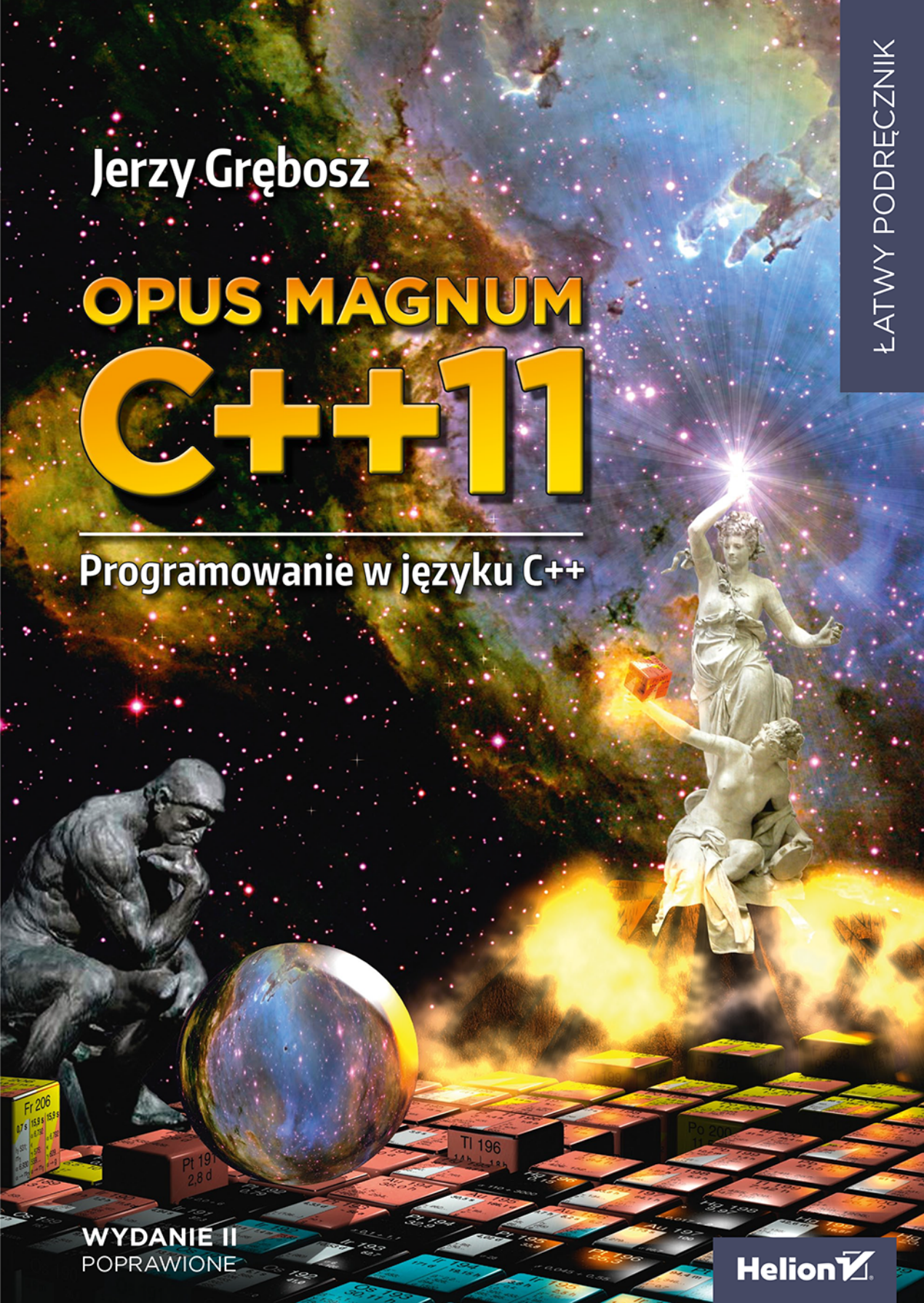

Jerzy Gręboszz
OPUS MAGNUM
C++11
Programowanie w języku C++
ŁATWY PODRĘCZNIK
WYDANIE II
POPRAWIONE
Helion

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki i opracowanie graficzne książki: Jerzy Grębosz

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

Wydanie drugie

ISBN: 978-83-283-6965-8

Printed in Poland

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/ocpp12
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Kody źródłowe wybranych przykładów dostępne są pod adresem:
ftp://ftp.helion.pl/przyklady/ocpp12.zip

Spis treści trzech tomów

0 Proszę tego nie czytać!

0.1 Zaprzyjaźnijmy się!

Jeśli mamy ze sobą spędzić parę godzin, to proponuję – przejdźmy na „ty". Moje nazwisko zobaczyłeś już na okładce, dodam więc tylko, że książka ta powstała na podstawie moich doświadczeń jako programisty w niemieckich, francuskich i włoskich instytutach fizyki jądrowej, z którymi mój macierzysty Instytut Fizyki Jądrowej im. Henryka Niewodniczańskiego Polskiej Akademii Nauk (w Krakowie) od lat prowadzi eksperymenty z dziedziny badania struktury jądra atomowego metodami spektroskopowymi. Eksperymenty te wymagają rozbudowanego oprogramowania, które współtworzyłem. Pierwotnie robiłem to w języku C klasycznym, ale gdy pojawił się język C++, nastąpił w tej pracy przełom. Ten niezwykły C++ szybko stał się jednym z najpopularniejszych i najważniejszych języków programowania i tak jest do dziś.

> Jestem oczarowany jasnością, prostotą i logiką programowania w języku C++ i dlatego proponuję Ci spędzenie ze mną paru chwil w krainie programowania orientowanego obiektowo.

Będzie to niezwykła podróż. C++ ciągle się udoskonala i jest bardzo profesjonalny. Stosowany jest tam, gdzie konieczna jest duża wydajność. Najczęściej używa się go do tworzenia gier komputerowych, do obliczeń naukowych, technicznych, w medycynie, w przemyśle, w bankowości... Na lądzie, na wodzie, w powietrzu i w przestrzeni kosmicznej. Nie wierzysz?

Możliwe, że nawet nie zdajesz sobie sprawy, ile razy korzystałeś z efektów pracy języka C++. Znane Ci zapewne programy Adobe Photoshop, Illustrator, Acrobat napisano przy użyciu języka C++. Komputerowe efekty w filmach „Gwiezdne wojny", „Władca Pierścieni", „Spider-Man" i innych wykonane zostały przez programy napisane w C++. W C++ napisano najistotniejsze procedury sterowania samolotami F-16 czy F-35. Niemal całe oprogramowanie sterowania silnikami okrętowymi olbrzymich kontenerowców i tankowców firmy MAN B&W Diesel A/S zostało napisane w C++.

NASA – amerykańska agencja lotów kosmicznych – posługuje się językiem C++ w wielu swych przedsięwzięciach. Zarówno w naziemnej kontroli lotów kosmicznych, jak i tam, w przestrzeni kosmicznej. Duża część oprogramowania w Międzynarodowej Stacji Kosmicznej została napisana w C++. Na orbicie okołoziemskiej krążą teleskopy kosmiczne, które potrafią swym zasięgiem penetrować tak odległe części

Wszechświata, że możemy obserwować najdawniejsze fazy jego powstania. I do tego użyto oprogramowania napisanego w C++.

W grafice na okładce tej książki nawiązuję właśnie do tego faktu.

Naprawdę więc nie tylko „na lądzie, wodzie i w powietrzu". Nawet na powierzchni planety Mars. W marsjańskim łaziku Curiosity pracuje program w C++ analizujący obraz z kamer i planujący dalszą trasę łazika.

Taka jest teraźniejszość języka C++; a przyszłość? Kto wie, może leży właśnie w Twoich rękach? Wiem, co mówię. Otrzymałem od czytelników moich książek wiele listów informujących, jak niezwykle potoczyły się ich dalsze, programistyczne losy. Teraz czas na Ciebie...

Książka ta jest następcą książki „Symfonia C++ Standard"

Na rynku księgarskim przez wiele lat obecna była moja książka „Symfonia C++ Standard". Miała ok. 1200 stron, wiele wydań i cieszyła się ogromną poczytnością. Gdy pojawił się standard języka zwany C++11, należało napisać nową książkę. To właśnie efekt tej pracy.

Przy pisaniu tej nowej książki korzystałem oczywiście z niektórych fragmentów tamtej, a stary tekst musiał zostać zmodyfikowany i adaptowany do nowego standardu. Nowy standard wprowadził wiele nowych, ciekawych rozwiązań i należało je także w przystępny sposób opisać. W rezultacie obecna książka ma o 500 stron więcej. To nie jest już nowe wydanie „Symfonii". To nowa książka. Ma też całkiem nowy tytuł. Nie chciałem popełnić tego samego błędu po raz drugi. Pierwsza moja książka miała tytuł „Symfonia C++", a druga – tytuł dość podobny: „Symfonia C++ Standard". W rezultacie wielu osobom, które chciały kupić tę nową książkę na aukcjach internetowych, sprzedawano tę starą, nadającą się wówczas już tylko na makulaturę.

Dlatego teraz postanowiłem zmienić tytuł tak wyraziście, żeby nie narażać czytelników na wspomniane nieporozumienia.

Dlaczego C++11, a nie C++17

Jeśli choć trochę interesujesz się językiem C++, to pewnie wiesz, że w chwili gdy obecne, drugie wydanie *Opusu* idzie do druku, powstały już standardy C++14 i C++17. Mógłbyś zapytać, dlaczego ta książka omawia standard C++11, a nie od razu C++14 lub C++17. Odpowiedź jest prosta: gdy pracowałem nad pierwszym wydaniem tej książki, mój kompilator ostrzegał mnie, że implementacja tych nowszych standardów jest w nim jeszcze *„eksperymentalna i niepełna"* . Nie mogłem więc w odpowiedzialny sposób kompilować w nim programów w standardzie C++17 ani tym bardziej opowiedzieć Ci o różnych aspektach danej nowej cechy, jeśli sam nie mogłem tego gruntownie sprawdzić. Z C++14 i C++17 musiałem więc wtedy jeszcze zaczekać. Z drugiej strony nie był to wielki problem, bo różnice między C++11, C++14 i C++17 są raczej niewielkie. Natomiast przejście ze starego standardu C++03 do C++11 było bardzo znaczące. Twórca tego języka, Bjarne Stroustrup, napisał wtedy: „**C++11 to jakby nowy język**".

> W końcu jednak przyszedł czas na omówienie C++14 i C++17. Napisałem więc dodatkowe rozdziały omawiające nowe cechy języka wprowadzone przez standardy C++14 i C++17.

Pierwotnie myślałem o rozszerzeniu tej książki, ale gdy zdałem sobie sprawę, że *Opus* jest i tak dość obszerny, postanowiłem wydać te zagadnienia w postaci odrębnej książki.

Powstał więc jakby osobny, czwarty tom uzupełniający tę książkę. Ma on tytuł *Opus magnum C++*. ***Misja w nadprzestrzeń C++14/17***.

Jeśli spodoba Ci się mój sposób opowiadania o C++, to poszukaj *Misji w nadprzestrzeń* w księgarniach. Jest ona jakby ciągiem dalszym tej książki.

Wersja języka

Absolutnym autorytetem w sprawie C++11 był dla mnie dokument standardu języka C++ (ISO International Standard ISO/IEC 14882:2011 – Programming Language C++). W razie jakichkolwiek niejasności odsyłam do tego tekstu. (Uprzedzam jednak, że jest napisany w bardzo sformalizowanym stylu i nie nadaje się jako podręcznik do nauki).

Oczywiście nawet w C++11 można pisać programy „po staremu". Można pisać nawet programy niemające z techniką orientowaną obiektowo nic wspólnego. Tylko że to tak, jakby zwiedzać Granadę, mając zamknięte oczy[1)].

Dlaczego ta książka jest taka gruba?

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

Pióro

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

Dla kogo jest ta książka

Książka ta jest napisana z myślą o czytelnikach, którym nieobce są takie pojęcia jak program, instrukcja, dyskietka, monitor. Oczywiście najlepiej by było, gdybyś znał już jakiś inny, choćby najprostszy język programowania. Żebyś wiedział, co to jest komputer, dysk, program, instrukcja. Jeśli tak nie jest, to – jak sądzę – wystarczy Ci 20-minutowa rozmowa z kolegą, który już programuje, lub przeczytanie jakiejś strony internetowej tłumaczącej, co to znaczy programować komputer.

Pisząc tę książkę, musiałem najpierw określić sobie, dla kogo będzie ona przeznaczona. Otóż jest ona dla tzw. szerokiego grona czytelników. Pisałem ją tak, by była podręcznikiem dla 14-letniego programisty amatora, jak i dla zawodowego

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

informatyka. Trudno te sprawy pogodzić, więc są w niej kompromisy w obu kierunkach. Co do jednego z takich kompromisów mam najwięcej obaw:

Czołem, bracia angliści!

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

Od czasu, gdy pojawiła się pierwsza „Symfonia C++", sytuacja ze znajomością języka angielskiego wśród naszych rodaków bardzo się zmieniła. W Krakowie nawet pani w kiosku na mojej ulicy mówi po angielsku. Wydawałoby się więc, że teraz można by już usunąć te przypisy o wymowie angielskich słów. No, ale przecież Kraków – z tysiącami zagranicznych turystów – nie jest w tym przypadku reprezentatywny. Chciałbym, żeby ta książka służyła także bardzo młodym ludziom w różnych, nawet odległych zakątkach Polski. Oni też mają prawo wzbić się do swego lotu do gwiazd.

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

A teraz o strukturze tej książki

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

Książka ta opisuje sam język C++. Pierwotnie odczuwałem pokusę, aby opowiedzieć także o klasach z biblioteki standardowej C++. Gdy jednak zobaczyłem, jak bardzo książka się rozrasta, zdecydowałem, że to jednak temat na osobną książkę. Ta i tak jest bardzo gruba. Ograniczyłem się więc tutaj do omówienia potrzebnych Ci w pierwszej fazie nauki bezwzględnie najważniejszych klas bibliotecznych: wektorów, stringów i oczywiście strumieni wejścia/wyjścia. Są one opisane bardzo obszernie.

Metodą kolejnych przybliżeń

Nie liczę na to, że czytając tę książkę, zrozumiesz wszystko od razu. Lubię powtarzać, że mądrość polega między innymi na umiejętności odróżnienia spraw ważnych od mniej ważnych. Ucząc się nowego języka programowania, nie daj się zasypać szczegó-

łami. Dlatego w tekście wielokrotnie sugeruję, żebyś przy pierwszym czytaniu książki opuścił niektóre paragrafy. Nie wszystkie niuanse danego zagadnienia są ważne już na samym początku nauki. Lepiej najpierw mieć ogólny pogląd, a dopiero potem zagłębiać się w trudniejsze szczegóły. Te bardziej szczegółowe fragmenty poprzedza zwykle znaczek

Te miejsca opatrzyłem także adnotacją: „dla wtajemniczonych". Są tam uwagi wybiegające nieco do przodu, ale zrozumieją je czytelnicy, którzy mają za sobą już pierwsze czytanie – czyli czytelnicy już „wtajemniczeni".

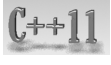
Nie szanuj tej książki. Przygotuj sobie kolorowe (tzw. niekryjące) flamastry (tzw. markery) i czytając, zakreślaj ważne dla Ciebie wyrazy czy myśli. Na marginesach pisz ołówkiem swoje uwagi i komentarze. Chociażby miały to być teksty typu: „Co za bzdura!" albo „Nie rozumiem, dlaczego...", albo „Porównaj dwie strony wcześniej". Ta aktywność zaprocentuje Ci natychmiast, bo mając tak osobisty stosunek do tekstu, łatwiej koncentrować na nim uwagę – co jest warunkiem *sine qua non* szybkiej nauki.

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

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

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

Znaki graficzne, piktogramy, numerki



Jeśli programowałeś w dawniejszych wersjach języka C++, a tę książkę czytasz, żeby poznać nowości w C++11, to zwróć uwagę na paragrafy, w których na marginesie pojawia się taki znak jak ten, który widzisz tu z lewej.

C++11 Czasem, gdy nowinka dotyczy tylko jednej linijki, ma on taką skróconą formę.

Na marginesie występują też i inne znaczki, ale nie ma sensu o nich pisać, bo ich znaczenie będzie dla Ciebie oczywiste. Natomiast w tekstach programów spotkasz miejsca oznaczone czarnymi numerkami.

23 Są one po to, żeby potem łatwo było nam te fragmenty programu omawiać. Dodatkowo są też numerki drugiego typu. Występują one we fragmentach, gdzie pokazany jest wygląd ekranu komputera, na którym nasz program wypisał jakiś tekst. Właśnie tam (czasami) pojawią się numerki „kwadratowe" 23. Jak widać, liczba jest ta sama, ale teraz jest ona na kwadratowym tle, które (umówmy się) oznacza ekran komputera. W ten sposób oznaczam, że dany tekst jest ekranowym rezultatem pracy instrukcji programu wcześniej oznaczonej jako 23.

> *Tych kwadratowych numerków nie stosuję często, bo zwykle dokładnie wiadomo, z której instrukcji pochodzi dany tekst. Czasem jednak obecność takiego kwadratowego numerka się nam przyda.*

W programach występują nazwy zmiennych. Celowo wymyślam dla nich polskie nazwy, bo wtedy wyraźnie widać, co w programie jest naszą twórczością, a co stanowią

(angielskie) słowa kluczowe języka C++. Zatem w naszych programach będziesz mógł na przykład spotkać się ze zmiennymi o nazwach przelicznik, moc_dopuszczalna, dzwiek. Oczywiście ten ostatni wyraz to dźwięk, ale tych polskich liter nie wolno mi użyć w programie. Nazwy zmiennych mają składać się z liter alfabetu angielskiego. Dlatego trzeba jednak pisać trochę dziwacznie: dzwiek. Wkrótce się z tym oswoisz.

Z drugiej strony jednak, gdy potem rozmawiamy o programie, to pozwalam sobie tych polskich liter tam używać, bo w polskim zdaniu wygląda to czytelniej. Zatem piszę, że „do funkcji fun wysyłamy argument dźwięk". W tych opisach pozwalam sobie nawet na coś więcej: pozwalam sobie na fleksję, czyli końcówki odmiany przez przypadki. Na przykład piszę, że: „inkrementujemy moc_dopuszczalną" lub że „do dźwięku przypisujemy wartość 330".

Pamiętaj, że te zabiegi spolszczające wolno mi robić poza programem, w jego opisie. Natomiast w samym programie nazwa zmiennej musi być zawsze bez tych polskich liter – i zawsze w mianowniku.

Szukajcie, a znajdziecie

Na koniec uwaga zecerska. W tekście książki czasem występują przypisy u dołu strony. Jeśli – z przyczyn typograficznych – jakiś przypis nie będzie mógł się pojawić u dołu strony, należy go szukać na stronie następnej.

Internetowe wsparcie

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

jerzy.grebosz@ifj.edu.pl

Z góry wyrażam moją wdzięczność za nawet najdrobniejsze uwagi. Proszę, napisz nawet o zauważonych błędach literowych czy swoich odczuciach, że jakiś fragment jest niejasny albo stanowczo za trudny.

Kod źródłowy przykładowych programów z tej książki można sobie pobrać ze strony WWW wydawnictwa, a także z mojej strony WWW w Internecie. Obecny adres mojej strony to:

https://www.ifj.edu.pl/private/grebosz/

Nawet jeśli ten adres się kiedyś zmieni, łatwo będzie znaleźć nową lokalizację za pomocą internetowej wyszukiwarki, podając hasło „Opus Magnum C++11" lub moje nazwisko.

Na mojej stronie WWW możesz też szukać odpowiedzi do ćwiczeń, które są zamieszczone na końcu poszczególnych rozdziałów tej książki. Na stronach WWW zawierających materiały uzupełniające do tej książki znajdziesz też ewentualną erratę tworzoną na bieżąco z uwag nadsyłanych przez czytelników.

Uwaga do wydania drugiego

Obecne wydanie **drugie** różni się tym od wydania pierwszego, że do tekstu wprowadzono poprawki zgromadzone dotychczas w erracie do wydania pierwszego.
Możesz ją znaleźć na stronie: https://www.ifj.edu.pl/private/grebosz/opus.html

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

Jeśli jesteś moim dawnym przyjacielem, czytelnikiem moich poprzednich książek, to pewnie zapytasz, dlaczego tak długo musiałeś czekać na tę nową książkę. Poprzednią napisałem przecież ponad 10 lat temu.

Sprawa jest przykra. Pisanie książki to dwa lata pracy. Bycie autorem to rodzaj samotności. Pamiętam, jak wiele propozycji wypraw w góry z przyjaciółmi musiałem odrzucać, bo obowiązek nakazywał mi siedzieć i pracować nad kolejnymi rozdziałami. W końcu jednak praca się kończy, książka idzie do wydawnictwa, a potem do czytelników. To jest dla mnie ogromna radość. Nagle jednak okazuje się, że jakiś mój przyjaciel-czytelnik postanowił „spiratować" książkę i umieścić ją w Internecie. Gdy zwracam się do administratorów danej strony, którzy za pieniądze pozwalają ludziom ściągać tę moją pracę – śmieją mi się w twarz. Usuwają ten link, a za dwa dni znowu on wraca. Znowu zwracam się do nich, a oni znowu się śmieją: „I co nam zrobisz?". W takiej sytuacji nie chce się pisać następnych książek. Też byś tego nie robił.

A jednak napisałem tę, którą trzymasz teraz w dłoniach. Dlaczego? Dlatego że coś się zaczęło zmieniać. Ludzie stali się świadomi, że okradanie polskich autorów sprawia, iż polskich książek jest mniej i są gorszej jakości. Zatem na dłuższą metę tracą inni czytelnicy. Stała się rzecz niezwykła: ostatniej *Symfonii Standard* nie „spiratowano". Na różnych forach internetowych widziałem nawet, jakie gromy rzucano na tych, którzy domagali się pirackich kopii. Może więc nie wszystko stracone? Postanowiłem zaryzykować. Zaryzykować nowe spotkanie z nowymi (i starymi) przyjaciółmi-czytelnikami.

A zatem wyruszamy razem w wielką przygodę, do gwiazd. Proszę zapiąć pasy, bo...

18 Deklaracje przyjaźni

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

18.1 Przyjaciele w życiu i w C++

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

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

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

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

Przełóżmy ten obrazek na język pojęć C++

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

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

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

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

Przykład deklaracji przyjaźni z funkcją:

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

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

Sama funkcja jest gdzieś w programie zdefiniowana następująco:

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

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

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

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

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

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

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

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

Najważniejsza cecha to:

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

Dzięki deklaracji przyjaźni możemy nadać dostęp do prywatnych składników naszej klasy nawet takiej funkcji, która nie mogłaby być funkcją składową naszej klasy z powodów zasadniczych.
Na przykład dlatego, że:
– jest już funkcją składową innej klasy
– albo musi być funkcją globalną (np. przeładowany operator <<).

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

18.2 Przykład: dwie klasy deklarują przyjaźń z tą samą funkcją

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

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

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

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

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

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

Po wykonaniu programu na ekranie zobaczymy:

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

Przyjrzyjmy się ciekawszym miejscom programu

❷ Wewnątrz definicji klasy Tpunkt widzimy deklarację przyjaźni. Klasa Tpunkt stwierdza tutaj, że ma zaufanie do funkcji sędzia.

Bardzo ważna uwaga:

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

Jak ten problem rozwiązać?

❶ Oto rozwiązanie. Jest to tak zwana **deklaracja zapowiadająca** (zwiastująca). Mówi ona: „Jakby co, to nazwa Tkwadrat jest nazwą klasy". To wszystko. Nie ma tu nic więcej na temat wewnętrznej struktury klasy Tkwadrat, ale to nie szkodzi, bo w momencie deklaracji przyjaźni te detale nie są jeszcze kompilatorowi potrzebne.

❸ To deklaracja przyjaźni w drugiej klasie. Podobnie: klasa Tkwadrat stwierdza tutaj, że ma zaufanie do funkcji sędzia.

❹ Oto definicja funkcji sędzia. Jest zdefiniowana jak najzwyklejsza funkcja.

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

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

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

obiekt.funkcja( )

bowiem funkcja sędzia nie jest funkcją składową żadnej klasy.

18.3 W przyjaźni trzeba pamiętać o kilku sprawach

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

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

> *Po prostu w deklaracji przyjaźni tylko rozmawiamy z kompilatorem, tłumacząc mu, że taka funkcja ma dostęp...*

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

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

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

Powtarzam więc wniosek:

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

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

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

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

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

Jak to zrobić w naszym niedawnym programie?

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

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

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

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

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

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

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

Taka definicja funkcji zaprzyjaźnionej ma następujące konsekwencje:

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

a) skorzystać z właśnie obowiązujących (wewnątrz deklaracji tej klasy) instrukcji using lub typedef,

b) skorzystać ze zdefiniowanych w tej klasie typów wyliczeniowych enum.

Nie zawsze można takim chwytem się posłużyć. Aby to było możliwe:

- Klasa, w której ma się to zdarzyć, nie może być tak zwaną *klasą lokalną*.

  Dotychczasowe nasze klasy były nielokalne (czyli zwykłe), tutaj więc nie ma przeszkód. (O klasach lokalnych porozmawiamy na str. 734).

- Tak definiowana funkcja zaprzyjaźniona nie może mieć nazwy z kwalifikatorem zakresu twierdzącej, że jest ona jakoby funkcją składową jakiejś innej klasy, podczas gdy naprawdę takiej funkcji składowej w tamtej klasie nie ma.

  To by były wręcz podchody! Wyobraź sobie: ktoś zdefiniował klasę i jest z niej dumny, a my – za pomocą deklaracji przyjaźni w naszej klasie – cichaczem dodefiniowujemy tej jego klasie piętnaście dodatkowych funkcji składowych!

Jeśli skorzystamy z tej możliwości, aby w klasie K, przy okazji deklaracji przyjaźni ze zwykłą funkcją f, od razu umieścić także definicję tej funkcji zaprzyjaźnionej f, to pamiętajmy, że wtedy ta funkcja nie jest widoczna na zewnątrz klasy K. Aby tak było, trzeba ją wcześniej deklarować:

```
void rezydent();                                                    ❶
////////////////////////////////////////////////////////////////////////////////
class K
{
    friend void rezydent()   { cout << „Jestem rezydent funkcja” << endl; }
};
////////////////////////////////////////////////////////////////////////////////
int main()
{
    rezydent();                                                     ❷
}
```

Zatem bez deklaracji ❶ kompilator w miejscu ❷ wykryje błąd, mówiąc, że nazwa rezydent nie jest mu znana. Wniosek: w każdej sytuacji najlepiej, żeby funkcja, którą klasa ma obdarzyć przyjaźnią, była już wcześniej zadeklarowana.

Przy przeładowaniu

W przypadku funkcji przeładowanych przyjacielem klasy K jest tylko ta wersja funkcji, która odpowiada liście argumentów widocznej w deklaracji przyjaźni w definicji danej klasy K.

```
class K
{
    //...
    friend void alarm(K obj, int k);
};

void alarm(float*, K obiekt);
void alarm(void);
void alarm(K obiekt, int i);                    //    to jest przyjaciel klasy K
```

18.4 Obdarzenie przyjaźnią funkcji składowej innej klasy

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

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

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

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

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

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

Po wykonaniu programu na ekranie pojawi się:

```
pilka lezy na tle boiska
```

Komentarz

❷ To jest deklaracja zwykłej funkcji składowej w klasie Tkwadrat. Argumentem jest obiekt klasy Tpunkt, stąd też konieczna była deklaracja zapowiadająca tę klasę ❶.

❸ To deklaracja przyjaźni. Tutaj jednak ważna uwaga. Jeśli przyjacielem ma być *funkcja składowa z innej klasy*, to ta klasa musi być już w tym momencie znana kompilatorowi. Dlatego najpierw w programie umieszczona jest deklaracja klasy Tkwadrat (z funkcją sędzia), a potem dopiero deklaracja klasy Tpunkt i niniejsze ogłoszenie przyjaźni.

❹ Oto definicja funkcji sędzia. Na pewno już przy deklaracjach zauważyłeś, że zmieniła się lista argumentów. Teraz argumentem jest tylko obiekt klasy Tpunkt. A co z obiektem klasy Tkwadrat, funkcja go przecież także potrzebuje?! Zapominasz, że teraz funkcja jest funkcją składową klasy Tkwadrat, a więc jest wywoływana na rzecz obiektu klasy Tkwadrat. Zresztą spójrz poniżej.

❻ Tak właśnie w main wywołujemy funkcję sędzia. Obiekt klasy Tpunkt wysyłany jest jako argument, a obiekt klasy Tkwadrat – przez ukryty wskaźnik this.

❺ Z faktu, iż funkcja sędzia jest funkcją składową klasy Tkwadrat, wynika, że odnosząc się do danej składowej swojej klasy, można posługiwać się zapisem: *składnik*, a nie zapisem: *obiekt*.*składnik* – widać to wyraźnie w tej linijce.
W stosunku do składników obiektu klasy zaprzyjaźnionej Tpunkt stosujemy tu zapis pt.x, ale w stosunku do składników swojej klasy: x, bok. Przedtem w tym miejscu było konieczne kw.x, kw.bok.

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

> *„Coś kręcisz!”* – zawołałeś zapewne – *„parę stron wcześniej wmawiałeś mi, że funkcja zaprzyjaźniona z klasą nie zawiera wskaźnika this!”.*

Podtrzymuję to!

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

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

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

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

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

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

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

kwiatki *czyli* this–>kwiatki

gdzie this oznacza wskaźnik do obiektu „mój_dom”.

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

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

Który z wariantów wybrać, decydujesz, rozważając wspomniane zalety funkcji zaprzyjaźnionej z cechami funkcji składowej. O podejmowaniu takich wyborów porozmawiamy jeszcze w jednym z następnych rozdziałów. („Przeładowanie operatorów” – str. 954).

18.5 Klasy zaprzyjaźnione

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

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

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

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

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

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

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

✧ Mówiliśmy o tym, że definicję *funkcji* zaprzyjaźnionej z klasą K można umieścić nawet w samym miejscu deklaracji przyjaźni, czyli wewnątrz klasy K (funkcja: przyjaciel – rezydent, str. 699).

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

Deklaracja przyjaźni jest oczywiście jednostronna

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

Dwie klasy mogą się przyjaźnić także z wzajemnością

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

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

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

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

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

```
class Tdruga;                    //    deklaracja zapowiadająca

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

```
};

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

Przyjaźń nie jest przechodnia

> Przyjaciel mojego przyjaciela nie jest moim przyjacielem.

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

Gdyby o to chodziło, to należałoby w klasie A zamieścić deklarację takiej przyjaźni:

```
friend class C;
```

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

Przyjaźń nie jest dziedziczna

> Przyjaciel mojej prababki nie jest moim przyjacielem.
>
> *Wtajemniczeni wiedzą, że klasa może mieć „potomstwo" (klasy pochodne). Przyjaźń nie jest dziedziczna – jeśli jakaś klasa chce mieć przyjaciela, to powinna to powiedzieć wyraźnie sama.*

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

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

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

18.6 Konwencja umieszczania deklaracji przyjaźni w klasie

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

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

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

18.7 Kilka otrzeźwiających słów na zakończenie

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

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

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

Naprawdę więc deklaracje przyjaźni będziemy stosowali głównie w sytuacjach:

- gdy będziemy chcieli, aby obiekt cout mógł wypisywać na ekranie treść składników obiektu naszej klasy (obdarzymy wtedy przyjaźnią klasę ostream),
- gdy będziemy chcieli, żeby na obiektach naszej klasy mogły sprawnie pracować globalne funkcje tzw. operatorowe (poznamy je na str. 954).

18.8 Ćwiczenia

I Przyjaźń polega na tym, że dana klasa K udziela zezwolenia innej funkcji/klasie:
a) na dostęp do pracy z obiektami klasy K,
b) na modyfikację jej składników w klasie K,
c) na dostęp do składników niepublicznych w obiektach klasy K,
d) na dostęp do składników niepublicznych w wyznaczonych obiektach klasy K.

II Klasa K oznajmia przyjaźń z funkcją/klasą P. Słowo friend umieszczone jest:
a) w deklaracji funkcji/klasy P uznanej za przyjaciela,
b) w klasie K, która ogłasza przyjaźń z funkcją/klasą P,
c) w klasie/funkcji P, w instrukcjach, które odnoszą się do prywatnych składników klasy K.

III Przyjaźń może deklarować:
a) klasa, b) funkcja globalna, c) funkcja składowa, d) każde z wymienionych.

IV Co może być przyjacielem?
a) funkcja globalna, c) funkcja składowa,
b) funkcja statyczna, d) klasa.

V Jedna funkcja może być przyjacielem
a) tylko jednej klasy, b) nawet wielu klas.

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

VII Jeśli funkcja-przyjaciel klasy K odnosi się do składników klasy K, to skąd wiadomo, którego obiektu klasy K to dotyczy?
a) działa to na wszystkie obiekty klasy K,
b) funkcja musi powiedzieć, o który konkretny obiekt jej chodzi.

VIII Przyjaciel klasy K otrzymuje prawo dostępu do składników niepublicznych klasy K
a) we wszystkich obiektach klasy K,
b) w wybranym obiekcie klasy K.

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

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

XI Klasa K deklaruje przyjaźń z funkcją składową klasy P. Jaka deklaracja klasy K musi poprzedzać definicję tej klasy?

a) wystarczy deklaracja zapowiadająca K, b) konieczna jest definicja klasy K.

XII Wybierz wszystkie poprawne zakończenia następującego zdania: Klasa K deklaruje przyjaźń z funkcją składową klasy P; ta funkcja składowa:

a) jest wywoływana na rzecz obiektu klasy K, która obdarzyła ją przyjaźnią,
b) jest wywoływana na rzecz obiektu swojej klasy P,
c) może pracować na składnikach swego obiektu klasy P,
d) może pracować na obiekcie klasy K pod warunkiem, że wie na którym.

XIII Klasa K deklaruje przyjaźń z klasą P. Czy funkcja składowa klasy P otrzymuje wskaźnik this pozwalający jej pracować na składnikach klasy K?

XIV Co to znaczy, że klasa K deklaruje przyjaźń z klasą P?

XV Co to znaczy, że przyjaźń nie jest przechodnia?

XVI Czy przyjaźń jest wzajemna?

XVII Gdzie należy umieszczać deklarację przyjaźni w klasie K. W jej części public czy private?

XVIII Funkcja składowa klasy P ma przydomek static. W klasie K chcemy zamieścić deklarację przyjaźni z tą funkcją. Gdzie umieszczamy ten przydomek?

a) po słowie friend, a przed typem rezultatu,
b) na samym końcu deklaracji.

Skorowidz

!

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

OPUS MAGNUM C++11

Programowanie w języku C++

Dzięki tej książce poznasz:

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

Jeżeli chcesz nauczyć się tego języka w łatwy, pogodny, przyjazny sposób, ta książka jest właśnie dla Ciebie.

Jedno C i same plusy!

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

Autor tej książki – wybitny specjalista pracujący nad wieloma znaczącymi projektami we francuskich, niemieckich i włoskich instytutach fizyki jądrowej, znany czytelnikom m.in. z genialnej *Symfonii C++* – postawił sobie za cel napisanie nowej, przekrojowej publikacji o tym języku, która w prostym, obrazowym stylu wprowadza czytelnika w fascynujący świat programowania zorientowanego obiektowo. Zobacz, jak potężny jest **C++ 11**.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-6965-8