ArrayStack with " << top
64               << " elements remaining" << std::endl;
65     delete [] arr;
66 }
67
68 // STACK
69
70 STACK* STACK::getInstance(int size) {
71     if (size > 100)
72         return new ListStack();
73     else
74         return new ArrayStack();
75 }
```

Możemy teraz skompilować ten plik, aby usyskać plik binarny ***mySTACKSImpl.o***

```
cpp> g++ -pedantic-errors -Wall -c mySTACKSImpl.cpp
```

Tylko plik ***mySTACKSImpl.o*** i plik nagłówkowy ***mySTACK.h*** jest teraz potrzebny, aby skompilować aplikację wykorzystującą nasz moduł:

P203: ***stacksApp.cpp*** Aplikacja

```
1 #include <iostream>
2 #include "mySTACK.h"
3
4 int main() {
5
6     mySTACKS::STACK* stack;
7
8     stack = mySTACKS::STACK::getInstance(120);
9     stack->push(1);
10    stack->push(2);
11    stack->push(3);
```

```
12      stack->push(4);
13      std::cout << stack->pop() << " ";
14      std::cout << stack->pop() << std::endl;
15      delete stack;
16
17      stack = mySTACKS::STACK::getInstance(50);
18      stack->push(1);
19      stack->push(2);
20      stack->push(3);
21      stack->push(4);
22      std::cout << stack->pop() << " ";
23      std::cout << stack->pop() << std::endl;
24      delete stack;
25  }
```

Kompilacja:

```
cpp> g++ -pedantic-errors -Wall -o stacksApp \
              stacksApp.cpp mySTACKSImpl.o
```

tworzy plik wynikowy ***stacksApp***, który po uruchomieniu daje

```
cpp> ./stacksApp
Creating ListStack
4 3
Deleting ListStack
 node 2
 node 1
Creating ArrayStack
4 3
Deleting ArrayStack with 2 elements remaining
```

— 24 —

Biblioteka standardowa

Jak wszystkie nowoczesne języki programowania, C++ dostarcza, prócz narzędzi związanych z samym językiem, bibliotekę gotowych funkcji, klas, szablonów itd. Biblioteka podzielona jest na wiele części: aby z nich korzystać, w tworzonym programie wystarczy użyć odpowiednich dyrektyw `#include` włączających pliki nagłówkowe opisujące funkcjonalność poszczególnych modułów biblioteki (na temat plików nagłówkowych patrz rozdz. 23.1, str. 505). Implementacja funkcji, klas itd. z biblioteki standardowej, w postaci plików binarnych, dostarczana jest przez producenta kompilatora C++ (można również korzystać z innych, niezależnych implementacji, zarówno komercyjnych jak i typu *open source*).

Biblioteka standardowa nie jest tak bogata jak w innych językach, gdyż nie uwzględnia takich zagadnień jak programowanie sieciowe, graficzne, bazodanowe itd. Skupia się głównie na implementacji kolekcji (kontenerów) i algorytmów na tych kolekcjach operujących. Ta część biblioteki standardowej nazywana jest, ze względów historycznych, standardową biblioteką wzorców (szablonów), czyli STL (od ang. *Standard Template Library*).

W odróżnieniu od Javy i wielu innych języków, biblioteka standardowa C++ dostarcza przede wszystkim *szablonów*; konkretyzując te szablony użytkownik tworzy klasy i funkcje operujące na (prawie) dowolnych typach danych. Ważną rolę pełnią w szczególności szablony klas kolekcyjnych, pozwalające tworzyć kolekcje obiektów o różnych własnościach i funkcjonalności. Elementy wstawiane do kolekcji mogą być typu wbudowanego (w szczególności mogą to być wskaźniki), albo typu obiektowego (w szczególności definiowanego przez nas samych). Zwykle typy takie powinny spełniać pewne wymagania, aby mogły współdziałać z algorytmami z biblioteki standardowej; na przykład posiadać domyślny i kopiujący/przenoszący konstruktor albo przeciążone operatory porównania czy przypisania.

Oczywiście nie cała biblioteka standardowa ma formę wzorców: w poprzednich rozdziałach poznaliśmy, niejako mimochodem, inne składniki tej biblioteki, które szablonami nie są, takie jak udogodnienia dostarczane przez dołączenie plików nagłówkowych ***cstdlib***, ***cstring***, ***cmath*** (te akurat składniki biblioteki są odziedziczone z C).

Rozdział niniejszy należy traktować jako wstęp do opisu biblioteki standardowej — więcej szczegółów można znaleźć w książkach cytowanych w rozdz. 1.2, str. 4.

Podrozdziały:

24.1 Kolekcje i iteratory

Podstawowym udogodnieniem dostarczanym przez bibliotekę standardową są **kolekcje**. Kolekcja jest strukturą danych złożoną z danych pewnego typu i wyposażoną w zestaw operacji, jakie na tych danych można wykonywać. Może to być dodawanie elementów, ich usuwanie, wyszukiwanie, porównywanie, kopiowanie, porządkowanie itd. W bibliotece standardowej zaimplementowano takie podstawowe struktury danych, jak listy, stos, kolejki, kopce, tablice asocjacyjne (słowniki), zbiory; korzystając z nich użytkownik może łatwo i efektywnie tworzyć własne, bardziej złożone struktury, jak choćby różnego typu drzewa czy grafy.

24.1.1 Wektory

Najprostsza kolekcja opisywana jest szablonem **vector** dołączanym poprzez włączenie nagłówka ***vector***. Modelem wektora jest tablica elementów tego samego typu, ale o nieokreślonej zawczasu pojemności. Elementy są umieszczane w dobrze zdefiniowanym porządku i mamy do nich dostęp swobodny, to znaczy znając pozycję (indeks) elementu możemy pobrać jego wartość lub go zmienić czy usunąć bez konieczności przeglądania wszystkich elementów wektora. Elementy możemy dodawać na dowolnej pozycji, choć najefektywniejsze (w stałym, niezależnym od rozmiaru wektora, czasie) jest dodawanie na koniec. W miarę jak elementów przybywa, wektorowi przydzielana jest automatycznie większa pamięć — programista nie musi się już martwić zarządzaniem pamięcią, jej przydzielaniem i zwalnianiem. Ponieważ **vector** jest szablonem klasy, a nie klasą, obiekty-wektory możemy tworzyć konkretyzując wzorzec dla pewnego typu — może to być również typ wbudowany, jak **int** czy **double**. Domyślny konstruktor tworzy wektor pusty (są też inne konstruktory). Do utworzonego wektora dodajemy elementy (takiego typu, jak ten dla którego wzorzec został skonkretyzowany) za pomocą metody **push_back**, która dodaje *kopię* obiektu na końcu wektora (choć obiekty tymczasowe mogą być przenoszone, a nie kopiowane, co oczywiście może być znacznie efektywniejsze). Kopię, a zatem trzeba zadbać, aby obiekt był dobrze „kopiowalny", a więc na przykład miał prawidłowo zdefiniowany konstruktor kopiujący (lub przenoszący, jeśli elementy przenosimy). Istnieje też metoda **emplace_back** pobierająca argumenty dla konstruktora obiektu i tworząca ten obiekt bezpośrednio w wektorze.

Elementy można pobierać na różne sposoby. Najprościej to zrobić za pomocą indeksowania: robimy to dokładnie tak jak dla normalnej tablicy (pierwszy element ma, jak zwykle, indeks zero). Taki sposób jest bardzo efektywny, ale nie jest sprawdzane, czy użyty indeks jest legalny (tak jak nie jest to sprawdzane w przypadku normalnych tablic). Jeśli chcemy, aby zostało sprawdzone, czy indeks mieści się w dozwolonym zakresie, czyli od zera do liczby o jeden mniejszej niż aktualny rozmiar wektora, to

można użyć metody **at** podając indeks jako argument. Przy tej metodzie dostępu zgłaszany jest wyjątek **out_of_range** (z nagłówka ***stdexcept***), gdy indeks jest nielegalny. Wyjątek ten możemy obsłużyć:

P204: ***at.cpp*** Wektory

```
#include <vector>
#include <stdexcept>
#include <iostream>
#include <string>
using namespace std;

int main() {
    vector<string> vs;                                          ①

    vs.push_back("Ola");
    vs.push_back("Ula");
    vs.push_back("Ela");
    vs.push_back("Ala");

    try {
        for ( int i = 0; i < 5 /* BLAD */; i++ )                ②
            cout << vs.at(i) << " " ;
    } catch(out_of_range) {
        cout << "\n*** Zly indeks! *** "
             << " wektor ma tylko " << vs.size()
             << " elementy!" << endl;
    }
    cout << endl;

    cout << "Pierwszy element: " << vs.front() << endl;         ③
    cout << "Ostatni  element: " << vs.back()  << endl;         ④

    vs.pop_back();                                              ⑤

    cout << "Po pop_back: ";
    int size = (int)vs.size();                                  ⑥
    for ( int i = 0; i < size; i++)
        cout << vs[i] << " " ;
    cout << endl;
}
```

W poniższym programie pętla przebiega 5 razy (②), choć w wektorze są tylko cztery elementy. Podczas ostatniego obrotu zgłaszany jest zatem wyjątek (ale jest on obsłużony):

```
Ola Ula Ela Ala
```

```
*** Zly indeks! ***  wektor ma tylko 4 elementy!

Pierwszy element: Ola
Ostatni  element: Ala
Po pop_back: Ola Ula Ela
```

Jak widzimy (①), obiekt vs jest obiektem klasy **vector<string>** powstałej z konkretyzacji szablonu **vector** dla typu **string**.

Kolekcja jest zatem kolekcją elementów, z których każdy jest obiektem klasy **string**. Metoda **size**, użyta w linii ⑥, zwraca, jak łatwo się domyślić, rozmiar kolekcji (nie tylko wektorów). Typem zwracanym jest pewien typ całkowity bez znaku, którego aliasem jest nazwa **vector::size_type** — dlatego użyliśmy rzutowania — można tu było użyć zwykłego **int**'a, ale narazilibyśmy się na ostrzeżenia kompilatora. Metoda **pop_back** (⑤) usuwa ostatni element kolekcji, ale go nie zwraca. Metody **front** i **back** (③ i ④) zwracają, przez referencję, pierwszy i ostatni element kolekcji bez ich usuwania.

24.1.2 Iteratory

Podstawowym narzędziem do wykonywania operacji na kolekcjach są **iteratory**. Pozwalają one poruszać się po elementach kolekcji w sposób niezależny od ich rzeczywistego typu. Są swego rodzaju uogólnieniem wskaźników. Ich typ jest zdefiniowany w definicji wzorca szablonu kolekcji za pomocą instrukcji **typedef** i ma postać **kol<Typ>::iterator**, gdzie kol jest nazwą szablonu kolekcji, a **Typ** jest typem użytym do skonkretyzowania tego szablonu. Użytkownik nie musi wiedzieć, jaki naprawdę ten typ jest. Na przykład nazwą (aliasem) typu iteratora związanego z wektorem liczb typu **int** jest **vector<int>::iterator**. Kolekcja może być kolekcją elementów niemodyfikowalnych (lub chcemy ją tak traktować); dla każdej kolekcji zatem jest też zdefiniowany osobny typ **const_iterator** (na przykład **vector<int>::const_iterator**).

Iteratory mają semantykę wskaźników do elementów tablicy, to znaczy, jeśli it jest iteratorem wskazującym na pewien element kolekcji, to `*it` jest referencją do wskazywanego elementu (a więc l-wartością), a `it->sklad` jest nazwą składowej sklad wskazywanego przez iterator obiektu (jeśli taka składowa istnieje). Podobnie, po '`++it`' iterator it wskazuje na następny element kolekcji.

Istnieją w klasach kolekcyjnych dwie ważne bezargumentowe metody zwracające iteratory: metoda **begin** i **end**. Metoda **begin** zwraca iterator wskazujący na pierwszy element kolekcji. Nieco bardziej skomplikowany jest wynik metody **end**. Nie jest to iterator wskazujący na ostatni element kolekcji, jak by się mogło wydawać, ale na nieistniejący element „tuż za ostatnim". [Biblioteka definiuje też globalne funkcje, zdefiniowane w nagłówku *iterator*, **begin** i **end**, które pobierają kolekcję a zwracają te same iteratory]. Jeśli przebiegamy kolekcję za pomocą iteratora, to osiągnięcie przez niego wartości zwracanej przez **end** oznacza, że przebiegliśmy już całą kolekcję. Zamiast więc sprawdzać, jak zwykle w pętlach przebiegających tablice, czy indeks jest mniejszy od wymiaru tablicy, sprawdzamy, czy iterator wciąż *nie* jest równy iteratorowi zwracanemu przez metodę **end**; jeśli jest, to pętlę należy przerwać, bo iterator wskazuje na element nieistniejący:

P205: *iter.cpp* Iteratory

```
#include <vector>
#include <iostream>
#include <string>
using namespace std;

int main() {
    vector<string> vs{"Ola", "Ula", "Ela","Ala"};

    for ( vector<string>::iterator ite = vs.begin();
                                   ite!= vs.end(); ++ite)
        cout << *ite << " ";
    cout << endl;

    // albo

    vector<string>::iterator it, kon = vs.end();

    for ( it = vs.begin(); it != kon; ++it )
        cout << *it << " ";
    cout << endl;

    // albo

    using SIT=vector<string>::iterator;        ①

    SIT iter, koniec = vs.end();

    for ( iter = vs.begin(); iter != koniec; ++iter )
        cout << *iter << " ";
    cout << endl;

    // a najlepiej

    for ( auto i = vs.begin(); i != vs.end(); ++i )
        cout << *i << " ";
    cout << endl;
}
```

Powyższy program ilustruje sposób konstruowania pętli z użyciem iteratorów. Jeśli w programie potrzebujemy wielu zmiennych iteratorowych danego typu, to wygodnie jest temu typowi nadać jakąś krótszą nazwę za pomocą **typedef** lub **using**, jak to zrobiliśmy w linii ①. Tak jak mówiliśmy, `++it` przesuwa iterator na następną pozycję w kolekcji, a `*it` oznacza element w kolekcji wskazywany przez iterator `it`.

Jeśli kolekcja jest niemodyfikowalna, to nie tylko możemy, ale musimy użyć ite-

ratora typu **const_iterator**. Z taką sytuacją spotykamy się bardzo często, gdy przesyłamy kolekcję jako argument funkcji, która z założenia nie powinna elementów tej kolekcji zmieniać, a wobec tego jej parametr jest typu ustalonego:

P206: *constit.cpp* Iteratory do obiektów niemodyfikowalnych

```
#include <vector>
#include <iostream>
using namespace std;

struct Person {
    char name[20];
    int  year;
    void print() const {
        cout << name << "-" << year << endl;
    }
} john = {"John",25}, mary = {"Mary",18}, sue = {"Sue",9};


void printPerson(const vector<const Person*>& list) {
    for (auto it = list.cbegin(); it != list.cend(); it++)
        (*it)->print();
}

int main() {
    vector<const Person*> list;

    list.push_back(&john);
    list.push_back(&mary);
    list.push_back(&sue);

    printPerson(list);
}
```

Zauważmy, że słowo kluczowe **const** pojawia się tu kilkakrotnie. W określeniu typu

```
vector<const Person*>
```

oznacza, że kolekcja jest wektorem wskaźników do ustalonych elementów, a zatem, że poprzez odwołania się do obiektów klasy **Person** wskazywanych przez elementy kolekcji nie można zmienić tych *obiektów*. Z kolei pierwsze **const** w określeniu typu parametru funkcji **printPerson** oznacza, że jest to kolekcja niemodyfikowalnych elementów, czyli niemodyfikowalnych wskaźników, bo taki jest typ tych elementów. A zatem wewnątrz funkcji **printPerson** nie można zmienić ani adresów zapisanych we wskaźnikach będących elementami kolekcji, ani obiektów przez te wskaźniki wskazywanych. Ponieważ funkcja, poprzez typ zadeklarowanego parametru, „obiecała”, że nie zmieni obiektów wskazywanych, a wywołuje na rzecz tych obiektów metodę **print**,

metoda ta *musiała* być zadeklarowana jako **const**. Funkcja „obiecała" też, że nie zmieni elementów kolekcji, czyli wskaźników, zatem użyty iterator *musiał* tu być typu **const_iterator**. Zauważmy, że użyliśmy tu metod **cbegin** i **cend** — są one analogiczne do **begin** i **end**, ale zwracają **const_iterator**.

Parametr funkcji **printPerson** jest typu referencyjnego. Jest to zwykle pożądane przy przesyłaniu do funkcji kolekcji — w przeciwnym przypadku kopiowane byłyby całe kolekcje, wraz ze wszystkimi elementami. Tego problemu nie było przy przesyłaniu tablic, kiedy tak naprawdę przesyłany był tylko wskaźnik do pierwszego elementu.

Prócz iteratorów **iterator** i **const_iterator** istnieje typ iteratora odwrotnego **reverse_iterator**, za pomocą którego można przebiegać kolekcje w odwrotnym kierunku; na przykład program

P207: ***revit.cpp*** Iteratory odwrotne

```
#include <vector>
#include <iostream>
#include <string>
using namespace std;

int main() {
    vector<string> vs{"Ola", "Ula", "Ela", "Ala"};

    for (auto r = vs.rbegin(); r != vs.rend(); ++r)
        cout << *r << " ";
    cout << endl;
}
```

drukuje

```
Ala Ela Ula Ola
```

Użyliśmy tu metod **rbegin** i **rend**, które zwracają właśnie iteratory odwrotne. Istnieje też **const_reverse_iterator**.

Ze względu na swoją funkcjonalność iteratory dzielą się na kilka kategorii. Z różnymi kolekcjami związane są różne kategorie iteratorów.

Tablica 24.1: Iteratory związane z kolekcjami

Kolekcja	Iterator
vector	dostępu bezpośredniego
list	dwustronny
forward_list	jednokierunkowy
deque	dostępu bezpośredniego
map	dwustronny
multimap	dwustronny
set	dwustronny

Tablica 24.1: Iteratory związane z kolekcjami

multiset	dwustronny
string	dostępu bezpośredniego
array	dostępu bezpośredniego
valarray	dostępu bezpośredniego

Na przykład iterator związany z wektorem (**vector**) pozwala nie tylko na przesuwanie się w obu kierunkach za pomocą operatorów zwiększania i zmniejszania, ale na stosowanie arytmetyki wskaźników, to znaczy np. dodanie liczby 3 do takiego iteratora daje iterator przesunięty o trzy elementy do przodu względem wyjściowego. Podobnie, po

```
vector<string> vs;
// ...
auto it = vs.begin() + vs.size()/2;
```

iterator it wskazuje na środkowy element wektora. Iteratory takie można też porównywać za pomocą operatorów relacyjnych (jak np. '>') czy odejmować (wynikiem jest liczba elementów pomiędzy pozycjami w kolekcji wskazywanymi przez te iteratory). Elementy wskazywane takimi iteratorami można odczytywać, jak i na nie przypisywać. Tego typu iteratory to **iteratory o dostępie bezpośrednim** (ang. *random access iterator*). Są one np. związane z wektorami (**vector**), kolejkami dwustronnymi (**deque**) i obiektami klasy **string**, które są traktowane jako kolekcje znaków.

Drugim typem iteratora jest **iterator dwukierunkowy** (ang. *bidirectional iterator*). Pozwala na poruszanie się po kolekcji w obu kierunkach za pomocą operatorów zwiększania i zmniejszania ('++' i '−−'), ale nie wspiera arytmetyki wskaźników, na przykład dodawania liczb całkowitych do iteratora. Takie iteratory można porównywać za pomocą '==' i '!=', ale nie za pomocą '<' i '>'. Są one związane na przykład z kolekcjami listowymi (**list**, z nagłówka ***list***).

Jeszcze bardziej ograniczona jest funkcjonalność iteratorów **jednokierunkowych** (ang. *forward iterator*). Są one podobne do dwukierunkowych, ale pozwalają na poruszanie się tylko w jednym kierunku: do przodu.

W końcu najbardziej ograniczone są możliwości iteratorów **wejściowych** i **wyjściowych** (ang. *input* i *output iterator*). Pozwalają one tylko na jednokrotny przebieg (ang. *single pass*) kolekcji w kierunku do przodu. Iteratory wejściowe pozwalają tylko na odczyt elementu wskazywanego, a wyjściowe tylko na przypisanie im wartości, ale nie odczyt. Takie iteratory związane są zwykle ze strumieniami (wejściowymi i wyjściowymi).

Tablica 24.2: Operacje iteratorowe

	Wyj.	Wej.	W przód	Dwukier.	Bezpośr.
Odczyt	Nie	Tak	Tak	Tak	Tak
Zapis	Tak	Nie	Tak	Tak	Tak
Iteracja	++	++	++	++, −−	++, −−, +, −, +=, −=

Tablica 24.2: Operacje iteratorowe

Porówn.	==, ! =	==, ! =	==, ! =	==, ! =, <, >, <=, >=

Iteratory są podstawowym elementem stosowanym w algorytmach i funkcjach operujących na kolekcjach. Pełnią one rolę łącznika między algorytmami (funkcjami) a strukturami danych, jakimi są kolekcje.

24.1.3 Operacje na kolekcjach

Na kolekcjach można wykonywać wiele operacji, choć należy pamiętać, że nie każdą z nich można wykonać na każdej kolekcji. Związane jest to z kwestią wydajności: niektóre kolekcje dopuszczają mniej operacji, ale za to są one wykonywane bardzo efektywnie. Na przykład, jak wspominaliśmy, różnica dwóch iteratorów odnoszących się do wektora daje liczbę elementów pomiędzy nimi — ponieważ w wektorze elementy są rozmieszczone w pamięci jeden przy drugim i rozmiar ich jest znany, jest to operacja bardzo szybka, sprowadzająca się do jednego odejmowania i jednego dzielenia. Dla listy (**list**) nie jest to takie proste, więc aby osiągnąć ten sam cel, należy użyć specjalnej funkcji **distance** o liniowym (a więc proporcjonalnym do rozmiaru listy) czasie działania:

P208: ***dist.cpp*** Operacje na kolekcjach

```
#include <iostream>
#include <vector>
#include <string>
#include <list>
using namespace std;

int main() {
    vector<string> vec;

#if   defined(__WIN32)
    cout << "Podaj slowa (^Z konczy):\n";
#elif defined(__linux)
    cout << "Podaj slowa (^D konczy):\n";
#else
    #error Nieznany system
#endif

    string s;
    while ( cin >> s ) vec.push_back(s);
    cin.clear();

    list<string> lis(vec.begin(),vec.end());          ①

```

```
24      cout << "Slowo do znalezienia: ";
25      cin  >> s;
26
27      auto sit = vec.cbegin();
28      auto lit = lis.cbegin();
29
30        // wektor
31      for ( ; sit != vec.cend(); ++sit)
32          if ( *sit == s ) break;
33      if ( sit != vec.cend() )
34          cout << "(vec) Slowo " << s << " na pozycji "
35               << sit - vec.cbegin() << endl;            ②
36      else
37          cout << "Slowo " << s << " nie wystapilo" << endl;
38
39        // lista
40      for ( ; lit != lis.cend(); ++lit)
41          if ( *lit == s ) break;
42      if ( lit != lis.cend() )
43          cout << "(lis) Slowo " << s << " na pozycji "
44               << distance(lis.cbegin(),lit) << endl;    ③
45      else
46          cout << "Slowo " << s << " nie wystapilo" << endl;
47  }
```

W linii ① zastosowaliśmy konstruktor listy (taki sam istnieje i dla pozostałych kolekcji) przyjmujący dwa iteratory związane z inną kolekcją, w tym przypadku wektorem. Elementy od tego wskazywanego przez pierwszy iterator włącznie do wskazywanego przez drugi z iteratorów *wyłącznie* są kopiowane do nowo tworzonej kolekcji. Tak więc lista w naszym przypadku będzie zawierać te same elementy co wektor. Ponieważ iterator związany z wektorem należy do kategorii iteratorów o dostępie swobodnym, można (②) użyć po prostu różnicy iteratorów do określenia pozycji — tu obliczmy ją względem pozycji zerowej, zwracanej przez **begin**. Dla listy (③) użyliśmy do tego funkcji **distance**, ponieważ iterator związany z listami należy do kategorii dwukierunkowych i nie wspiera arytmetyki wskaźników. Przykładowy wydruk:

```
Podaj slowa (^D konczy):
Ala
Ela
Ula
^D
Slowo do znalezienia: Ula
(vec) Slowo Ula na pozycji 2
(lis) Slowo Ula na pozycji 2
```

Do przerwania wczytywania danych użyty tu został znak końca danych, czyli Ctrl-D (pod Windows byłoby to Ctrl-Z).

Zwykła tablica może być, przynajmniej do pewnego stopnia, traktowana jak kolekcja. Wskaźniki do elementów tablicy pełnią wtedy rolę iteratorów. Na przykład po

```
int arr[] = {1,4,6,8,9};
vector<int> v(arr+1,arr+4);
```

wektor v będzie zawierał liczby 4, 6 i 8 bo wskaźnik `arr+1` wskazuje na liczbę 4, a wskaźnik `arr+4` na liczbę 9, a zgodnie z tym, co mówiliśmy, przedział obejmuje element wskazywany przez iterator pierwszy, ale już nie obejmuje elementu wskazywanego przez iterator drugi.

Jak wspomnieliśmy, kolekcją (znaków) jest obiekt klasy **string**. Kilka przykładów ilustrujących tę cechę napisów w stylu C++ podaliśmy już w rozdz. 17.2 na stronie 366.

Inną, często stosowaną operacją jest usuwanie elementów kolekcji. Służy do tego metoda **erase**. Jej argumentem powinien być iterator wskazujący na usuwany element albo para iteratorów; w tym drugim przypadku usuwane są elementy od wskazywanego pierwszym iteratorem włącznie do wskazywanego drugim iteratorem wyłącznie. Na przykład po

```
vector<Person> os;

os.push_back(Person("Jenny"));
os.push_back(Person("Jill"));
os.push_back(Person("Jane"));
os.push_back(Person("Janet"));

os.erase(os.begin()+1, os.end());
```

z wektora os usunięte zostaną wszystkie elementy prócz pierwszego. Elementów nie należy usuwać w pętli, bo po usunięciu pierwszego z nich stan kolekcji się zmienia — pozostałe elementy zmieniają swoje pozycje, co może doprowadzić do chaosu.

Podobnie można do kolekcji dodawać nowe elementy za pomocą metody **insert**. Argumentem wskazującym, na której pozycji (a ściślej, *przed* którym elementem) nowe elementy powinny się pojawić, jest oczywiście iterator. Z kolei elementy do wstawienia mogą pochodzić z innej kolekcji; ich zakres wskazywany jest parą iteratorów:

```
double tab[] = {2, 4, 6, 7, 8, 1.5, 5, 7};
list<double> lis(5);
lis.insert(lis.begin(),10.5);
lis.insert(lis.begin(),tab+1,tab+3);
```

W tym fragmencie zwróćmy uwagę na linię drugą. Tworzymy tu listę o pięciu elementach, którym nadawane są wartości domyślne. W naszym przypadku elementami były liczby, o wartości domyślnej 0, ale dla obiektów użyty byłby konstruktor domyślny. Zatem powinien on wtedy istnieć! Następnie dodajemy liczbę 10.5 na początek listy. Pozostałe 5 elementów przesuwa się w prawo: po tej operacji lista ma już

sześć elementów. W czwartej linii na nowej początkowej pozycji (a więc przed wstawione tam przed chwilą 10.5) wstawiamy dwie liczby z tablicy tab: liczby z pozycji 1 i 2, czyli liczby 4 i 6. Zatem teraz lista ma osiem elementów:

```
4 6 10.5 0 0 0 0 0
```

Dla list (również dla kolekcji typu **deque**, ale nie **vector**), prócz metod operujących na końcu kolekcji, szczególnie efektywnie zaimplementowane są metody operujące na jej początku. Analogicznie do metod **push_back**, **pop_back** i **back**, o których już mówiliśmy, działają metody **push_front**, **pop_front** i **front**. W miarę możności należy raczej korzystać z funkcji operujących na końcach kolekcji, a nie np. z funkcji **insert** czy **erase**, których implementacja jest zwykle wolniejsza.

24.2 Algorytmy i obiekty funkcyjne

Biblioteka standardowa dostarcza ponad sto **algorytmów**, czyli szablonów funkcji operujących na kolekcjach. Argumentami tych funkcji nie są kolekcje jako takie, a iteratory.

Jako argumenty wielu algorytmów występują też funkcje albo, jeszcze częściej, obiekty funkcyjne.

24.2.1 Algorytmy

Omówimy pokrótce kilka algorytmów dostarczanych przez bibliotekę standardową. Nie będzie to przegląd pełny, ale pozwoli zorientować się Czytelnikowi, w jaki sposób algorytmy stosować. Większość algorytmów udostępniana jest poprzez włączenie pliku nagłówkowego *algorithm*.

Do bardziej popularnych i użytecznych należą algorytmy sortujące. Użycie ich jest proste:

P209: *sort.cpp* Sortowanie

```
#include <iostream>
#include <vector>
#include <list>
#include <string>
#include <algorithm>
#include <iterator>
using namespace std;

int main() {
    vector<string> vec{"Paris", "London", "Warsaw",
                       "Berlin", "Lisbon", "Oslo"};
    auto ini = vec.begin(), fin = vec.end();

    list<string> lis(vec.size()-2);                    (1)
```

```
15      copy(ini+1, fin-1, lis.begin());                          ②
16
17      sort(ini,fin);                                            ③
18      lis.sort();                                               ④
19
20      copy(ini, fin,                                            ⑤
21           ostream_iterator<string>(cout, " "));
22      cout << endl;
23
24      copy(lis.begin(),lis.end(),
25           ostream_iterator<string>(cout, " "));
26      cout << endl;
27  }
```

W programie tworzymy wektor napisów. Następnie tworzymy listę (①) o początkowej wielkości o dwa mniejszej niż rozmiar wektora. Algorytm Algorytm **copy** (②) pobiera zakres (w postaci pary iteratorów) z kolekcji źródłowej i iterator z kolekcji docelowej wskazujący, od którego miejsca elementy mają być nadpisywane. Zauważmy, że algorytm nie tworzy nowych elementów docelowego kontenera, tylko nadpisuje istniejące. Dlatego właśnie tworząc listę musieliśmy zadbać o to, by miała ona wystarczający rozmiar.

Następnie sortujemy wektor algorytmem**sort** (③) — jak zwykle, zakres elementów do posortowania przekazujemy jako parę iteratorów. Funkcja **sort** oczekuje iteratorów dostępu swobodnego. Dlatego nie można jej użyć do posortowania listy — dla list jednak istnieje *metoda* **sort** (patrz ④).

Podobny mechanizm działa również dla innych algorytmów wymagających iteratorów swobodnego dostępu — kolekcje dostarczające tylko dwukierunkowych iteratorów mają zamiast nich odpowiadające im metody (np. **merge**, **remove**, **reverse** itd.).

Zwróćmy jeszcze uwagę na linię ⑤. Tu również używamy funkcji **copy**. Kopiujemy elementy z jednej kolekcji, określone za pomocą iteratorów, do drugiej, określonej przez jeden iterator, tak jak poprzednio, przy kopiowaniu wektora do listy. Tym razem iterator będący trzecim argumentem jest nieco dziwny. Jest to iterator utworzony z szablonu **ostream_iterator** (z nagłówka ***iterator***) przez konkretyzację dla typu **string**. Argumentami konstruktora tej konkretyzacji jest strumień wyjściowy, w naszym przypadku cout, oraz napis, który będzie pełnił rolę separatora pomiędzy elementami, w naszym przykładzie znak odstępu. W ten sposób strumień wyjściowy jest traktowany jak kolekcja elementów, które mają być wypisane. Iterator związany z tą kolekcją jest typu wyjściowego (*output iterator*), a więc jest jednokierunkowy, jednoprzebiegowy (*one pass*) i nie dopuszcza odczytu elementów. Wydruk programu to:

```
Berlin Lisbon London Oslo Paris Warsaw
Berlin Lisbon London Warsaw
```

Jak widzieliśmy, funkcja **copy** kopiuje elementy z jednej kolekcji do drugiej nadpisując już *istniejące* elementy. Dlatego w programie powyżej (linia ①) musieliśmy utworzyć listę od razu odpowiedniego rozmiaru. To nie zawsze jest takie proste — często

nie wiemy z góry ile elementów trzeba będzie skopiować. W takich sytuacjach możemy użyć specjalnej funkcji **back_inserter**, która zwraca obiekt zachowujący się jak iterator wyjściowy, ale który dodaje nowe elementy do kolekcji (wywołując **push_back**). Na przykład w poniższym przykładzie kopiujemy elementy z jednego wektora do drugiego (początkowo pustego), ale tylko takie, które spełniają predykat przesłany jako ostatni argument do funkcji **copy_if**:

P210: ***backins.cpp*** Funkcja *back_inserter*

```
#include <algorithm>     // copy
#include <iostream>
#include <iterator>      // inserters
#include <vector>

int main() {
    std::vector<int> v{1, 2, 3, 4, 5, 6, 7};
    std::vector<int> emp;
    std::copy_if(v.begin(), v.end(),
                 std::back_inserter(emp),
                 [](auto n){ return n%2 != 0; });
    std::copy(emp.begin(), emp.end(),
              std::ostream_iterator<int>(std::cout, "\n"));
}
```

(istnieją też funkcje **front_inserter** i **inserter**).

Przekazywanie funkcji do algorytmów jest bardzo częste. Na przykład funkcje sortujące mogą mieć dodatkowy argument określający jak elementy mają być porównywane (domyślnie są porównywane za pomocą operatora <). Powinien to być wskaźnik do funkcji, lambda lub obiekt funkcyjny (cokolwiek, co zachowuje się jak funkcja) pełniącej rolę **komparatora**, czyli funkcji pobierającej dwa argumenty typu takiego, jakiego są elementy kolekcji i zwracającej wartość logiczną (**bool**) odpowiadającą na pytanie *czy pierwszy argument jest* ściśle *mniejszy od drugiego?*
Na przykład w poniższym programie funkcja **compar** (①) uznaje każdą liczbę nieparzystą za mniejszą od dowolnej liczby parzystej a liczby o tej samej parzystości porządkuje w sposób naturalny (rosnąco). Ten komparator użyty jest w linii ④ jako wskaźnik funkcyjny.
Definiujemy też prostą strukturę **Evens** (②) z przeciążonym operatorem wywołania: obiekty tej klasy mogą również służyć jako komparatory (⑤) — w tym przypadku liczby parzyste są uważane za mniejsze od nieparzystych, a dla liczb o tej samej parzystości porządek będzie odwrócony (malejący).
W końcu w linii ⑥ jako komparatora użyliśmy lambdy (porządkującej liczby malejąco, bez względu na parzystość).
Funkcja (szablon) **printVec** (③) służy do wypisywania wektorów.

P211: ***sortev.cpp*** Komparatory

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

bool compar(int a, int b) {                                    ①
    return (a+b)%2 != 0 ? a%2 != 0 : a < b;
}

struct Evens {                                                 ②
    bool operator()(int a, int b) {
        return (a+b)%2 != 0 ? a%2 == 0 : b < a;
    }
};

template <typename T>
void printVec(const std::vector<T>& vec) {                     ③
    copy(vec.cbegin(), vec.cend(),
        std::ostream_iterator<T>(std::cout, " "));
    std::cout << '\n';
}

int main() {
    std::vector<int> vec{2, 5, 2, 9, 1, 5, 7, 4};
    printVec(vec);

    sort(vec.begin(), vec.end(), compar);                      ④
    printVec(vec);

    sort(vec.begin(), vec.end(), Evens{});                     ⑤
    printVec(vec);

    sort(vec.begin(), vec.end(),
         [](int a, int b) {return b < a;});                    ⑥
    printVec(vec);

}
```

Program drukuje

```
2 5 2 9 1 5 7 4
1 5 5 7 9 2 2 4
4 2 2 9 7 5 5 1
9 7 5 5 4 2 2 1
```

Ogólnie, funkcje zwracające wartość logiczną nazywamy **predykatami**. Są one

używane przez bardzo wiele algorytmów, nie tylko sortujących. Na przykład funkcja (a właściwie, jak pamiętamy, wzorzec funkcji) **count_if** zlicza liczbę tych elementów kolekcji, określonej za pomocą dwóch iteratorów, dla których predykat wywołany z elementem kolekcji jako jedynym argumentem zwraca **true**. W poniższym programie predykat **parzysta** został użyty (linia 21) do zliczenia liczby elementów parzystych wektora:

P212: *predyk.cpp* Predykaty

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  #include <iterator>
5  using namespace std;
6
7  bool parzysta(int a) {
8      return (a&1) == 0;
9  }
10
11 int main() {
12     vector<int> vec;
13     int d;
14     while ( cin >> d ) vec.push_back(d);
15     cin.clear();
16
17     cout << "W ciagu liczb ";
18     copy(vec.begin(),vec.end(),
19          ostream_iterator<int>(cout, " "));
20     cout << "\njest "
21          << count_if(vec.begin(), vec.end(), parzysta)
22          << " liczb parzystych\n";
23 }
```

Wydruk tego programu:

```
2 4 -2 7 5 4 31 4 5 -4
^D
W ciagu liczb 2 4 -2 7 5 4 31 4 5 -4
jest 6 liczb parzystych
```

Bardzo obszerna jest klasa algorytmów wyszukujących. Podstawowe funkcje (szablony) to **find** i **find_if**. Funkcja **find** poszukuje w kolekcji pierwszego elementu równego obiektowi przesłanemu jako argument, natomiast **find_if** poszukuje pierwszego elementu, dla którego spełniony jest pewien predykat, to znaczy pierwszego elementu kolekcji dla którego funkcja definiująca ten predykat zwróci **true**. Kolekcja, jak zwykle, zadana jest parą iteratorów. W razie sukcesu obie funkcje zwracają iterator do

znalezionego elementu. Jeśli wyszukiwanie zakończyło się porażką, zwracany jest iterator **end()**. Przykład na obie te funkcje znajdziemy w następującym programie:

P213: *szuk.cpp* Wyszukiwanie

```
#include <vector>
#include <iostream>
#include <string>
#include <cctype>    // tolower
#include <algorithm>
#include <iterator>
using namespace std;

bool czy_na_A(string& s) {
    return s[0] == 'A';
}

void mala(string& imie) {
    imie[0] = tolower(imie[0]);
}

int main() {
    vector<string> vs;

    vs.push_back("Magda");  vs.push_back("Anna");
    vs.push_back("Monika"); vs.push_back("Agata");
    vs.push_back("Ala");    vs.push_back("Urszula");

    vector<string>::iterator k;

    k = find(vs.begin(), vs.end(), "Anna");
    if ( k != vs.end() )
        cout << *k << " znaleziona\n";
    else
        cout << "Anna nie znaleziona\n";

    k = find(vs.begin(), vs.end(), "Basia");
    if ( k != vs.end() )
        cout << *k << " znaleziona\n";
    else
        cout << "Basia nie znaleziona\n";


    cout << "\nZ imion\n";
    copy(vs.begin(),vs.end(),
         ostream_iterator<string>(cout," "));
    cout << "\nnastepujace zaczynaja sie na \'A\':\n";
```

```
43      k = vs.begin();
44      while ( k < vs.end() ) {
45          k = find_if(k, vs.end(), czy_na_A);
46          if ( k != vs.end() ) cout << *k++ << " ";
47      }
48      cout << endl;
49
50      for_each(vs.begin(),vs.end(),mala);
51      cout << "\nPo zamianie na male:\n";
52      copy(vs.begin(),vs.end(),
53           ostream_iterator<string>(cout," "));
54      cout << endl;
55  }
```

W liniach 26 i 32 użyta jest funkcja **find**, po czym sprawdzane jest, czy otrzymany iterator nie jest równy vs.end(), co oznaczałoby porażkę wyszukiwania.

Następnie z wektora imion wyszukiwane są w pętli (linie 44-47) imiona zaczynające się na literę 'A'. W tym celu korzystamy z funkcji **find_if** i predykatu **czy_na_A** (zdefiniowanego w liniach 9-11).

W linii 50 korzystamy z funkcji **for_each**. Jest to przykład **algorytmu modyfikującego** kolekcję, w odróżnieniu od algorytmów wyszukujących, które są **algorytmami niemodyfikującymi**. Działanie jego polega na wywołaniu dla każdego elementu kolekcji ze wskazanego przedziału pewnej określonej przez użytkownika funkcji (lub obiektu funkcyjnego). Argument przekazywany jest przez referencję, tak, aby możliwa była modyfikacja jego oryginału. Wartość zwracana funkcji, jeśli istnieje, jest ignorowana. W naszym przypadku funkcją tą jest funkcja **mala**, która w przekazanym przez referencję napisie zmienia pierwszą literę na małą. Wydruk programu jest następujący:

```
Anna znaleziona
Basia nie znaleziona

Z imion
Magda Anna Monika Agata Ala Urszula
nastepujace zaczynaja sie na 'A':
Anna Agata Ala

Po zamianie na male:
magda anna monika agata ala urszula
```

24.2.2 Obiekty funkcyjne

Jak widzieliśmy, ważną rolę odgrywają w algorytmach funkcje. W rzeczywistości nie muszą to być funkcje: wystarczy, że jest to „coś, co da się wywołać". Jak pamiętamy z rozdziału o przeciążeniu operatora wywołania (rozdz. 19.4.3, str. 417), jeśli w klasie przeciążony został operator wywołania, to nazwa obiektu tej klasy z podanymi

argumentami w nawiasach powoduje wywołanie metody **operator()**. Obiekt staje się zatem **obiektem wywoływalnym** (ang. *callable object*). Takie właśnie obiekty mogą pełnić rolę **obiektów funkcyjnych** — można je wstawić tam, gdzie może wystąpić funkcja. Obiekty funkcyjne mają z punktu widzenia algorytmów tę przewagę nad funkcjami, że tworząc je można, na przykład poprzez konstruktor, przekazać do nich dodatkową informację prócz samych tylko argumentów wywołania, których liczba i typ są określone przez algorytm. Obiekty funkcyjne zapewniają zatem większą elastyczność algorytmów. Są też często efektywniejsze od zwykłych funkcji, bo metoda **operator()** może być rozwinięta, podczas gdy przekazywanie zwykłej funkcji przez wskaźnik nie pozwala kompilatorowi na jej rozwinięcie.

Przyjrzyjmy się następującemu programowi:

P214: *compr.cpp* Obiekty funkcyjne

```
#include <iostream>
#include <cmath>      // sqrt
#include <vector>
#include <algorithm>  // sort, copy
#include <iterator>
using namespace std;

struct Komp {
    enum Sposoby {
        wg_sumy_cyfr,
        wg_ilosci_dzielnikow,
        wg_wartosci,
        wg_wart_odwrotnie
    };

    Komp(Sposoby sposob): sposob(sposob) { }

    bool operator()(int n1, int n2);

    class BrakKomparatora { };

private:
    Sposoby sposob;

    static int suma_cyfr(int n);
    static int ilosc_dzi(int n);
};

bool Komp::operator()(int n1, int n2) {
    switch (sposob) {
        case wg_sumy_cyfr:
            return suma_cyfr(n1) < suma_cyfr(n2);
        case wg_ilosci_dzielnikow:
```

```
34              return ilosc_dzi(n1) < ilosc_dzi(n2);
35          case wg_wartosci:
36              return n1 < n2;
37          case wg_wart_odwrotnie:
38              return n2 < n1;
39          default:
40              throw BrakKomparatora();
41                          // nigdy się nie zdarzy
42      }
43  }
44
45  int Komp::suma_cyfr(int n) {
46      // suma cyfr liczby całkowitej n (licząc
47      // wszystkie  cyfry ze znakiem  dodatnim)
48      n = n >= 0 ? n : -n;
49      int s = 0;
50      while (n) { s += n%10; n /= 10; }
51      return s;
52  }
53
54  int Komp::ilosc_dzi(int n) {
55      // ilość dodatnich dzielnikow liczby całkowitej
56      // n (wliczając jedynkę i samą liczbę n)
57      n = n > 0 ? n : -n;
58      if ( n < 3 ) return n;
59      int sr = (int) sqrt(n+0.5);
60      int ilosc = (sr*sr == n ? 1 : 2);
61      for (int i = 2; i <= sr; ++i) if (n%i == 0) ilosc += 2;
62      return ilosc;
63  }
64
65  int main() {
66      int tab[] = {7, 4, 8, 12, 13, 119, 16, 6};
67      vector<int> v(tab,tab+sizeof(tab)/sizeof(int));
68      v.push_back(64);
69
70       // sortujemy różnymi sposobami i wyświetlamy wyniki
71
72      cout <<   "Sortujemy wg. sumy cyfr\n";
73      sort(v.begin(),v.end(),Komp(Komp::wg_sumy_cyfr));
74      copy(v.begin(),v.end(),ostream_iterator<int>(cout," "));
75
76      cout << "\nSortujemy wg. ilosci dzielnikow\n";
77      sort(v.begin(),v.end(),Komp(Komp::wg_ilosci_dzielnikow));
78      copy(v.begin(),v.end(),ostream_iterator<int>(cout," "));
79
```

```
80
81      cout << "\nSortujemy wg. wartosci\n";
82      sort(v.begin(),v.end(),Komp(Komp::wg_wartosci));
83      copy(v.begin(),v.end(),ostream_iterator<int>(cout," "));
84
85      cout << "\nSortujemy wg. wartosci odwrotnie\n";
86      sort(v.begin(),v.end(),Komp(Komp::wg_wart_odwrotnie));
87      copy(v.begin(),v.end(),ostream_iterator<int>(cout," "));
88
89      cout << endl;
90  }
```

Program sortuje wektor na wiele różnych sposobów. Komparator jest tym razem nie funkcją, ale obiektem klasy **Komp**. W klasie tej przeciążony jest operator **operator()** (linie 18 i 29-43). Jedyny konstruktor klasy wymaga argumentu typu wyliczeniowego, który określa następnie sposób sortowania. Odpowiedni element wyliczenia jest zapamiętywany w składowej sposob obiektu.

W funkcji **main** wywołujemy **sort** i jako wartość trzeciego argumentu podajemy utworzony „na miejscu" anonimowy obiekt klasy **Komp**. W samym obiekcie zapamiętane jest zatem kryterium sortowania, które określiliśmy argumentem przesłanym do konstruktora. W trakcie sortowania obiekt będzie wielokrotnie wywoływany przez algorytm z elementami kolekcji jako argumentami: dzięki temu jednak, że jest to obiekt, może przechowywać dodatkową informację w swoich składowych. W naszym przypadku jest to składowa sposob, dzięki której przy każdym wywołaniu wybrana zostaje odpowiednia gałąź instrukcji **switch** (linia 30). Wydruk z programu:

```
Sortujemy wg. sumy cyfr
12 4 13 6 7 16 8 64 119
Sortujemy wg. ilosci dzielnikow
13 7 4 6 8 119 16 12 64
Sortujemy wg. wartosci
4 6 7 8 12 13 16 64 119
Sortujemy wg. wartosci odwrotnie
119 64 16 13 12 8 7 6 4
```

Obiekty funkcyjne stosuje się również do tworzenia manipulatorów. W rozdz. 16.3.2 (str. 333) mówiliśmy o manipulatorach z argumentami: jest szereg takich manipulatorów zdefiniowanych w bibliotece dołączanej poprzez nagłówek ***iomanip***, ale jak tworzyć własne takie manipulatory? Przyjrzyjmy się przykładowi:

P215: ***maniparg.cpp*** Manipulatory jako obiekty funkcyjne

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
```

```
5  struct maniparg {
6      string str;
7      maniparg(int ile, char c) : str(ile,c) { }
8      ostream& operator()(ostream& s) const {
9          return s << str;
10     }
11 };
12
13 ostream& operator<<(ostream& s, const maniparg& manip) {
14     return manip(s);
15 }
16
17 int main() {
18     cout << maniparg(7,'*') << "To jest maniparg"
19          << maniparg(3,'!') << maniparg(7,'*') << endl;
20
21     maniparg trzywykrzykniki(3,'!');
22     maniparg siedemgwiazdek (7,'*');
23
24     cout << siedemgwiazdek  <<  "To jest maniparg"
25          << trzywykrzykniki << siedemgwiazdek  << endl;
26 }
```

Tworzymy klasę **maniparg** i dla niej przeciążamy w zwykły sposób operator wstawiania do strumienia '<<' (linie 13-15). Funkcja przeciążająca zadziała, według normalnych zasad, gdy po prawej stronie operatora '<<' pojawi się konkretny obiekt klasy **maniparg**. Przesłanymi argumentami będą strumień i sam obiekt. Z kolei funkcja przeciążająca operator '<<' wywołuje przesłany obiekt (linia 14), czyli funkcję **operator()** z klasy **maniparg** (zdefiniowaną w liniach 8-10). Funkcja ta została skonstruowana w ten sposób, że otrzymuje poprzez argument obiekt strumienia i wypisuje do niego napis skonstruowany z ile znaków c (przesłanych do konstruktora obiektu). Zwraca referencję do tegoż strumienia, tak że i funkcja przeciążająca operator '<<' również zwraca ten strumień, tak jak to jest wymagane. Po co ta konstrukcja?

Otóż tworząc obiekt klasy **maniparg** możemy przekazać do niego poprzez konstruktor i zapamiętać w jego składowych dowolną ilość informacji. W naszym przypadku przekazujemy tam znak i liczbę całkowitą określającą, ile razy ten znak ma być wstawiony do strumienia. Na podstawie tej informacji tworzona jest składowa str, czyli napis zawierający znak powtórzony odpowiednią liczbę razy. Z tej składowej korzysta metoda **operator()** wywoływana z funkcji przeciążającej operator '<<'. Widzimy to w linii 18, w której tworzymy obiekt klasy **maniparg** przekazując poprzez argumenty konstruktora znak gwiazdki i liczbę 7. Ponieważ obiekt klasy **maniparg** pojawił się po prawej stronie operatora '<<', wywoływana jest funkcja zdefiniowana w liniach 13-15 i przekazywany jest do niej ten obiekt. Jest on wywoływany, przekazując z kolei strumień poprzez argument. Metoda **operator()** ma zatem komplet informacji: strumień i dane ze składowej obiektu, które do tego strumienia mają być wpisane. Program drukuje

```
*******To jest maniparg!!!*******
*******To jest maniparg!!!*******
```

Zauważmy, że równie dobrze możemy utworzyć zawczasu obiekty klasy **maniparg** nadając im nazwy, a potem używać ich wielokrotnie, jak robimy to z obiektami trzywykrzykniki i siedemgwiazdek w liniach 24-25.

24.3 Przykłady

Na zakończenie tego krótkiego wstępu do biblioteki standardowej przeanalizujmy jeszcze kilka przykładów ilustrujących jej zastosowania.

Ważną klasą, a właściwie szablonem klas, jest **pair** z nagłówka ***utility***. Szablon ten opisuje proste, ale często bardzo przydatne struktury składające się z pary składowych pewnych typów, które mogą być różne; mogą to być zarówno typy wbudowane jak i definiowane w programie. Tak więc, jeśli potrzebne nam są często pary (**string**, **int**), na przykład do opisu imienia i wieku osób, to właściwym typem byłaby konkretyzacja **pair<string,int>**. Struktura ta zawiera dwie składowe o nazwach first — w tym przypadku typu **string** — oraz second, w naszym przypadku typu **int**. Klasa jest tak zaimplementowana, o co już sami nie musimy się martwić, że prawidłowo działają operatory porównywania ('==' i '!='), przypisania itd. — jeśli oczywiście jest tak dla typów składowych.

Taką właśnie klasę tworzymy w poniższym programie. Dane o osobach czytane są z pliku tekstowego ***pary.dat*** o następującej zawartości:

```
Ania  17 Zosia  7 Ala 19
Ula   36 Asia  18
Ola    4
```

Przy wczytywaniu (linia 46) dane są bezpośrednio kopiowane do wektora — obiektu klasy konkretyzującej wzorzec **vector**, której elementami są pary typu **pair<string,int>**. Na wektorze tym dokonujemy następnie kilku operacji ilustrujących algorytmy z biblioteki standardowej i zastosowanie obiektów funkcyjnych:

P216: ***pary.cpp*** Pary

```
1  #include <fstream>
2  #include <string>
3  #include <utility>   // pair
4  #include <vector>
5  #include <algorithm> // copy, sort, itd.
6  #include <iostream>
7  using namespace std;
8
9  typedef pair<string,int> PARA;
10 typedef vector<PARA>     VECT;
11 typedef VECT::iterator   VECTIT;
```

```
12
13 template <typename P>
14 class niepelnoletni {
15     int wiek;
16 public:
17     niepelnoletni(int wiek) : wiek(wiek) { }
18
19     bool operator()(const P& p) const {
20         return p.second < wiek;
21     }
22 };
23
24 template <typename P1, typename P2>
25 ostream& operator<<(ostream& str, const pair<P1,P2>& p) {
26     return str << "[" << p.first << "," << p.second << "]";
27 }
28
29 template <typename P1, typename P2>
30 istream& operator>>(istream& str, pair<P1,P2>& p) {
31     return str >> p.first >> p.second;
32 }
33
34 template <typename P>
35 bool komp(const P& p1, const P& p2) {
36     return p1.second < p2.second;
37 }
38
39 int main() {
40     ifstream plik("pary.dat");
41
42     PARA   p;
43     VECT vec;
44     VECTIT it, fin;
45
46     while (plik >> p) vec.push_back(p);
47
48     cout << "Po wczytaniu:\n";
49     fin=vec.end();
50     for (it = vec.begin(); it != fin; ++it)
51         cout << *it << " ";
52
53     cout << "\nNajstarsza "
54          << *max_element(vec.begin(),vec.end(),komp<PARA>)
55          << ", najmlodsza "
56          << *min_element(vec.begin(),vec.end(),komp<PARA>);
57
```

```
58      it = remove_if(vec.begin(),fin,
59                     niepelnoletni<PARA>(18));
60      vec.erase(it,vec.end());
61
62      cout << "\nPo usunieciu niepelnoletnich:\n";
63      fin=vec.end();
64      for (it = vec.begin(); it != fin; ++it)
65          cout << *it << " ";
66
67      sort(vec.begin(),fin,komp<PARA>);
68
69      cout << "\nPo uporzadkowaniu:\n";
70      fin=vec.end();
71      for (it = vec.begin(); it != fin; ++it)
72          cout << *it << " ";
73
74      cout << endl;
75  }
```

W liniach 9-11 definiujemy aliasy nazw typów, których będziemy używać w dalszym ciągu. Nazwa **PARA** jest synonimem nazwy typu powstającego z konkretyzacji szablonu **pair**, w którym typem składowej first jest **string**, a typem składowej second jest **int**. Zatem „prawdziwą" nazwą tego typu jest **pair<string,int>**. Z kolei **VECT** (linia 10) jest synonimem nazwy klasy powstającej z konkretyzacji wzorca **vector**, w której typem elementów wektora jest typ **pair<string,int>** czyli **PARA**. Bez użycia **typedef** musielibyśmy jako nazwy tego typu wektora używać nieporęcznego **vector<pair<string,int> >**. Zwróćmy uwagę, że odstęp pomiędzy znakami '>' byłby tu konieczny: bez niego parser zinterpretowałby dwuznak '>>' jako operator wyjmowania ze strumienia.

W linii 43 tworzymy obiekt vec typu **VECT**. Obiekt ten jest, na razie pustym, wektorem elementów typu **PARA**. W linii 40 otwieramy strumień wejściowy plik związany z plikiem dyskowym ***pary.dat*** (patrz rozdz. 16.5, str. 342). W jaki sposób zachodzi czytanie z pliku obiektów typu **PARA**? Jest to typ przez nas zdefiniowany, a więc operator '>>' nie wie jak takie obiekty wyjąć ze strumienia. Dlatego musieliśmy dla tego typu przeciążyć operator '>>'. Zrobiliśmy to w liniach 29-32. Zdefiniowany tam szablon zadziała dla par (obiektów klas konkretyzujących wzorzec **pair**) dowolnych typów, pod warunkiem, że dla typów składowych first i second działanie '>>' jest dobrze określone. W naszym przypadku typami składowych są **string** i **int**, więc nie będzie żadnych kłopotów.

W liniach 49-51 drukujemy zawartość kolekcji vec. Drukowanymi elementami kolekcji są obiekty typu **PARA** przez nas zdefiniowanego, więc musieliśmy dla niego przeciążyć operator '<<'. Zrobiliśmy to w liniach 24-27, analogicznie do jak dla operatora '>>'.

Linia 54

```
*max_element(vec.begin(),vec.end(),komp<PARA>)
```

demonstruje użycie funkcji (algorytmu) **max_element** znajdującej największy element kolekcji. Kolekcja jest jak zwykle określona za pomocą iteratorów. Gdybyśmy nie podali trzeciego argumentu, to do porównań elementów zostałby użyty operator '<'. Elementami kolekcji są obiekty typu **PARA**, więc aby to zadziałało, w klasie **PARA** operator '<' musiałby być przeciążony. To mogliśmy zrobić, ale w tym przypadku zastosowane zostało inne rozwiązanie. Jako trzeci argument podaliśmy funkcję porównującą obiekty (predykat w postaci szablonu — linie 34-37). Zauważmy, że jako parametru wzorca użyliśmy tu tylko jednego typu (a nie pary typów). Wzorzec zadziała więc dla każdego typu, w którym istnieją i dają się porównywać za pomocą '<' składowe o nazwie second. Oczywiście typ **PARA** spełnia te warunki. Jak widać z definicji, komparator będzie porównywał obiekty typu **PARA** według wieku, będącego w nich składową second. Funkcja **max_element** zwraca iterator wskazujący największy obiekt kolekcji.

Na podobnej zasadzie działa funkcja **min_element** użyta w linii 56, która, jak łatwo się domyślić, znajduje najmniejszy element kolekcji. Prawidłowość działania obu funkcji widzimy z wydruku

```
Po wczytaniu:
[Ania,17] [Zosia,7] [Ala,19] [Ula,36] [Asia,18] [Ola,4]
Najstarsza [Ula,36], najmlodsza [Ola,4]
Po usunieciu niepelnoletnich:
[Ala,19] [Ula,36] [Asia,18]
Po uporzadkowaniu:
[Asia,18] [Ala,19] [Ula,36]
```

Często się zdarza, że z kolekcji chcemy usunąć elementy spełniające pewien warunek. Można to wygodnie zrobić za pomocą algorytmu **remove_if** użytego w linii 58. Jak zwykle ciąg elementów kolekcji zadany jest przez iteratory będące pierwszymi dwoma argumentami wywołania. Jako trzeci argument podajemy funkcję (wskaźnik do funkcji) albo obiekt funkcyjny, który wywołany z elementem kolekcji jako argumentem zwraca wartość logiczną **true**, jeśli element ten ma zostać usunięty. W naszym przypadku jest to obiekt klasy konkretyzującej szablon **niepelnoletni** dla typu **PARA**. Tworząc w linii 59 obiekt tej klasy posyłamy do konstruktora liczbę, która w obiekcie zapamiętana zostanie jako składowa wiek (patrz definicja szablonu w liniach 13-22). Musimy też przeciążyć w klasie operator wywołania (linie 19-21) tak, aby zwracał **true** jeśli warunek jest spełniony, czyli gdy wiek osoby jest mniejszy od tego zapamiętanego w składowej wiek obiektu.

Funkcja **remove_if**, wbrew swej nazwie, *nie* usuwa niechcianych elementów. Przestawia je tylko tak, aby znalazły się na końcu kolekcji. Tak więc po wywołaniu funkcji liczba elementów kolekcji nie zmienia się. Zmienia się tylko ich kolejność: elementy, dla których warunek nie był prawdziwy znajdują się na początku, a te dla których warunek był spełniony na końcu kolekcji. Funkcja zwraca iterator do pierwszego elementu z tych „niechcianych", dlatego typem zmiennej it z linii 58 jest **VECTIT** czyli **VECT::iterator**. Gdyby takich elementów w ogóle nie było, funkcja zwróciłaby iterator vec.end(). Fizycznego usunięcia elementów, a więc skrócenia kolekcji, dokonujemy za pomocą wywołania funkcji składowej o nazwie **erase** na rzecz kolekcji (linia 60). Pobiera ona jako argumenty iteratory określające podciąg kolekcji przeznaczony do

usunięcia — w naszym przykładzie jest to podciąg od elementu wskazywanego przez it do końca.

W linii 67 sortujemy pozostałe w wektorze elementy za pomocą wywołania znanego nam już algorytmu **sort**. Do porównywania elementów używamy takiego samego komparatora jak ten, który został zastosowany w funkcjach **max_element** i **min_element**

Jako drugi przykład rozpatrzmy program ilustrujący użycie map (słowników). Wzorzec klas reprezentujących słownki nazywa się **map** (z nagłówka ***map***). Aby skonkretyzować ten szablon należy podać dwa typy, **Typ1** i **Typ2**,

```
#include <map>
// ...
map<Typ1,Typ2> mapa;
```

które określają typy tzw. kluczy i wartości. Poszczególne elementy mapy będą parami typu **pair<const Typ1,Typ2>**. Zwróćmy uwagę, że typ kluczy jest modyfikowany tak, aby odpowiadać typowi *ustalonemu* (**const**), a więc klucze są niemodyfikowalne, wartości związane z kluczem mogą natomiast być modyfikowalne. W mapie nie mogą wystąpić dwa elementy z takim samym kluczem. Najbliższym „kuzynem" mapy w Javie jest klasa **TreeMap**. Ważną różnicą jest jednak to, że w mapach w C++ zarówno kluczami jak i wartościami mogą być obiekty typów wbudowanych, jak **int** czy **double**. Pamiętać tylko trzeba, że dla typu kluczy musi być zdefiniowane porównywanie za pomocą operatora '<' (bo słowniki są w C++ implementowane jako drzewa czerwono-czarne, a nie „prawdziwe" mapy haszowane). Tak oczywiście jest dla typów liczbowych czy dla typu **string**; jeśli klucze są typu przez nas zdefiniowanego, to musimy poprzez mechanizm przeciążania działanie operatora '<' zdefiniować (odpowiedni komparator można też przesłać jako trzeci typ dla szablonu **map** podczas jego konkretyzacji).

Każdy element mapy, będąc obiektem typu **pair**, ma składowe first i second odpowiadające odpowiednio kluczowi i związanej z tym kluczem wartości. Operator indeksowania ('[]') jest w mapach przeciążony w ten sposób, że wyrażenie 'mapa[key]' jest referencją do wartości związanej z kluczem key. Jeśli w mapie taki klucz nie występuje, to błędu nie będzie: nowy element mapy zostanie automatycznie utworzony, a jako wartość wstawiona zostanie wartość domyślna odpowiedniego typu (wartość zerowa dla typów liczbowych). Na przykład we fragmencie kodu

```
1    map<string,int> slownik;
2    slownik["Jan"] = 20;
3    slownik["Ira"] = ++slownik["Ula"] + 18;
```

zostanie w linii 2 dodany do początkowo pustej mapy slownik element o kluczu "Jan" i wartości 20. W linii 3 najpierw tworzony jest element ["Ula",0], a następnie jego wartość (czyli zero) jest inkrementowana. Zatem po opracowaniu wyrażenia '++slownik["Ula"]' element z kluczem "Ula" ma wartość 1. Wartością całej prawej strony przypisania jest zaś 19.

Opracowanie lewej strony przypisania z linii 3 spowoduje utworzenie elementu ["Ira", 0] i następnie przypisanie do wartości skojarzonej z kluczem "Ira" wartości

prawej strony przypisania, czyli liczby 19. Po wykonaniu tego fragmentu elementami słownika są zatem pary ["Jan",20], ["Ula",1] i ["Ira",19].

Rozpatrzmy jako przykład następujący program:

P217: ***mapy.cpp*** Mapy (słowniki)

```
#include <iostream>
#include <iomanip>   // left, setw
#include <string>
#include <map>
#include <utility>   // pair
#include <algorithm>
using namespace std;

using Emp = pair<string,int>;
using MAP = map<string,Emp>;

class Zakr {
    int min,max;
public:
    Zakr(int min,int max) : min(min), max(max) {}

    bool operator()(const pair<const string,Emp>& p) const {
        int zarobki = p.second.second;
        return  (min < zarobki) && (zarobki < max);
    }
};

void druk(const MAP& m) {
    for (auto [key, emp] : m) {
        auto [name, zar] = emp;
        cout << "Klucz: "   << left << setw(7) << key
             << "Imie: "    << setw(10) << name
             << "Zarobki: " << zar << endl;
    }
}

int main() {
    MAP emp;

    emp["jan"]    = Emp("Jan K.",    1900);
    emp["piotr"]  = Emp("Piotr M.",  2100);
    emp["ola"]    = Emp("Ola S.",    3100);
    emp["prezes"] = Emp("Prezes",    9900);
    emp["adam"]   = Emp("Adam A.",   1600);
    emp["emilia"] = Emp("Emilia P.", 2600);
```

```
41
42      druk(emp);
43
44      int mn = 1100, mx = 2000;
45      int ile = count_if(emp.begin(),emp.end(), Zakr(mn,mx));
46
47      cout << ile << " osob ma zarobki w zakresie od "
48           << mn << " do " << mx << endl;
49  }
```

W liniach 9 i 10 definiujemy alias **Emp** dla typu **pair<string,int>** oraz alias **MAP** dla typu słownikowego **map<string,Emp>**. Będzie to słownik złożony z elementów (par) w których klucz jest typu **const string**, a wartość jest parą złożoną z napisu i liczby. W linii 33 tworzymy pusty obiekt (słownik) typu **MAP**, a w liniach 35-40 dodajemy do tego slownika kilka elementów. Na przykład w linii 35 tworzony jest nowy element słownika o kluczu "jan" i wartości będącej obiektem typu **Emp** (a więc parą typu **pair<string,int>**) ze składową first "Jan K." i składową second równą 1900.

Tak utworzony słownik przesyłamy do funkcji **druk**, która drukuje kolejne elementy słownika. Moglibyśmy użyć tu „normalnej" iteracji z iteratorem, powiedzmy, it wskazującym na pojedyncze elementy słownika, czyli obiekty typu **pair<const string, pair<string,int> >**. Zatem '`it->first`' byłoby kluczem (np. "jan"), a '`it->second`' parą złożona z napisu i liczby. Jeśli więc chcielibyśmy wypisać zarobki osoby odpowiadającej elementowi słownika, musielibyśmy użyć składni '`it->second.second`'. Pierwszy wybór składowej odbywa się przez operator „strzałki", bo iterator ma semantykę wskaźnika, ale drugi wybór już nie: wyrażenie '`it->second`' jest referencją do obiektu typu **Emp**, a nie wskaźnikiem, więc używamy tu kropki.

Nie musimy jednak tak robić: możemy „odpakować" pary używając składni pokazanej w liniach 24 i 25. W nawiasach kwadratowych nadajemy nazwy elementom pary, specyfikując ich typy jako **auto**, bo kompilator je zna. W linii 24 odpakowujemy elementy mapy, więc key odpowiada kluczom a emp wartościom. Te wartości są obiektami **pair**, więc rozpakowujemy je znowu aby dostać imię i zarobki (linia 25).

Wydruk programu mógłby wyglądać tak:

```
Klucz: adam   Imie: Adam A.   Zarobki: 1600
Klucz: emilia Imie: Emilia P. Zarobki: 2600
Klucz: jan    Imie: Jan K.    Zarobki: 1900
Klucz: ola    Imie: Ola S.    Zarobki: 3100
Klucz: piotr  Imie: Piotr M.  Zarobki: 2100
Klucz: prezes Imie: Prezes    Zarobki: 9900
2 osob ma zarobki w zakresie od 0 do 2000
```

W linii 45 tworzymy wywoływalny obiekt typu **Zakr** przekazując do konstruktora dwie liczby określające pewien zakres. Jest to obiekt funkcyjny który może być wywołany z elementem słownika jako argumentem, a zatem obiekt ten może pełnić rolę predykatu w wywołaniu funcji **count_if** z linii 45:

```
count_if(emp.begin(),emp.end(),Zakr(mn,mx));
```

Algorytm **count_if** zlicza liczbę tych elementów kolekcji określonej dwoma iteratorami, dla których predykat będący trzecim argumentem wywołania zwraca **true**. W naszym przypadku, dla każdego elementu kolekcji (słownika) emp wywołana będzie metoda **operator()**, która zwróci **true**, jeśli dla danego elementu słownika składowa określająca zarobki mieści się w zakresie definiowanym przez składowe min i max obiektu funkcyjnego Zakr(mn,mx).

Zauważmy, że typ **MAP** zdefiniowany został jako **map<string,Emp>**, a to oznacza, jak już podkreślaliśmy, że typem elementów tej kolekcji *nie* będzie wcale typ **pair<string,Emp>**, tylko typ **pair<const string,Emp>**. Dlatego właśnie taki jest typ parametru metody **operator()**.

24.4 Lista algorytmów

Biblioteka standardowa dostarcza oczywiście dużo więcej narzędzi niż te, o których tu wspomnieliśmy. Pełny ich opis można znaleźć w książkach cytowanych w rozdz. 1.2, str. 4.

Poniżej podamy tylko nazwy i skrócony opis algorytmów z biblioteki standardowej, bez szczegółów; powinno to ułatwić poszukiwanie właściwego dla problemu nad którym pracujemy (na przykład ze strony en.cppreference.com[1]):

***** Non-modifying sequence operations** — header ***algorithm***

all_of, **any_of**, **none_of** (C++11) checks if a predicate is true for all, any or none of the elements in a range

for_each applies a function to a range of elements

for_each_n (C++17) applies a function object to the first n elements of a sequence

countcount_if returns the number of elements satisfying specific criteria

mismatch finds the first position where two ranges differ

find, **find_if**, **find_if_not** (C++11) finds the first element satisfying specific criteria

find_end finds the last sequence of elements in a certain range

find_first_of searches for any one of a set of elements

adjacent_find finds the first two adjacent items that are equal (or satisfy a given predicate)

search searches for a range of elements

search_n searches a range for a number of consecutive copies of an element

***** Modifying sequence operations** — header ***algorithm***

copy, **copy_if** (C++11) copies a range of elements to a new location

copy_n (C++11) copies a number of elements to a new location

copy_backward copies a range of elements in backwards order

[1] https://en.cppreference.com

move (C++11) moves a range of elements to a new location

move_backward (C++11) moves a range of elements to a new location in backwards order

fill copy-assigns the given value to every element in a range

fill_n copy-assigns the given value to N elements in a range

transform applies a function to a range of elements

generate assigns the results of successive function calls to every element in a range

generate_n assigns the results of successive function calls to N elements in a range

remove, **remove_if** removes elements satisfying specific criteria

remove_copy, **remove_copy_if** copies a range of elements omitting those that satisfy specific criteria

replace, **replace_if** replaces all values satisfying specific criteria with another value

replace_copy, **replace_copy_if** copies a range, replacing elements satisfying specific criteria with another value

swap swaps the values of two objects

swap_ranges swaps two ranges of elements

iter_swap swaps the elements pointed to by two iterators

reverse reverses the order of elements in a range

reverse_copy creates a copy of a range that is reversed

rotate rotates the order of elements in a range

rotate_copy copies and rotate a range of elements

shift_left, **shift_right** (C++20) shifts elements in a range

shuffle (C++11) randomly re-orders elements in a range

sample (C++17) selects n random elements from a sequence

unique removes consecutive duplicate elements in a range

unique_copy creates a copy of some range of elements that contains no consecutive duplicates

*** **Partitioning operations** — header ***algorithm***

is_partitioned (C++11) determines if the range is partitioned by the given predicate

partition divides a range of elements into two groups

partition_copy (C++11) copies a range dividing the elements into two groups

stable_partition divides elements into two groups while preserving their relative order

partition_point (C++11) locates the partition point of a partitioned range

*** **Sorting operations** — header ***algorithm***

is_sorted (C++11) checks whether a range is sorted into ascending order

is_sorted_until (C++11) finds the largest sorted subrange

sort sorts a range into ascending order

partial_sort sorts the first N elements of a range

partial_sort_copy copies and partially sorts a range of elements

stable_sort sorts a range of elements while preserving order between equal elements

nth_element partially sorts the given range making sure that it is partitioned by the given element

*** **Binary search operations on sorted ranges** — header ***algorithm***

lower_bound returns an iterator to the first element not less than the given value

upper_bound returns an iterator to the first element greater than a certain value

binary_search determines if an element exists in a certain range

equal_range returns range of elements matching a specific key

*** **Other operations on sorted ranges** — header ***algorithm***

merge merges two sorted ranges

inplace_merge merges two ordered ranges in-place

*** **Set operations on sorted ranges** — header ***algorithm***

includes returns true if one set is a subset of another

set_difference computes the difference between two sets

set_intersection computes the intersection of two sets

set_symmetric_difference computes the symmetric difference between two sets

set_union computes the union of two sets

*** **Heap operations** — header ***algorithm***

is_heap (C++11) checks if the given range is a max heap

is_heap_until (C++11) finds the largest subrange that is a max heap

make_heap creates a max heap out of a range of elements

push_heap adds an element to a max heap

pop_heap removes the largest element from a max heap

sort_heap turns a max heap into a range of elements sorted in ascending order

*** **Minimum/maximum operations** — header ***algorithm***

max returns the greater of the given values

max_element returns the largest element in a range

min returns the smaller of the given values

min_element returns the smallest element in a range

minmax (C++11) returns the smaller and larger of two elements

minmax_element (C++11) returns the smallest and the largest elements in a range

clamp (C++17) clamps a value between a pair of boundary values

*** **Comparison operations** — header ***algorithm***

equal determines if two sets of elements are the same

lexicographical_compare returns true if one range is lexicographically less than another

compare_3way (C++20) compares two values using three-way comparison

lexicographical_compare_3way (C++20) compares two ranges using three-way comparison

*** **Permutation operations** — header ***algorithm***

is_permutation (C++11) determines if a sequence is a permutation of another sequence

next_permutation generates the next greater lexicographic permutation of a range of elements

prev_permutation generates the next smaller lexicographic permutation of a range of elements

*** **Numeric operations** — header ***numeric***

iota (C++11) fills a range with successive increments of the starting value

accumulate sums up a range of elements

inner_product computes the inner product of two ranges of elements

adjacent_difference computes the differences between adjacent elements in a range

partial_sum computes the partial sum of a range of elements

reduce (C++17) similar to std::accumulate, except out of order

exclusive_scan (C++17) similar to std::partial_sum, excludes the ith input element from the ith sum

inclusive_scan (C++17) similar to std::partial_sum, includes the ith input element in the ith sum

transform_reduce (C++17) applies a functor, then reduces out of order

transform_exclusive_scan (C++17) applies a functor, then calculates exclusive scan

transform_inclusive_scan (C++17) applies a functor, then calculates inclusive scan

***** Operations on uninitialized memory** — header ***memory***

uninitialized_copy copies a range of objects to an uninitialized area of memory

uninitialized_copy_n (C++11) copies a number of objects to an uninitialized area of memory

uninitialized_fill copies an object to an uninitialized area of memory, defined by a range

uninitialized_fill_n copies an object to an uninitialized area of memory, defined by a start and a count

uninitialized_move (C++17) moves a range of objects to an uninitialized area of memory

uninitialized_move_n (C++17) moves a number of objects to an uninitialized area of memory

uninitialized_default_construct (C++17) constructs objects by default-initialization in an uninitialized area of memory, defined by a range

uninitialized_default_construct_n (C++17) constructs objects by default-initialization in an uninitialized area of memory, defined by a start and a count

uninitialized_value_construct (C++17) constructs objects by value-initialization in an uninitialized area of memory, defined by a range

uninitialized_value_construct_n (C++17) constructs objects by value-initialization in an uninitialized area of memory, defined by a start and a count

destroy_at (C++17) destroys an object at a given address

destroy (C++17) destroys a range of objects

destroy_n (C++17) destroys a number of objects in a range

— 25 —

Dynamiczna identyfikacja typu

W językach obiektowych, jak wiemy, istnieje w warunkach dziedziczenia rozróżnienie między typem statycznym obiektu a jego typem dynamicznym (rozdz. 21.4, str. 471). Dzięki mechanizmowi polimorfizmu zazwyczaj nie musimy się tym zajmować. Są jednak sytuacje, gdy rozpoznanie w programie prawdziwego typu obiektu jest konieczne, bo na przykład zależy od tego dalszy przebieg programu. Mechanizm dynamicznego rozpoznawania typów nazywany jest RTTI (od *run-time type identification*) i obejmuje dwa główne zagadnienia: rozpoznanie typu w celu porównania go z typem innego obiektu oraz rozpoznanie typu w celu sprawdzenia poprawności i wykonania rzutowania (konwersji). Do realizacji pierwszego z tych zadań służy operator **typeid**. Do dynamicznego rzutowania i sprawdzania jego poprawności służy operator **dynamic_cast**.

Należy pamiętać, że mechanizm RTTI obciąża program wynikowy dodatkowym kodem, a zatem zmniejsza jego efektywność. Dlatego kompilatory pozwalają, poprzez użycie odpowiednich opcji, na włączanie lub wyłączanie tego mechanizmu. Która z tych opcji jest domyślna — zależy od kompilatora.

PODROZDZIAŁY:

25.1 Operator *typeid*

Argumentem operatora **typeid** (z nagłówka ***typeinfo***) może być nazwa typu lub dowolne wyrażenie o określonej wartości. Operator zwraca identyfikator typu argumentu, który jest obiektem klasy **type_info**. Klasa ta, poprzez przeciążenie operatorów '==' i '!=' zapewnia możliwość porównywania obiektów reprezentujących typy, na przykład:

```
#include <typeinfo>
// ...
double x = 1.5;
// ...
if (typeid(x) == typeid(double)) ... // true
if (typeid(x) == typeid(36.0))   ... // true
if (typeid(x) == typeid(3))      ... // false
if (typeid(x) != typeid(3))      ... // true
if (typeid(x) != typeid(int))    ... // true
```

Klasa **type_info** posiada metodę **name**, która zwraca C-napis zawierający nazwę

typu: nazwy te nie muszą pokrywać się ze standardowymi nazwami typów, choć mogą. Na przykład

```
typeid(int).name()
```

zwraca nazwę '**int**' w środowisku ***VC++***, ale ***g++*** i Intelowski ***icpc*** dają po prostu nazwę 'i'. Zwykle nazwy nie są nam do niczego potrzebne, ważne jest tylko porównywanie typów.

Ciekawsze i bardziej pożyteczne jest rozpoznawanie typów w warunkach dziedziczenia. Szczegóły rozpatrzmy na przykładzie następującego programu:

P218: ***rtti.cpp*** Dynamiczne rozpoznawanie typu

```
#include <iostream>
#include <typeinfo>
using namespace std;

struct Pojazd { };
struct Samochod : Pojazd { };

struct Budynek {
    virtual ~Budynek() { }
};
struct Stacja : Budynek { };



int main() {
    Pojazd    poj;
    Samochod  sam1,sam2;
    Samochod* p_sam1 = &sam1;
    Pojazd*   p_sam2 = &sam2;
    Pojazd&   r_sam1 = sam1;
    cout << "    poj: " << typeid(poj).name()      << endl
         << "   sam1: " << typeid(sam1).name()     << endl
         << "   sam2: " << typeid(sam2).name()     << endl
         << " p_sam1: " << typeid(p_sam1).name()   << endl
         << " p_sam2: " << typeid(p_sam2).name()   << endl
         << "*p_sam1: " << typeid(*p_sam1).name() << endl
         << "*p_sam2: " << typeid(*p_sam2).name() << endl
         << " r_sam1: " << typeid(r_sam1).name()   << endl;

    cout << "Typy *p_sam1 i *p_sam2 sa "
         << (typeid(*p_sam1) == typeid(*p_sam2) ?
                   "takie same\n" : "rozne\n")    << endl;

    Budynek  bud;
    Stacja   sta1,sta2;
```

```
36      Stacja*  p_sta1 = &sta1;
37      Budynek* p_sta2 = &sta2;
38      Budynek& r_sta1 = sta1;
39      cout << "    bud: " << typeid(bud).name()      << endl
40           << "   sta1: " << typeid(sta1).name()     << endl
41           << "   sta2: " << typeid(sta2).name()     << endl
42           << " p_sta1: " << typeid(p_sta1).name()   << endl
43           << " p_sta2: " << typeid(p_sta2).name()   << endl
44           << "*p_sta1: " << typeid(*p_sta1).name() << endl
45           << "*p_sta2: " << typeid(*p_sta2).name() << endl
46           << " r_sta1: " << typeid(r_sta1).name()   << endl;
47
48      cout << "Typy *p_sta1 i *p_sta2 sa "
49           << (typeid(*p_sta1) == typeid(*p_sta2) ?
50                     "takie same\n" : "rozne\n");
51  }
```

Mamy w programie dwie pary klas: klasę **Pojazd** i dziedziczącą z niej klasę **Samochod** oraz analogicznie klasę **Budynek** i dziedziczącą z niej klasę **Stacja**. Między tymi parami klas zachodzi jednak głęboka różnica: para **Pojazd** ← **Samochod** nie zawiera metod wirtualnych, a więc nie są to klasy polimorficzne. Natomiast para **Budynek** ← **Stacja** jest polimorficzna, bo destruktor klasy **Budynek** jest wirtualny, a dziedziczenie jest publiczne (nie napisaliśmy tego jawnie, ale użyliśmy słowa kluczowego **struct**, a nie **class**). Przypatrzmy się wydrukowi tego programu:

```
     poj: 6Pojazd                          ( 1)
    sam1: 8Samochod                        ( 2)
    sam2: 8Samochod                        ( 3)
  p_sam1: P8Samochod                       ( 4)
  p_sam2: P6Pojazd                         ( 5)
 *p_sam1: 8Samochod                        ( 6)
 *p_sam2: 6Pojazd                          ( 7)
  r_sam1: 6Pojazd                          ( 8)
Typy *p_sam1 i *p_sam2 sa rozne            ( 9)

     bud: 7Budynek                         (11)
    sta1: 6Stacja                          (12)
    sta2: 6Stacja                          (13)
  p_sta1: P6Stacja                         (14)
  p_sta2: P7Budynek                        (15)
 *p_sta1: 6Stacja                          (16)
 *p_sta2: 6Stacja                          (17)
  r_sta1: 6Stacja                          (18)
Typy *p_sta1 i *p_sta2 sa takie same       (19)
```

Linie 1-3 i 11-13 wydruku nie wymagają komentarza: typ jest dokładnie taki, jaki jest typ obiektu (wiodące znaki są dodawane do nazw typów przez kompilator; nie mu-

simy się nimi przejmować — inny kompilator może wewnętrznie używać innych nazw typów). Ponieważ argumentami operatora **typeid** są tu obiekty, do których odnosimy się przez ich nazwę, a nie poprzez wskaźnik lub referencję, żadnego polimorfizmu tak czy owak nie ma.

Typem drukowanym w liniach 4-5 (oraz 14-15) wydruku jest typ *wskaźnika*, a nie wskazywanego przez ten wskaźnik obiektu (nazwa 'P8Samochod' to nazwa typu *wskaźnik do 8Samochod*; 'P' od *pointer*). Porównując wydruk z linii 4 i 5 oraz z linii 14 i 15 widzimy, że w tym przypadku polimorfizm też nie ma nic do rzeczy: typem jest prawdziwy, zadeklarowany typ *wskaźnika*.

Jeśli p jest wskaźnikiem, to wyrażenie **typeid(p)** określa typ tego wskaźnika, a nie typ obiektu wskazywanego przez ten wskaźnik.

Inaczej jest, kiedy pytamy bezpośrednio o typ obiektu wskazywanego przez wskaźnik, a więc o typ wartości wyrażenia *p, gdzie p jest wskaźnikiem. Ponieważ do obiektu odnosimy się teraz przez wskaźnik, polimorfizm może zadziałać, pod warunkiem, że mamy do czynienia z klasami polimorficznymi, a więc zadeklarowana w nich jest choć jedna metoda wirtualna; może nią być sam destruktor, jak to jest w naszym przykładzie dla pary klas **Budynek** ← **Stacja**.

Spójrzmy na linie 6 i 7 wydruku. Prawdziwym typem obiektów wskazywanych zarówno przez wskaźnik p_sam1 jak i p_sam2 jest typ pochodny **Samochod**. Jednak typem wskaźnika w pierwszym przypadku jest **Samochod***, a w drugim **Pojazd***. Ponieważ te klasy nie są polimorficzne, typ obiektów wskazywanych *p_sam1 i *p_sam2 zostanie rozpoznany według deklaracji wskaźników, a więc statycznie. Tak więc znalezionym typem wartości wyrażeń *p_sam1 i *p_sam2 będzie odpowiednio **Samochod** i **Pojazd**. Inaczej jest dla obiektów będących wartościami wyrażeń *p_sta1 i *p_sta2. Prawdziwy ich typ to typ pochodny **Stacja**. Odnosimy się do tych obiektów przez wskaźnik, a klasy są polimorficzne. Zatem tym razem rozpoznany zostanie w obu wypadkach prawdziwy typ wskazywanych obiektów — patrz linie 16 i 17.

Rozpoznawanie prawdziwych typów dla klas polimorficznych zachodzi, jak wiemy, również wtedy, gdy do obiektów odnosimy się poprzez referencję. Przykład mamy w liniach 8 i 18: bez polimorfizmu rozpoznany został typ statyczny **Pojazd**, według zadeklarowanego typu referencji r_sam1, a nie prawdziwy typ obiektu, do którego ta referencja się odnosi, czyli **Samochod** (linia 8). Natomiast dla klas polimorficznych rozpoznany został prawdziwy typ obiektu (linia 18).

Linie 9 i 19 wskazują jeszcze raz, że porównanie typów obiektów wskazywanych przez wskaźniki lub referencje może zawieść dla klas, które polimorficzne nie są. Typy obiektów *p_sam1 i *p_sam2 zostały rozpoznane jako różne (linia 9), choć tak naprawdę są takie same, tylko typy wskaźników wskazujących na te obiekty są różne. Dla klas polimorficznych typy *p_sta1 i *p_sta2 zostały prawidłowo rozpoznane jako takie same (linia 19), mimo że typy wskaźników były różne.

25.2 Operator *dynamic_cast*

Operator **dynamicznego rzutowania** (konwersji) stosuje się, gdy prawidłowość przekształcenia nie może być sprawdzona na etapie kompilacji, bo zależy od typu obiektu klasy polimorficznej, który znany jest dopiero w czasie wykonania. Jego użycie ma sens w sytuacjach, gdy mamy do czynienia z obiektami różnych klas powiązanych ze sobą hierarchią dziedziczenia.

Wyobraźmy sobie, że zdefiniowane są klasy **A** i dziedzicząca z niej klasa **B**. Klasy muszą być polimorficzne, a więc w klasie **A** musi istnieć choć jedna metoda wirtualna (patrz rozdz. 21.4, str. 471). Załóżmy, że istnieje też funkcja, której parametr ma typ **A*** lub **A&**. Czy można wywołać tę funkcję z argumentem typu **B*** lub **B**? Ponieważ obiekt klasy **B** można traktować jako obiekt klasy **A**, bo zawiera na pewno wszystkie składowe zadeklarowane w klasie **A** (i, być może, inne), więc takie wywołanie jest legalne i konwersja od typu **B*** do typu **A*** zajdzie niejawnie. Na marginesie zauważmy, że mówimy tu o przekazywaniu obiektów do funkcji przez wskaźnik lub referencję: *nie* przez wartość. Wartości typu **B** nie mogą wystąpić w roli wartości typu **A** na stosie programu, bo, na przykład, mają inny rozmiar (formalnie jest to możliwe, ale wiąże się z „szatkowaniem" obiektu — na stosie położony zostanie podobiekt klasy **A** obiektu klasy **B**, co zazwyczaj nie jest tym czego byśmy chcieli).

Inna jest sytuacja, jeśli, odwrotnie, mamy zmienną typu **B***, a przypisać jej chcielibyśmy wartość typu **A***. Obiekt wskazywany przez ten wskaźnik może, ale nie musi być obiektem klasy pochodnej **B**. Konwersja zatem w tę stronę, od wskaźnika na obiekt klasy bazowej do wskaźnika na obiekt klasy pochodnej, może się nie powieść. Musi zatem być dokonana jawnie. Ale kompilator nie jest w stanie stwierdzić, czy takie przekształcenie jest prawidłowe, bo nie wie, jaki będzie typ obiektu wskazywanego podczas wykonania programu. Nie można zatem użyć statycznego operatora konwersji. I właśnie wtedy przydaje się operator konwersji dynamicznej **dynamic_cast**. Składnia jego użycia jest następująca:

```
dynamic_cast<Typ>(wyrazenie)
```

Zachodzą dwa przypadki:

- Typ **Typ** jest typem wskaźnikowym do polimorficznej klasy pochodnej, czyli **Typ** = **B***, a wartością wyrażenia wyrazenie jest wskazanie na obiekt klasy bazowej **A**, czyli wartość o typie **A***. Operacja rzutowania polega wtedy na sprawdzeniu, czy obiekt wskazywany przez wyrazenie jest w rzeczywistości obiektem klasy pochodnej **B**; po tym teście wartością całego tego wyrażenia staje się
  - wartość wskaźnikowa typu **B*** wskazująca na obiekt wskazywany przez wyrazenie, jeśli test wypadł pomyślnie;
  - zero (wskaźnik pusty, **nullptr**), jeśli test się nie powiódł, czyli obiekt wskazywany przez wyrazenie nie jest obiektem klasy **B**. Można to w programie sprawdzić odpowiednim **if**'em i podjąć stosowne działania bez przerywania programu.
- Typ **Typ** jest typem referencyjnym do polimorficznej klasy pochodnej, czyli **Typ** = **B&**, a wartością wyrażenia wyrazenie jest l-wartość obiektu klasy **A**. Operacja

rzutowania polega wtedy na sprawdzeniu, czy obiekt wyrazenie jest w rzeczywistości obiektem klasy pochodnej **B**; po tym teście,

- jeśli wypadł pomyślnie, wartością całego wyrażenia staje się referencja typu **B&** do obiektu wyrazenie;
- jeśli wypadł niepomyślnie, bo obiekt wyrazenie nie jest typu **B**, to zgłaszany jest wyjątek typu **bad_cast** (z nagłówka ***typeinfo***), który można przechwycić i podjąć stosowne działania bez przerywania programu. Tym razem nie można przekazać informacji o niepowodzeniu zerową wartością wyrażenia, bo nie ma czegoś takiego jak pusta referencja (tak jak jest pusty, czyli zerowy, wskaźnik). Dlatego do obsługi tego przypadku wybrano metodę zgłoszenia wyjątku.

Rozpatrzmy przykład:

P219: ***prgr.cpp*** Rzutowanie dynamiczne

```
#include <iostream>
#include <string>
using namespace std;

class Program {
protected:
    string name;
public:
    Program(string n)
        : name(n)
    { }
    virtual void print() = 0;
    virtual ~Program() { };
};

class Freeware : public Program {
public:
    Freeware(string n)
        : Program(n)
    { }
    void print() {
        cout << "Free : " << name << endl;
    }
};

class Shareware : public Program {
    int price;
public:
    Shareware(string n, int c)
        : Program(n), price(c)
    { }
```

```
32      void print() {
33          cout << "Share: "  << name
34               << ", price " << price << endl;
35      }
36      int getPrice() {
37          return price;
38      }
39  };
40
41  int total(Program* prgs[], int size) {
42      Shareware* sh;
43      int tot = 0;
44      for (int i = 0; i < size; ++i) {
45          prgs[i]->print();
46          if ( sh = dynamic_cast<Shareware*>(prgs[i]) )
47              tot += sh->getPrice();
48      }
49      return tot;
50  }
51
52  int main() {
53      Freeware anjuta("Anjuta"); Shareware wz("WinZip",30);
54      Freeware mysql("MySQL");   Shareware rar("RAR",25);
55
56      Program* prgs[] = { &anjuta, &wz, &mysql, &rar };
57
58      int tot = total(prgs, sizeof(prgs)/sizeof(prgs[0]));
59
60      cout << "\nTotal: $" << tot << endl;
61  }
```

Definiujemy w tym programie klasę abstrakcyjną **Program** i dwie klasy pochodne: **Freeware** i **Shareware**. Klasy są polimorficzne, bo w klasie bazowej zadeklarowana została metoda (czysto) wirtualna **print**. Zauważmy, że w klasie **Shareware** zadeklarowaliśmy dodatkowe składowe, których nie było w klasie bazowej. Są nimi pole price i metoda **getPrice**. Funkcja **total** pobiera tablicę wskaźników typu **Program***. W pętli, dla każdego wskaźnika z tablicy, wywołuje metodę **print**. Żadne rzutowanie nie jest potrzebne: polimorfizm zadba o to, aby za każdym razem wywołana została metoda z prawdziwej klasy wskazywanego obiektu. Ale funkcja **total** oprócz wypisania listy programów ma obliczyć całkowity ich koszt. Nie da się jednak wywołać metody **getPrice** dla wskaźnika typu **Program***, bo w klasie **Program** takiej metody nie ma. Wiemy jednak, że niektóre programy są klasy **Shareware** i tylko dla nich cena jest w ogóle zdefiniowana. Rzutujemy zatem wskaźnik typu bazowego **Program***, który jest typem każdego elementu tablicy, na typ pochodny **Shareware*** (linia 46). Jeśli to się powiedzie, to zmienna sh typu **Shareware*** będzie niezerowa i będzie wskazywać na obiekt, który jest na pewno obiektem klasy pochodnej **Shareware**, a nie klasy

Freeware. Zatem teraz będzie możliwe wywołanie dla tego obiektu metody **getPrice** (linia 47). Jeśli natomiast wskazywany przez kolejny element tablicy obiekt nie jest obiektem typu **Shareware** (tylko **Freeware**, bo innej możliwości nie ma), to zmienna sh będzie zerowa, test w instrukcji warunkowej **if** z linii 46 da wartość **false** i wywołania funkcji **getPrice** nie będzie — nie mogłoby się ono powieść, bo w klasie **Freeware** metody **getPrice** nie ma.

Wynikiem działania programu jest

```
Free : Anjuta
Share: WinZip, price 30
Free : MySQL
Share: RAR, price 25

Total: $55
```

Aby zilustrować drugą możliwość, w powyższym programie moglibyśmy zapisać funkcję **total** następująco:

```
1     int total(Program* prgs[], int size) {
2         int tot = 0;
3         for (int i = 0; i < size; ++i) {
4             prgs[i]->print();
5             try {
6                 Shareware& sh =
7                     dynamic_cast<Shareware&>(*prgs[i]);
8                 tot += sh.getPrice();
9             } catch(bad_cast) { }
10        }
11        return tot;
12    }
```

Tym razem rzutujemy (linia 7) nie wskaźnik, ale wartość obiektową wskazywaną przez ten wskaźnik: prgs[i] jest wskaźnikiem, więc *prgs[i] jest l-wartością wskazywanego obiektu. Jeśli rzutowanie do typu **Shareware&** powiedzie się, to odczytujemy i dodajemy cenę. Jeśli nie powiedzie się, to znaczy obiekt nie był klasy pochodnej **Shareware**, zgłaszany jest wyjątek i sterowanie nie dochodzi do linii 8, tylko wchodzi do frazy **catch**, gdzie wyjątek ten jest po prostu ignorowany, po czym program przechodzi do wykonania kolejnego obrotu pętli.

Skorowidz