Podstawy języka C++

Skrypt akademicki

Paweł Kłeczek

pkleczek@agh.edu.pl
home.agh.edu.pl/~pkleczek

v1.0.9
(2021-05-21)

Spis treści

Rozdział 1

Wprowadzenie do języka C++

C++ to język programowania ogólnego przeznaczenia, zaprojektowany przez Bjarne'a Stroustrupa jako rozszerzenie języka C, wspierający mechanizmy obiektowości i abstrakcji danych oraz statycznej kontroli typów[1].

C++ to język bardzo rozwinięty pod względem dostępnych operatorów i prostoty notacji, co pozwala na wygodne tworzenie abstrakcji danych oraz stosowanie różnych paradygmatów programowania: proceduralnego, zorientowanego obiektowo, oraz generycznego. Wyróżnia się dużą wydajnością kodu obiektowego, możliwością bezpośredniego dostępu do zasobów sprzętowych i funkcji systemowych, łatwością tworzenia i korzystania z bibliotek (napisanych w językach C++, C lub innych), niezależnością od konkretnych uwarunkowań związanych ze sprzętem bądź systemem operacyjnym (co sprawia, że kod napisany w języku C++ jest bardzo przenośny) oraz niewielkim środowiskiem uruchomieniowym.

Język C++ stosowany jest głównie do pisania wysokopoziomowych aplikacji (np. wykorzystujących interfejs graficzny bądź komunikację sieciową) oraz systemów operacyjnych.

1.1 Programowanie zorientowane na obiekty

Ten rozdział ma dać Ci ogólne pojęcie, pewną intuicję, na czym polega programowanie obiektowe. Część użytych pojęć i mechanizmów zostanie omówiona dopiero w rozdziałach 2 i 8.

Często można spotkać stwierdzenie, że „język C++ to język obiektowy" (często: w opozycji do „strukturalnego języka C") – co to twierdzenie oznacza oraz czy jest ono prawdziwe? Nim odpowiemy na to pytanie warto zapoznać się z podstawowymi pojęciami związanymi z **programowaniem zorientowanym na obiekty** (ang. object-oriented programming, OOP), zwanym też **programowaniem zorientowanym obiektowo**. Pojęcia te nie są związane z konkretnym językiem programowania, lecz z pewną ogólną koncepcją podejścia do programowania. Co ważne, nie istnieją ścisłe definicje żadnego z tych pojęć; większość z nich to terminy bardzo ogólne (często nie ograniczające się wyłącznie do OOP), jednak w poniższym zestawieniu skupiono się na kontekście OOP oraz (w wybranych przypadkach) na języku C++. Wspomniane pojęcia to:

- **Obiekt** (ang. object) to moduł, który wiąże (grupuje) dane z procedurami, które operują na tych danych.
- **Klasa** (ang. class) to swego rodzaju „przepis" określający sposób tworzenia i korzystania z obiektów danego rodzaju (tj. ogólne zachowanie każdego obiektu tego rodzaju) – wszystkie obiekty danej klasy posiadają ten sam identyczny zbiór *cech* i *zachowań* (przy czym konkretne *wartości* cech mogą różnić się pomiędzy tymi obiektami). Innymi słowy, klasy umożliwiają definiowanie **abstrakcyjnych typów danych** (ang. abstract data type, ADT).
  Przykładowo, w realnym świecie spotykamy się z różnymi egzemplarzami dmuchanych piłek do gry – są piłki wykonane z tkaniny albo z tworzyw sztucznych, piłki duże i małe, jednokolorowe albo wielobarwne itd. Mimo to wszystkie te egzemplarze posiadają identyczny zbiór *cech* (np. materiał wykonania, rozmiar, kolor) oraz *zachowań* (np. każdą dmuchaną piłkę można napompować). Możemy więc powiedzieć, że egzemplarze te są *obiektami* (konkretnymi przykładami) pewnej *klasy* (pewnej *abstrakcji*) – dmuchanej piłki do gry – posiadającej wspomniane cechy oraz definiującej wspomniane zachowania.
- **Dziedziczenie** (ang. inheritance) to zdolność do tworzenia nowych obiektów (lub klas) z wykorzystaniem istniejących, przy czym większość cech i zachowań obiektu macierzystego (lub klasy macierzy-

[1] Statyczność kontroli typów oznacza, że wykonywana jest ona jeszcze na etapie kompilacji programu.

stej) jest zachowywana również w obiekcie pochodnym (lub klasie pochodnej) bez konieczności ich ponownego definiowania – programista skupia się na określeniu *różnic* między bytem macierzystym, a bytem pochodnym.

- **Enkapsulacja** (ang. encapsulation) to pewna idea, zgodnie z którą kod operujący na obiekcie nie powinien mieć swobodnego dostępu do szczegółów implementacyjnych takiego obiektu – dostęp do danych i operacji jest możliwy wyłącznie poprzez publiczny **interfejs programistyczny aplikacji** (ang. application programming interface, API) danego obiektu, czyli przez zbiór jego publicznych operacji i publicznych danych[2]. Innymi słowy enkapsulacja to zdolność do zagwarantowania, że obiekt będzie używany wyłącznie zgodnie z jego specyfikacją – jest ona kluczowa do zabezpieczenia obiektu przed „uszkodzeniem" (np. naruszeniem niezmienników).
  Przykładowo, abstrakcja wyrażająca prawdopodobieństwo powinna przyjmować wartości wyłącznie z zakresu $\langle 0, 1 \rangle$ – nadanie wartości spoza tego zakresu będzie naruszeniem niezmiennika prawdopodobieństwa.
  Co więcej, dzięki zastosowaniu enkapsulacji programista ma większą swobodę w modyfikowaniu szczegółów implementacyjnych obiektów w sposób niezauważalny dla użytkowników tych obiektów – bez konieczności zmiany kodu korzystającego z obiektów (poza obiektem widoczne są jedynie zmiany w jego *publicznym* API).
- **Polimorfizm** (ang. polymorphism) oznacza możliwość wykorzystania tego samego API dla obiektów różnych typów z użyciem tego samego symbolu. Szczególnym rodzajem polimorfizmu jest **podtypowanie** (ang. subtyping), w przypadku którego kod korzystający z obiektu pewnej klasy powiązanej hierarchią dziedziczenia może być wykonany bez potrzeby znajomości tego, czy jest to obiekt klasy macierzystej czy jednej z jej klas potomnych – nawet wówczas, gdy implementacje tak samo nazwanej operacji różnią się pomiędzy klasami w hierarchii.
  Przykładowo, w przypadku hierarchii klas opisujących figury geometryczne każda z klas powinna posiadać operację służącą do uzyskania pola figury o identycznym interfejsie (np. nie przyjmuje argumentów i zwraca wartość zmiennoprzecinkową), przy czym z oczywistych względów implementacja tej operacji będzie się różniła pomiędzy konkretnymi klasami kształtów. Niemniej kod korzystający z obiektu *pewnej* klasy figury geometrycznej powinien być w stanie uzyskać pole powierzchni konkretnej figury opisywanej przez ten obiekt bez znajomości tego, jakiej dokładnie klasy jest to obiekt.

W środowisku osób związanych z programowaniem nie ma zgody co do definicji programowania zorientowanego na obiekty:

- Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional, 1994)[3]:
  „Programy zorientowane na obiekty składają się z obiektów. Obiekt wiąże dane z procedurami, które operują na tych danych. (. . .)"
- Peter Wegner (1987):
  „OOP = obiekty + klasy + dziedziczenie"
- Encyclopædia Britannica:
  „Programowanie zorientowane na obiekty – użycie predefiniowanych modularnych jednostek programistycznych (obiektów, klas, podklas itd.) w celu uczynienia programu szybszym i łatwiejszym w zarządzaniu. Języki wspierające programowanie zorientowane na obiekty pozwalają zarządzać złożonością w dużych programach. Obiekty «opakowują»dane i operacje na tych danych w taki sposób, że jedynie operacje są publiczne, podczas gdy szczegóły (implementacyjne) struktur danych pozostają ukryte. To przesłanianie informacji uprościło wielkoskalowe programowanie poprzez umożliwienie programiście myślenia o każdym fragmencie programu w oderwaniu od pozostałych. Dodatkowo, obiekty mogą być wywodzone z ich bardziej ogólnych form, «dziedzicząc»ich możliwości. Taka hierarchia obiektów umożliwia definiowanie wyspecjalizowanych obiektów bez konieczności ponownego definiowania tego, co zostało już zawarte w obiektach bardziej ogólnych."

Niemniej elementem wspólnym każdej z definicji jest pewien elementarny sposób postrzegania programu komputerowego – nie jako zbioru *działań* służących do manipulowania danymi (obiektami), ale jako zbioru

[2] Samo pojęcie API nie odnosi się wyłącznie do zagadnienia enkapsulacji – jest ono szersze i ogólnie oznacza interfejs lub protokół komunikacji pomiędzy różnymi fragmentami programu określony w celu uproszczenia implementacji i utrzymania oprogramowania. Specyfikacja API może się zatem obejmować specyfikację procedur, struktur danych, klas, zmiennych itp.

[3] Książka ta potocznie nazywana jest „The Gang of Four" i stanowi kompendium wzorców projektowania zorientowanego na obiekty

obiektów, których stan można zmieniać z użyciem pewnych działań. Innymi słowy, w przypadku programowania zorientowanego na obiekty w centrum zainteresowania programisty są obiekty, a nie procedury – stąd mówimy o **paradygmacie obiektowym** (ang. object-oriented paradigm, OO paradigm). To przeniesienie uwagi z procedur na obiekty jest niezmiernie istotne, gdyż umożliwiło tworzenie złożonych systemów informatycznych. W dawnych czasach królowało proceduralne podejście do programowania[4]: dane w programie były tylko luźno powiązane z funkcjami operującymi na tych danych, podobnie luźne były powiązania między funkcjami. Takie podejście sprawiało, że w złożonych systemach (w których mogły występować setki funkcji i zmiennych) pojawiał się problem małej czytelności kodu, zwiększonego ryzyka błędów (przypadkowej modyfikacji danych), oraz trudności w efektywnym rozszerzaniu istniejącej funkcjonalności.

Podobnie jak w przypadku definicji OOP, w środowisku osób związanych z programowaniem nie ma zgody co do warunków, które dany język programowania musi spełniać, aby zostać uznanym za **język zorientowany na obiekty** (ang. object-oriented language, OOL), czyli potocznie za „język obiektowy" – w najbardziej radykalnym ujęciu w przypadku OOL „wszystko jest obiektem", a dodatkowo (w zależności od definicji) powinien on umożliwiać enkapsulację i dziedziczenie. Według Bjarne'a Stroustrupa język (lub technika) może zostać uznany za „zorientowany obiektowo" tylko wówczas, gdy bezpośrednio wspiera[5]:

- **abstrakcję** – zapewnia pewną formę realizacji idei klas i obiektów,
- **dziedziczenie** – możliwość wyprowadzania nowych abstrakcji z abstrakcji już istniejących, oraz
- **dynamiczny polimorfizm** – pewną formę wiązania w czasie wykonania programu.

Sama idea programowania zorientowanego na obiekty swoimi korzeniami sięga języka Simula 67, natomiast została w pełni rozwinięta w języku Smalltalk 80 – zaprojektowanym jako język „czysto obiektowy", o jednolitej składni. Prawdziwa powszechność i sukces komercyjny idei OOP przyszły jednak dopiero wraz z językami C++ i (później) Java. W praktyce jednak wiele współczesnych popularnych języków programowania (np. C++, Java) nie tyle *jest* obiektowych, co *wspiera OOP w większym lub mniejszym stopniu* (w zależności od przyjętej definicji OOP). Przykładowo:

- **„Wszystko jest obiektem"**. W czystym modelu obliczeń OOP wszystkie typy są klasami, a wszystkie obliczenia realizowane są poprzez przekazywanie komunikatów (czyli wywołania metod). W językach C++ i Java typy wbudowane (np. `int`, `double`) nie są obiektami w rozumieniu OOP (operacje na typach wbudowanych nie są zdefiniowane przez te typy, lecz „zaszyte" w zasadach języka programowania). Z kolei w języku Python faktycznie każdy typ – nawet tzw. typy proste (np. `int`, `float`) – jest klasą, a przykładowo zastosowanie operatora dodawania dla dwóch obiektów typu prostego skutkuje wywołaniem stosownej metody pierwszego z nich (czyli `a + b` jest tłumaczone na `a.__add__(b)`.
- **Enkapsulacja.** Język C++ wspiera enkapsulację (za pomocą takich elementów języka, jak: specyfikatory dostępu, funkcje zaprzyjaźnione, klasy zaprzyjaźnione), natomiast Python – nie (w języku Python wszystkie dane i operacje określone przez obiekt są wyłącznie publiczne).

Mimo to wszystkie języki programowania wspierające OOP pozwalają m.in. na:

- Stosowanie podejścia *bottom-up* do projektowania programu. W podejściu tym najpierw należy się skupić na dokładnym zaprojektowaniu podstawowych „klocków", które następnie łączy się w bardziej złożone konstrukcje. (W podejściu „top-bottom", stosowanym tradycyjnie w programowaniu proceduralnym, najpierw projektuje się procedurę główną – w której wymienia się inne procedury niezbędne do jej działania – a dopiero potem przystępuje się do projektowania poszczególnych podprocedur.)
- Ponowne wykorzystanie kodu, w szczególności możliwość definiowania nowych klas w oparciu o istniejące klasy.

1.2 Cechy szczególne języka C++

Oto najważniejsze wyróżniki języka C++:

- Język C++ jest **wieloparadygmatowym językiem programowania** (ang. multiparadigm programming language), lub inaczej **językiem hybrydowym** (ang. hybrid language) – można w nim stosować jednocześnie różne paradygmaty programowania (m.in. programowanie proceduralne, obiektowe i generyczne), a także programować na poziomie asemblera. Co istotne, język C++ *wspiera* podejście obiektowe, ale go *nie wymusza*.
- Język C++ umożliwia bezpośrednie zarządzanie wolną pamięcią.

[4] Różnica między podejściem proceduralnym, strukturalnym i obiektowym została przystępnie omówiona na stronie difference-between.info oraz w tym poście.

[5] zob. Why C++ is not just an Object-Oriented Programming Language

- Projekt języka zakłada, że żadna nowa (względem języka C) cecha języka C++ nie może mieć negatywnego wpływu na szybkość działania programu lub zapotrzebowanie na pamięć operacyjną. Dzięki temu dobrze napisany program w C++ jest z reguły co najmniej równie szybki (a czasem wręcz szybszy), jak jego odpowiednik napisany w C.
- Język C++ zakłada statyczną kontrolę typów, natomiast posiada też elementy dynamicznej kontroli typów[6].
- Język C++ umożliwia stosowanie RAII – techniki programistycznej, która wiąże przejęcie i zwolnienie zasobu z inicjowaniem i usuwaniem zmiennych[7].

Warto wspomnieć, że język C++ posiada również bogatą bibliotekę standardową – dzięki czemu programiści nie muszą tworzyć wielu podstawowych funkcjonalności „od zera" – jednak nie jest to jego cecha szczególna.

1.2.1 Style programowania wspierane przez język C++

Poniższy rozdział opiera się w pełni na artykule Bjarne'a Stroustrupa Why C++ is not just an Object-Oriented Programming Language

Język C++ został celowo zaprojektowany w taki sposób, aby wpierał te style, które zasadniczo[8] są dobre i użyteczne (to, czy są one zgodne z koncepcją obiektowości, nie miało większego znaczenia)[9]:

1. Abstrakcja – czyli możliwość bezpośredniej reprezentacji pojęć bezpośrednio w programie oraz ukrywanie nieistotnych szczegółów (implementacyjnych) za dobrze zdefiniowanym interfejsem – jest kluczowa dla każdego elastycznego i czytelnego systemu, niezależnie od jego złożoności.
2. Enkapsulacja – czyli zdolność do zagwarantowania, że abstrakcja będzie używana wyłącznie zgodnie ze specyfikacją – jest kluczowa do zabezpieczenia abstrakcji przed „uszkodzeniem".
3. Polimorfizm – czyli zdolność zapewnienia tych samych interfejsów dla obiektów zawierających różne ich implementacje – jest kluczowa do uproszczenia kodu korzystającego z abstrakcji.
4. Dziedziczenie – czyli zdolność do tworzenia nowych abstrakcji z wykorzystaniem istniejących – to jeden z najpotężniejszych środków do tworzenia użytecznych abstrakcji.
5. Generyczność – czyli zdolność do parametryzacji typów i funkcji za pomocą innych typów (oraz wartości) – jest niezbędna do wyrażania kontenerów bezpiecznych pod względem typowania oraz do wyrażania ogólnych algorytmów.
6. Współistnienie z innymi językami i systemami – to kluczowa cecha języka, niezbędna aby program w nim napisany mógł funkcjonować w każdym „realnym" środowisku uruchomieniowym[10].
7. Zwięzłość i szybkość działania – są kluczowe dla każdego języka służącego do programowania systemowego.
8. Statyczna kontrola typów – to integralna właściwość rodziny języków programowania, do której należy C++; cenna ze względu na udzielone gwarancje projektowe oraz zapewnienie wydajności w kwestii czasu wykonania i niezbędnych zasobów pamięciowych.

1.3 Standardy języka C++

Ze słownika SJP PWN: *standaryzacja* «wprowadzenie jednolitych norm, zwłaszcza w przemyśle»

Standard języka programowania określa, jak działa taki język – jakie konstrukcje są dopuszczalne i jaki będzie efekt ich wykonania.

---

[6] Dynamiczna kontrola typu polega na jego kontroli w trakcie wykonywania programu.

[7] Mechanizm RAII został omówiony w rozdziale 9.6 RAII i cykl życia obiektów

[8] wg Bjarne'a Stroustrupa

[9] Przytoczone style oraz kluczowe konstrukcje językowe zostały dokładniej podsumowane w artykule Bjarne'a Stroustrupa Why C++ is not just an Object-Oriented Programming Language.

[10] Wymóg współistnienia z innymi systemami jest kluczowy dla każdego języka, który chce uchodzić za **język ogólnego przeznaczenia** (ang. general-purpose programming language) – niemal każdy realny system zawiera bowiem pewne elementy napisane w innych językach programowania i zaprojektowane zgodnie z zasadami obcymi dla „głównego" języka. Aby język był faktycznie ogólnego przeznaczenia, musi zapewniać możliwość współdzielenia danych z fragmentami programu napisanych w innych językach, musi pozwalać na wykonywanie fragmentów kodu napisanych w innych językach, oraz na wykonywanie swojego kodu przez kod napisany w innych językach.

Język C++ jest standaryzowany przez Międzynarodową Organizację Normalizacyjną (ISO), przy czym najnowsza wersja standardu została ratyfikowana i opublikowana w grudniu 2017 r. jako dokument ISO/IEC 14882:2017 (potocznie znany jako standard C++17).

Oto krótki rys historyczny standaryzacji języka C++[11]:

- W 1998 r. grupa robocza ISO opracowała pierwszy standard C++, nazywany nieformalnie C++98.
- W 2003 r. został opracowany tzw. standard C++03, który usuwał błędy standardu C++98.
- Duże zmiany przyniósł dopiero tzw. standard C++11, upubliczniony w 2011 r., wprowadzający nowe funkcjonalności do jądra języka i do biblioteki standardowej.
- W 2014 r. upubliczniono tzw. standard C++14, mający na celu usunięcie błędów w standardzie C++11 oraz wprowadzenie drobnych poprawek.
- W 2017 r. opublikowano tzw. standard C++17, który wprowadza liczne zmiany zarówno do składni języka, jak i do biblioteki standardowej.

Ważne

Mnogość standardów sprawia, że należy zachować czujność korzystając ze starych podręczników do C++ oraz materiałów dostępnych w Internecie – często prezentują one rozwiązania, które w obecnych czasach świadczą o nieznajomości współczesnego języka C++ oraz o złym stylu programistycznym!

Ponieważ przenoszenie projektów na nowe standardy wymaga nakładu pracy, wiele firm (zwłaszcza z branż, gdzie krytyczna jest niezawodność już napisanego kodu: telekomunikacja, bankowość) wciąż pracuje w oparciu o standard C++98 bądź C++03. Zazwyczaj mija też pewien okres od chwili opublikowania standardu do chwili, gdy zaproponowane rozwiązania uzyskają stabilne wsparcie ze strony poszczególnych kompilatorów języka C++.

1.4 Filozofia języka C++

W całym okresie istnienia języka C++ kierunki jego rozwoju były wyznaczone (i ograniczone) następującymi zasadami, opisanymi przez Bjarne'a Stroustrupa w dokumencie „Evolving a language in and for the real world: C++ 1991-2006", m.in.:

- Rozwój musi się skupiać na rozwiązywaniu „rzeczywistych" problemów, a zaproponowane nowe funkcjonalności języka powinny być możliwe do bezpośredniego użycia w programach rozwiązujących te problemy.
- Każda funkcjonalność powinna być możliwa do zaimplementowania (w sposób możliwie oczywisty).
- Programiści powinni mieć swobodę wyboru swojego stylu programowania, a styl ten powinien być w pełni wspierany przez język C++.
- Umożliwienie skorzystania z pewnych funkcjonalności jest ważniejsze niż zapobieganie każdemu możliwemu sposobowi niewłaściwego użycia języka C++.
- Niejawne naruszanie systemu typów jest zabronione, natomiast dopuszczalne jest jego jawne naruszanie (tj. wówczas, gdy programista wyraźnie sobie tego zażyczy stosując odpowiedni operator rzutowania).
- Typy zdefiniowane przez użytkownika powinny mieć takie samo wsparcie i wydajność, co typy wbudowane.
- Funkcjonalności języka niewykorzystywane w kodzie danego programu nie powinny negatywnie wpływać na utworzone pliki wykonywalne (np. w postaci niższej wydajności).
- Język C++ powinien koegzystować z innymi istniejącymi językami programowania, a nie rozwijać swoje własne, niekompatybilne z innymi językami środowisko programistyczne.

1.5 Język C a język C++

W dokumencie „Rationale for International Standard Programming Languages C", zawierającym uzasadnienie decyzji podjętych przy projektowaniu standardu C99, można przeczytać, że:
„[Celem standardu C99 jest] minimalizacja niekompatybilności z językiem C++. (. . .) Komitet [standaryzacyjny] popiera zasadę zachowywania możliwie dużego wspólnego zbioru [funkcjonalności] (. . .). Taka zasada powinna umożliwić maksymalizację cech wspólnych obu języków przy jednoczesnym zachowaniu

[11] Więcej o standardach języka C++ przeczytasz na stronie C++ Standardization (wiki).

rozróżnienia między nimi oraz umożliwieniu ich niezależnego rozwoju. (. . .) Choć niektóre cechy języka C++ mogą być zawarte również w języku C, **intencją Komitetu nie jest, aby język C stał się językiem C++**.

W związku z tym poniższy (prosty) program napisany w języku C:

```c
#include <stdio.h>
#include <stdlib.h>

#define N_MONTHS 12

int main(void) {
  int DAYS_IN_MONTHS[N_MONTHS] = {
      31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
  };
  int i;
  double* ptr = (double*) NULL;

  for (i = 0; i < N_MONTHS; i++) {
    printf("Month #%d has %d days.\n", (i+1), DAYS_IN_MONTHS[i]);
  }
  printf("ptr: %p\n", (void*) ptr);

  return EXIT_SUCCESS;
}
```

będzie jednocześnie poprawnym (pod względem składni i działania) programem w języku C++, jednak taka zgodność nie musi występować w przypadku każdego programu napisanego w języku C (zwłaszcza, gdy korzysta on z zaawansowanych konstrukcji i mechanizmów języka).

Analogiczny program napisany z wykorzystaniem niektórych[12] elementów języka C++ wygląda następująco:

```c
#include <cstdlib>
// Funkcjonalność we/wy jest zawarta w pliku nagłówkowym `iostream`
#include <iostream>

int main() {
    // Użycie const do definiowania stałych (w C służyły do tego dyrektywy
    // preprocesora; w C++ const stanowi fragment typu).
    const int N_MONTHS = 12;
    const int DAYS_IN_MONTHS[N_MONTHS] = {
            31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
    };

    // Deklaracja zmiennych w (niemal) dowolnym miejscu
    // w kodzie - na przykładzie pętli.
    for (int i = 0; i < N_MONTHS; i++) {
        // Wypisywanie na konsolę z użyciem strumienia `std::cout`.
        std::cout << "Month #" << (i+1) << " has "
                  << DAYS_IN_MONTHS[i] << " days." << std::endl;
    }

    // Nowy literał `nullptr` jako wskaźnik pusty (zamiast dyrektywy NULL).
    double* ptr = nullptr;
    std::cout << "ptr: " << ptr << std::endl;

    return EXIT_SUCCESS;
}
```

[12] W dalszej części podręcznika poznasz m.in. kontener `std::array` służący do trzymania ciągu elementów o ustalonej długości, oraz konstrukcję *range-based for* umożliwiającą wygodne iterowanie po elementach tablicy.

Na chwilę obecną najważniejszą różnicą dla Ciebie będzie wypisywanie z użyciem obiektu `std::cout`, oznaczającym standardowe „znakowe urządzenie wyjściowe” – zwykle konsolę komputera[13]. W powyższym przykładzie operator << służy do umieszczania danych w obiekcie `std::cout`.

Jeśli jednak na razie wolisz korzystać ze znanej Ci z języka C funkcji `printf()` – możesz tak robić po dołączeniu pliku nagłówkowego `<cstdio>`[14].

W wielu programach (zwłaszcza w poradnikach internetowych) możesz spotkać tuż poniżej dyrektyw `#include` instrukcję

```
using namespace std;  // zła praktyka programistyczna!!
```

która sprawia, że nie musisz poprzedzać nazw klas, obiektów i funkcji z biblioteki standardowej przedrostkiem „`std::`” – jest to jednak zła praktyka programistyczna[15].

1.5.1 Różnice między językami C i C++

W tabeli 1.1 zestawiono podstawowe różnice między językami C i C++ dotyczące podejścia do programowania oraz pewnych kluczowych mechanizmów.

Pod względem składni język C++ *w przybliżeniu* stanowi *nadzbiór* języka C – oznacza to, że z grubsza wszystkie elementy języka znane Ci z języka C możesz zastosować również w języku C++, natomiast język C++ daje również liczne nowe możliwości (których nie posiada język C). Istnieje jednak kilka wyjątków od tej reguły – w kolejnych rozdziałach omówione zostaną te z nich, które dotyczą poznanych przez Ciebie zagadnień języka C (wyjątki te zostaną przedstawione w subiektywnym porządku od najbardziej do najmniej dla Ciebie istotnych).

Rozszerzenia nazw plików źródłowych i nagłówkowych

Język C++ nie precyzuje sposobu nazywania plików źródłowych i nagłówkowych, nie obowiązuje również w tej kwestii spójna konwencja. Jednak pisząc program w języku C++ *nie należy* stosować rozszerzeń „`.c`” dla plików źródłowych i „`.h`” dla plików nagłówkowych, tylko np. odpowiednio „`.cpp`” i „`.hpp`”, gdyż przykładowo to na podstawie rozszerzenia pliku źródłowego narzędzia w stylu CMake określają, czy skompilować go z użyciem kompilatora języka C czy C++.

Nagłówki biblioteki standardowej

W przypadku niektórych plików nagłówkowych biblioteki standardowej języka C, o nazwach postaci `xxx.h`, biblioteka standardowa języka C++ udostępnia ich odpowiedniki o nazwach postaci `cxxx` (tj. z przedrostkiem „`c`” i bez rozszerzenia „`.h`”). Poza nielicznymi wyjątkami, każdy plik nagłówkowy `xxx.h` zawarty w bibliotece standardowej języka C++ umieszcza te deklarowane identyfikatory, które analogiczny plik nagłówkowy `cxxx` umieszcza w przestrzeni nazw[16] (`std`), w *globalnej* przestrzeni nazw.

Aby uniknąć zaśmiecania przestrzeni nazw w miarę możliwości zalecane jest korzystanie z plików nagłówkowych `cxxx`. Zwróć jednak uwagę, że niektóre pliki nagłówkowe `cxxx` automatycznie umieszczają ten sam identyfikator zarówno w przestrzeni nazw `std`, jak i w globalnej przestrzeni nazw (np. `<cstdio>` umieszcza identyfikator `printf` w obu tych przestrzeniach nazw).

Więcej o zagadnieniu tych „zgodnościowych” plików nagłówkowych przeczytasz tu.

Typ `bool`

Standard C99 wprowadzał wbudowany typ logiczny `_Bool`, natomiast identyfikatory `bool`, `true` i `false` stanowiły odpowiednie makra zdefiniowane w bibliotece standardowej (a nie w jądrze języka) – przez co aby skorzystać z tych identyfikatorów należało dołączyć plik nagłówkowy `<stdbool.h>`.

---

[13] Obiekt `std::cout` został omówiony dokładniej w rozdziale 7.5 Operacje wejścia/wyjścia w oparciu o strumienie.

[14] O tym, na czym polega różnica między plikami nagłówkowymi `<stdio.h>` i `<cstdio>` przeczytasz w rozdziale 1.5.1 Nagłówki biblioteki standardowej.

[15] O tym, dlaczego stosowanie `using namespace std;` to zła praktyka przeczytasz w rozdziale 16 Przestrzenie nazw.

[16] O przestrzeniach nazw przeczytasz w rozdziale 16 Przestrzenie nazw.

Tablica 1.1. Różnice między językami C i C++ w podejściu do programowania

Aspekt	C	C++
typ programowania	proceduralny [1ex]Język C umożliwia wyłącznie programowanie proceduralne – skupione na procedurach (operujących na danych), a nie na danych.	wieloparadygmatowy [1ex]Język C++ umożliwia zarówno programowanie proceduralne, jak i zorientowane na obiekty (na których wykonywane są operacje) i na klasy, a nie na procedury.
bezpieczeństwo i spójność danych	niewielkie [1ex]Język C (jako język proceduralny) nie koncentruje się na danych – udostępnia w zasadzie tylko funkcjonalność grupowania danych w struktury (bez enkapsulacji) oraz kwalifikator `const`.	duże [1ex]Język C++ (jako język wspierający OOP) umożliwia enkapsulację danych w klasach (z użyciem specyfikatorów dostępu).
organizacja programu[a]	funkcje	funkcje, klasy, przestrzenie nazw
sygnalizowanie sytuacji wyjątkowych[b]	kody błędów [1ex](zwykłe zwracanie pewnych umownych – w ramach danego programu – wartości)	mechanizm wyjątków [1ex](przerywanie „normalnego” wykonania programu; spójny pomiędzy programami)
generyczność[c]	niewielka [1ex]podstawowe wsparcie („generyczny wybór”) dopiero w C11	duża [1ex](szablony, koncepty, przeciążanie funkcji, polimorfizm. . .)
zarządzanie zasobami	brak	ścisłe [1ex]RAII, konstruktory i destruktory

[a] Przez „organizację programu” rozumiane są mechanizmy pozwalające podzielić program na mniejsze części pod względem funkcjonalnym oraz pozwalające uniknąć konfliktów identyfikatorów.
[b] Chodzi o sytuacje sprawiające, że dalsze wykonywanie kodu jest niemożliwe.
[c] *Generyczność* to możliwość implementacji algorytmu w sposób uniwersalny, bez konieczności podawania konkretnego typu danych (np. algorytm sortowania powinien działać dla dowolnego typu definiującego relację porządku liniowego).

W języku C++ `bool` to *słowo kluczowe* oznaczające wbudowany typ logiczny, podobnie jak słowami kluczowymi są `true` i `false` (czyli funkcjonalność ta jest zawarta w jądrze języka i nie ma konieczności dołączania dodatkowych plików nagłówkowych, aby z niej skorzystać).

Wskaźnik pusty

Do reprezentacji wartości wskaźnika pustego służy:

- w języku C: *stała symboliczna* `NULL` (zdefiniowana w bibliotece standardowej)
- w języku C++: *literał* `nullptr`

Pre- i postinkrementacja a iterowanie po zakresie obiektów

Począwszy od wprowadzenia standardu C++11, który wprowadził konstrukcję *range-based* `for` *loop*[17], stosunkowo rzadko zachodzi konieczność „ręcznej” inkrementacji wartości licznika pętli.

[17] zob. rozdz. 7.6.2 Range-based `for` loop

Natomiast aby uniknąć różnych innych niespodzianek, do inkrementacji i dekrementacji (zwłaszcza obiektów typów klasowych, np. iteratorów) staraj się używać operatorów „pre” (`++x` i `--x`).

Słowo kluczowe `auto`

W języku C słowo kluczowe **`auto`** jest specyfikatorem klasy przechowywania.

Standard C++11 zmienił znaczenie słowa kluczowego **`auto`** – obecnie stanowi ono tzw. *zastępczy symbol specyfikatora typu* (został on opisany w rozdz. 7.6.1 Zastępczy symbol specyfikatora typu: `auto`).

Inicjalizacja struktur

Standard C99 dopuścił możliwość odwoływania się podczas inicjalizacji struktury do jej pól z użyciem zapisu `.<nazwa_pola>`[18], natomiast w języku C++ analogiczna funkcjonalność ma zostać wprowadzona w standardzie C++20.

Literały znakowe

W języku C literały znakowe są typu **`int`**, natomiast w C++ są typu **`char`**.

Niejawna konwersja z typu `void*`

W języku C++ mamy do czynienia ze ściślejszą typizacją niż w języku C – język C++ nie pozwala m.in. na *niejawną* konwersję typu **`void*`** na inny typ postaci `T*`, przykładowo:

```
void* ptr;
int* i = ptr;  // BŁĄD: niejawna konwersja z `void*` - niedozwolone w C++
```

Zamiast tego język C++ zaleca użycie operatora **`reinterpret_cast`**:

```
void *ptr;
int* i = reinterpret_cast<int*>(ptr);
```

Prototypy funkcji bez parametrów

W języku C prototyp funkcji nieposiadający parametrów (np. **`int`** `foo()`) oznaczał, że parametry nie są określone – możliwe było zatem wywołanie takiej funkcji z jednym lub kilkoma parametrami (np. `foo(1, "abc")`). Był to relikt pochodzący z czasów jeszcze przed C89, jednak pozostawiony w celu zachowania kompatybilności wstecznej (choć standardy języka C jasno odradzały użycia tego rozwiązania).

W języku C++ prototyp funkcji nieposiadający parametrów oznacza, że funkcja ta nie przyjmuje żadnych argumentów. (W języku C należało w tym celu użyć typu **`void`**, np. **`int`** `bar(`**`void`**`)`, co również jest formą dopuszczalną – lecz niezalecaną – w języku C++.)

Definiowanie struktur, wyliczeń i unii

Ogólna składnia używana do definiowania typów strukturalnych, typów wyliczeniowych i typów unii jest identyczna w obu językach, przykładowo:

```
// Definicja typu strukturalnego `struct S` -- identyczna w C i C++.
struct S {
  /* ... */
};
```

Jednak w języku C aby skorzystać później z takiego typu należy użyć pełnej jego nazwy, czyli w powyższym przypadku:

```
struct S s;  // [C] Definicja zmiennej strukturalnej typu `struct S`
```

[18] tzw. *designated initializers*

Ponieważ konieczność poprzedzania zdefiniowanej przez nas etykiety typu znacznikiem **`struct`**, **`enum`** albo **`union`** jest męcząca, można zdefiniować odpowiedni alias:

```
typedef struct {
  /* ... */
} S;

S s;  // [C] Dzięki aliasowi nie trzeba stosować znacznika `struct`.
```

W języku C++ nie ma konieczności stosowania znaczników podczas korzystania ze wspomnianych typów:

```
struct S {
  /* ... */
};

S s;  // [C++] Nie trzeba stosować znacznika `struct`.
```

Dopuszczalne identyfikatory

Zwróć uwagę, że język C++ wprowadza szereg nowych elementów względem języka C – w szczególności definiuje znacznie więcej słów kluczowych (np. **`template`**, **`new`**, **`delete`**. . .). Oznacza to, że próba zbudowania programu zawierającego fragmenty kodu skopiowane z innego programu napisanego w języku C może zakończyć się błędem związanym z konfliktem oznaczeń.

Inne sposoby realizacji tych samych operacji

Język C++ dopuszcza stosowanie pewnych funkcjonalności języka C, jednak jednocześnie udostępnia własne funkcjonalności służące rozwiązaniu tych samych problemów, przykładowo:

- Język C++ udostępnia wyspecjalizowane operatory rzutowania (np. **`static_cast`**, **`const_cast`** itp.).
- W języku C++ do dynamicznego zarządzania pamięcią służą pary *operatorów* **`new`**/**`new[]`** i **`delete`**/**`delete[]`** *w przybliżeniu* odpowiadające parze *funkcji bibliotecznych* `malloc()` i `free()` [19].

1.6 Gdzie szukać pomocy?

Język C++ zawiera sporo niuansów, przez co nawet doświadczeni programiści zwykle wspierają się w swojej pracy różnymi materiałami źródłowymi – nie sposób zapamiętać *całej* dostępnej funkcjonalności języka C++ (wraz z biblioteką standardową). Dlatego niezmiernie istotne jest wiedzieć gdzie i jak szukać pomocy podczas projektowania i kodowania programów w C++.

1.6.1 cppreference.com

Strona cppreference.com zawiera kompletną specyfikację języka C++ (wraz z funkcjonalnością biblioteki standardowej). Należy jednak pamiętać, że strona tworzona jest przez grupę entuzjastów C++, stąd istnieje ryzyko drobnych pomyłek względem standardu – są one jednak na bieżąco poprawiane.

1.6.2 cplusplus.com

Strona cplusplus.com zawiera nie tylko kompletną specyfikację języka C++ (wraz z funkcjonalnością biblioteki standardowej), lecz także przystępny kurs C++. Należy jednak pamiętać, że strona tworzona jest przez grupę entuzjastów C++, stąd istnieje ryzyko drobnych pomyłek względem standardu – są one jednak na bieżąco poprawiane.

[19] zob. rozdz. 13.1.1 Dynamiczna alokacja pamięci

1.6.3 Stack Overflow

Serwis Stack Overflow zrewolucjonizował całą sferę IT – to de facto liczące ponad 8 mln użytkowników[20] forum pytań i odpowiedzi, które dodatkowo posiada system nagradzania dobrych pytań i trafnych odpowiedzi (oraz karania bezwartościowych pytań i błędnych odpowiedzi).

W praktyce, jeśli natrafisz na jakikolwiek problem podczas tworzenia programu – czy to związany z decyzjami projektowymi, czy elementami języka, czy wreszcie na błędy kompilacji – w pierwszej kolejności poszukaj odpowiedzi na Stack Overflow. Jeśli nie znajdziesz satysfakcjonującej odpowiedzi „na Stacku", samemu zadaj pytanie – zwykle błyskawicznie uzyskasz poprawną odpowiedź udzieloną przez osobę z bogatym doświadczeniem.

Ważne

Korzystając z serwisu Stack Overflow zwracaj baczną uwagę na punkty przydzielone danemu pytaniu i danej odpowiedzi. W szczególności omijaj odpowiedzi, które mają mniej niż 0 punktów – zwykle są to odpowiedzi błędne.

O systemie głosowania (nagradzania i karania) w serwisie Stack Overflow przeczytasz na stronie Why is voting important? (Stack Overflow).

[20] Stan na koniec 2017 r.

Rozdział 2

Obiekty i klasy – podstawy

W tym rozdziale zapoznasz się z najbardziej podstawowymi pojęciami związanymi z programowaniem obiektowym oraz z równie podstawowymi mechanizmami języka C++ wspierającymi obiektowość.

2.1 Czym jest obiekt?

Obiekt (ang. object) to moduł wiążący dane z kodem funkcji służących do wykonywania na tych danych określonych zadań[1]. Obiekty w programowaniu obiektowym mają na celu modelowanie świata rzeczywistego – w którym stan obiektu i jego zachowanie są ze sobą ściśle powiązane. Stosowanie obiektów redukuje złożoność problemu oraz ułatwia zarządzanie kodem.

> **Informacja**
>
> *Pojęcie obiektu nie ma jednej powszechnie przyjętej definicji. W ogólnym ujęciu obiekt to miejsce w pamięci, w którym mogą być przechowywane dane pewnego określonego typu.*
>
> *Programiści C++ mają dość swobodne podejście do używania terminu „obiekt" – niektórzy stosują go wyłącznie w odniesieniu do instancji klas, inni do każdych danych modyfikowanych w programie (a terminu „wartość" do danych tylko do odczytu). . .*
>
> *W niniejszym skrypcie termin* obiekt *będzie stosowany w pierwszym kontekście, tj. w odniesieniu do instancji klas, a termin* obiekt danych *– w drugim (jako dane dowolnego typu, na których program może operować).*

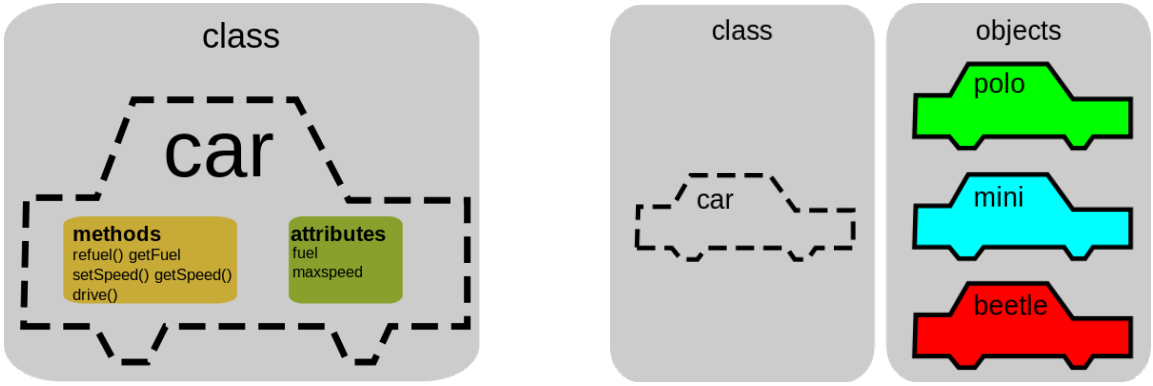
2.2 Czym jest klasa?

Klasa (ang. class) to swego rodzaju „przepis" określający sposób tworzenia i korzystania z obiektów danego rodzaju (tj. ogólne zachowanie każdego obiektu tego rodzaju) – wszystkie obiekty danej klasy posiadają ten sam identyczny zbiór *cech* i *zachowań* (przy czym konkretne *wartości* cech mogą różnić się pomiędzy tymi obiektami)[2]. Przykładowo, w realnym świecie spotykamy się z różnymi egzemplarzami dmuchanych piłek do gry – są piłki wykonane z tkaniny albo z tworzyw sztucznych, piłki duże i małe, jednokolorowe albo wielobarwne itd. Mimo to wszystkie te egzemplarze posiadają identyczny zbiór *cech* (np. materiał wykonania, rozmiar, kolor) oraz *zachowań* (np. każdą dmuchaną piłkę można napompować). Możemy więc powiedzieć, że egzemplarze te są *obiektami* (konkretnymi przykładami) pewnej *klasy* (pewnej *abstrakcji*) – dmuchanej piłki do gry – posiadającej wspomniane cechy oraz definiującej wspomniane zachowania. Analogiczny przykład pokazano na rysunku 2.1.

W języku C++ pojęciu *abstrakcji* odpowiada pojęcie klasy, natomiast obiekty (egzemplarze) danej klasy nazywamy jej **instancjami** (ang. instances). Przykładowo, instrukcja `Rectangle rect_a, rect_b;` tworzy dwa obiekty (`rect_a` i `rect_b`) będące instancjami klasy `Rectangle`. Każdy z tych obiektów posiada własny, *odrębny* zestaw zmiennych i metod (zmiana stanu jednego obiektu danej klasy nie powoduje zmiany stanu innego obiektu tej samej klasy).

[1] Obiekt stanowi zatem rozwinięcie koncepcji *struktury danych* (zob. rozdz. 2.5.2 Struktury danych a abstrakcyjne typy danych).

[2] Innymi słowy, klasy umożliwiają definiowanie *abstrakcyjnych typów danych* (zob. rozdz. 2.5.2 Struktury danych a abstrakcyjne typy danych).



(a) (b)

Rysunek 2.1. Klasa i jej obiekty: (a) klasa określa rodzaj cech (bez ich precyzowania) oraz zachowanie *każdego* obiektu; (b) klasa służy do tworzenia konkretnych obiektów (o sprecyzowanych cechach) zgodnie z zadanym „przepisem” (Źródło: Wikipedia)

2.3 Użytkownik programu a użytkownik kodu programu

Zwykle gdy programista mówi o osobach korzystających z jego programu, określa ich mianem *użytkowników*. Podobnie projektant klas projektuje i implementuje klasy dla *użytkowników* tej klasy. W tym drugim przypadku użytkownikiem jest programista (użytkownik *kodu* programu), a nie „użytkownik końcowy” gotowej aplikacji. Projektując klasy powinniśmy mieć na względzie wygodę korzystania z niej przez jej użytkowników – a więc innych programistów. Interfejs klasy powinien być możliwie prosty i intuicyjny, uwzględniający typowe przypadki użycia klasy, a korzystanie z klasy nie powinno wymagać znajomości jej wewnętrznych mechanizmów.

2.4 Struktura klasy – składowe (pola i metody)

Klasy definiuje się za pomocą słowa kluczowego `class` lub `struct`[3]. Najprostsza definicja klasy ma następującą postać:

Listing 2.1. Ogólny schemat definicji klasy.

```
struct ClassName {
  member1;
  member2;
  ...
};
```

gdzie `ClassName` to identyfikator klasy. Ciało klasy zawiera opcjonalne deklaracje **składowych** (ang. members) klasy – **pól** (ang. fields) lub **metod** (ang. methods). Pola i metody to odpowiednio zmienne i funkcje powiązane z daną klasą.

Składowe klasy deklarujemy analogicznie do zwykłych zmiennych i funkcji, np.:

Listing 2.2. Klasa `LightSwitch` reprezentująca wyłącznik światła.

```
struct LightSwitch {
  bool is_switched_on_;  // pole klasy
  void do_switch() { is_switched_on_ = !is_switched_on_; }  // metoda klasy
};
```

Informacja

W niniejszym skrypcie (podobnie jak w skrypcie do nauki języka C) przyjęto standard kodowania wzorowany na wytycznych ze strony C++ Core Guidelines (B. Stroustrup, H. Sutter). W zasadzie w odniesieniu do klas i składowych obowiązują zatem znane Ci już z języka C konwencje nadawania identyfikatorów:

[3] Różnica zostanie wyjaśniona w rozdziale 2.5 Enkapsulacja.

- *Nazewnictwo klas nie różni się niczym od nazewnictwa typów strukturalnych z języka C (tj. każdy człon z dużej litery).*
- *Nazewnictwo pól i metod nie różni się wiele od nazewnictwa zmiennych i funkcji w języku C – podstawowy fragment nazwy może składać się z członów zapisywanych z użyciem małych liter i rozdzielonych podkreśleniami, natomiast (tu różnica) na końcu tego fragmentu dodaje się znak podkreślenia[4] (_), aby później łatwo odróżnić nazwy pól od nazw zmiennych lokalnych (i żeby nie dochodziło do przesłaniania pól przez zmienne lokalne – jak w przykładzie 2.3).*

Listing 2.3. Przesłanianie nazw w metodach.

```
class Point2D {
public:
  Point2D(double x, double y) {
    // PROBLEM: Parametr `x` przesłania pole `x`.
    x = x;
    y = y;
  };
private:
  double x;
  double y;
};
```

2.5 Enkapsulacja

Pojęcia *enkapsulacja* używa się w dwóch kontekstach[5]:

- W odniesieniu do konstrukcji języka umożliwiających powiązanie danych z funkcjami operującymi na tych danych.
- W odniesieniu do mechanizmów języka ograniczających dostęp do pewnych składników obiektu.

Enkapsulacja ma na celu oddzielenie interfejsu klasy od jego implementacji. Klasa, która realizuje enkapsulację, *ukrywa* swoją implementację przed użytkownikami takiej klasy – w szczególności użytkownicy nie mają bezpośredniego dostępu do danych, a jedynie dostęp poprzez interfejs. Ogólna zasada mówi: użytkownik klasy powinien mieć dostęp do jej metod, ale nie do jej danych.

Główne zalety stosowania enkapsulacji to:

- Uniemożliwienie użytkownikowi klasy przypadkowego „popsucia” jej stanu (np. poprzez ustawienie niedozwolonych wartości pola klasy).
- Zmiana implementacji klasy („wewnętrznych detali” – np. sposobu reprezentacji danych) nie ma wpływu na poprawność działania modułów korzystających z takiej klasy.

Do realizacji idei enkapsulacji w C++ służy m.in. mechanizm kontroli dostępu[6].

Przykładem enkapsulacji znanym z życia codziennego będzie choćby mechanizm wydawania przesyłek na poczcie. Pracownik urzędu pocztowego pełni rolę interfejsu:

- uniemożliwia adresatom przesyłek bezpośredniego dostępu do nich. Dzięki temu użytkownicy systemu pocztowego nie mają możliwości zabrać nie swojej przesyłki („popsuć” stan systemu), a zarazem
- dzięki niemu adresaci nie muszą zastanawiać się „w którym pomieszczeniu na zapleczu i na którym regale znajduje się moja przesyłka” – co byłoby szczególnie problematyczne podczas reorganizacji pomieszczeń (zmiany implementacji).

2.5.1 Kontrola dostępu

Mechanizm kontroli dostępu wspomaga osiągnięcie enkapsulacji – zapewnia, że użytkownicy klasy nie mają możliwości „przypadkiem” (bądź celowo) „uszkodzić” stanu obiektu takiej klasy.

[4] Nie istnieje spójna konwencja wyróżniania pól klasy – w zależności od konkretnego standardu kodowania stosuje różne prefiksy albo sufiksy nazwy.

[5] zob. rozdz. 1.1 Programowanie zorientowane na obiekty

[6] zob. rozdz. 2.5.1 Kontrola dostępu

Język C++ definiuje **specyfikatory dostępu** (ang. access specifiers), aby ograniczyć dostęp do składowych klasy, w szczególności:

- Składowe zadeklarowane jako **publiczne** (ang. public) są dostępne we wszystkich częściach programu – stanowią interfejs programistyczny klasy (tj. jej API).
- Składowe zadeklarowane jako **prywatne** (ang. private) są dostępne jedynie wewnątrz metod danej klasy – nie ma do nich dostępu w kodzie korzystającym z takiej klasy. Składowe prywatne realizują zatem enkapsulację – poprzez ukrycie implementacji fragmentów klasy.

Specyfikatory dostępu w języku C++ to m.in. odpowiednio `public` dla składowych publicznych i `private` dla składowych prywatnych[7].

Definicja klasy korzystającej z kontroli dostępu ma następującą postać:

```
<class|struct> ClassName {  // można użyć albo `class`, albo `struct`
  access_specifier_1:
    member1;
    ...
  access_specifier_2:
    member2;
    ...
  ...
};
```

Ważne

W języku C++ różnica między użyciem do zdefiniowania klasy słowa kluczowego `class` i `struct` polega *wyłącznie* na domyślnym specyfikatorze dostępu do składowych:

- W przypadku `class` – domyślnie dostęp *prywatny*.
- W przypadku `struct` – domyślnie dostęp *publiczny*.

Zatem jeśli zdefiniujemy z użyciem `struct` klasę posiadającą wyłącznie pola, w zasadzie nie będzie się ona różniła od struktury danych utworzonej za pomocą słowa kluczowego `struct` w języku C.

Klasa `LightSwitch` z przykładu 2.2 miała wszystkie składowe (domyślnie) publiczne, przez co naruszała zasady enkapsulacji – każdy mógłby zmienić stan tego obiektu bezpośrednio modyfikując pole `is_switched_on_`. Rozsądniej byłoby ograniczyć użytkownikom tej klasy dostęp do jej składowych tak, aby mogli oni jedynie korzystać z metody `do_switch()` (zob. przykład 2.4). W tym celu dokonano następujących zmian:

- Zmieniono słowo kluczowe użyte w definicji klasy `LightSwitch` ze `struct` na `class` – w przypadku tego typu klas, reprezentujących *abstrakcyjny typ danych* lepiej dla bezpieczeństwa domyślnie traktować składowe jako prywatne.
- Ustawimy metodzie `do_switch()` dostęp publiczny – niezbędne, aby stała się częścią API klasy `LightSwitch` i była widoczna dla jej użytkowników.
- Jawnie ustawimy polu `is_switched_on_` dostęp prywatny – dla czytelności kodu (oraz aby uniknąć niespodzianek w przypadku ewentualnej ponownej zmiany słowa kluczowego użytego do zdefiniowania klasy. . .).

Listing 2.4. Przykład użycia kontroli dostępu do składowych.

```
class LightSwitch {
public:
  void do_switch() { is_switched_on_ = !is_switched_on_; }
  bool is_switched() { return is_switched_on_; }
private:
  bool is_switched_on_;
};
```

[7] Trzeci typ dostępu – dostęp chroniony (o specyfikatorze `protected`) – poznasz w rozdziale 8.1.2 Składowe chronione.

2.5.2 Struktury danych a abstrakcyjne typy danych

Struktura danych (ang. data structure) w ogólnym ujęciu wyraża konkretną reprezentację danych, zatem przedstawia punkt widzenia implementacji (a nie użytkownika). Z kolei klasa, która w pełni korzysta z koncepcji abstrakcji danych (ang. data abstraction) i enkapsulacji, stanowi **abstrakcyjny typ danych** (ang. abstract data type, ADT). W przypadku abstrakcyjnego typu danych to projekt klasy troszczy się o jej poprawną implementację, natomiast użytkownika takiej klasy interesuje jedynie jej funkcjonalność.

W związku z tym słowo kluczowe użyte do zdefiniowania danej klasy w języku C++ powinno oddawać jej przeznaczenie:

- definiując klasę pełniącą rolę *struktury danych* używaj słowa kluczowego `struct`, a
- definiując klasę pełniącą rolę *abstrakcyjnego typu danych* używaj słowa kluczowego `class`.

2.6 Korzystanie ze składowych klasy

Po zadeklarowaniu obiektu dostęp do jego (publicznych) składowych uzyskujemy poprzez zastosowanie operatora `.` (kropka), który umieszczamy pomiędzy nazwą obiektu a nazwą składowej – podobnie jak w przypadku korzystania ze struktur w języku C, np.:

```
LightSwitch ls;
ls.do_switch(); // dostęp do składowej
```

Podobnie w przypadku wskaźnika do obiektu typu klasowego (tu: `LightSwitch* ls_ptr`) możemy skorzystać z jednego z dwóch równoważnych zapisów:

```
(*ls_ptr).do_switch();
ls_ptr->do_switch();  // użycie operatora ->
```

przy czym zapis z użyciem operatora `->` jest preferowany ze względu na większą przejrzystość kodu.

2.6.1 Definiowanie metod

Metody możemy definiować na dwa sposoby – albo w ciele klasy, albo poza nim. Z pierwszego sposobu skorzystaliśmy np. w przykładzie 2.4. Drugi sposób zdefiniowania metody `do_switch()` przedstawia przykład 2.5.

Listing 2.5. Przykład definiowania metody poza ciałem klasy.

```
class LightSwitch {
public:
  void do_switch();
private:
  bool is_switched_on_;
};

void LightSwitch::do_switch() {  // niezbędne użycie operatora zasięgu `::`
  is_switched_on_ = !is_switched_on_;
}
```

Operator zasięgu `::` pozwala zaznaczyć, że funkcja zdefiniowana poza klasą jest w istocie metodą klasy `LightSwitch`, a nie „zwykłą” funkcją. Co do zasady umożliwia on również dostęp do składowych klasy `LightSwitch` wewnątrz definicji metody `do_switch()`, np.:

```
void LightSwitch::do_switch() {  // niezbędne użycie operatora zasięgu `::`
  LightSwitch::is_switched_on_ = !LightSwitch::is_switched_on_;  // zły styl
  // (W powyższej instrukcji użycie operatora zasięgu jest zbędne.)
}
```

przy czym ponieważ zgodnie z zasadami języka C++ dostęp do składowych wewnątrz definicji klasy oraz wewnątrz definicji jej metod nie wymaga **kwalifikacji** (ang. qualification) – czyli w tym przypadku podania nazwy klasy wraz z operatorem zasięgu – nie musimy poprzedzać nazwy pola `is_switched_on_` przedrostkiem `LightSwitch::` (a skoro nie musimy, to w celu poprawy przejrzystości kodu go nie stosujemy).

Który sposób definicji wybrać? Krótka odpowiedź brzmi – poza ciałem klasy, przy czym klasa powinna być zdefiniowana w pliku nagłówkowym, a metoda w pliku źródłowym[8]. (Dłuższa odpowiedź brzmi – to zależy.)

Dla zainteresowanych...

Co do zasady typy danych powinno się definiować w plikach nagłówkowych.

Umieszczenie długich implementacji w definicji klasy (a więc wewnątrz pliku nagłówkowego) niepotrzebnie wydłużyłoby czas preprocessingu jednostek translacji dołączających ten plik nagłówkowy – w związku z tym co do zasady metody powinno się definiować w pliku źródłowym.

Z drugiej strony jednak krótkie metody o ciele złożonym z pojedynczej instrukcji[9] mogą spokojnie zostać umieszczone w definicji klasy w pliku nagłówkowym, gdyż nie powodują istotnego zwiększenia rozmiaru pliku nagłówkowego, za to znacznie poprawiają czytelność pliku źródłowego zawierającego pozostałe definicje metod.

Dawniej istotny w tej kwestii był również aspekt wydajnościowy związany z wplataniem funkcji, gdyż zdefiniowanie metody w pliku nagłówkowym stanowiło dla kompilatora sugestię, że można taką metodę potraktować jak funkcję wplataną[10] i zastosować dodatkowe optymalizacje kodu.

Podczas definiowania metod poza klasą należy pamiętać, że odwoływanie się do składowych klasy bez użycia kwalifikowanych nazw jest możliwe dopiero po podaniu kwalifikowanej nazwy metody. Ponieważ jednak typ zwracany przez metodę znajduje się przed nazwą metody, w takiej sytuacji musimy podać nazwę kwalifikowaną (zob. przykład 2.6).

Listing 2.6. Przykład definiowania metody poza ciałem klasy, gdy typ zwracany przez metodę stanowi składową klasy.

```
class WindowManager {
public:
  class ScreenIndex {
    ...
  };
  ScreenIndex set_screen_index();
  ...
};
// Typ zwracany jest widoczny przed (kwalifikowaną) nazwą metody,
// dlatego również wymaga kwalifikacji.
WindowManager::ScreenIndex WindowManager::set_screen_index() {
  ...
}
```

2.7 Konstruktor i destruktor

Ponieważ klasy, jako abstrakcyjne typy danych, muszą zwykle zapewniać pewne niezmienniki, inicjalizacja instancji klas i ich usuwanie jest bardziej złożone niż dla struktur danych znanych z języka C – konstruktory i destruktory to specjalne metody klasy, o niemal dowolnej złożoności, używane właśnie w celu konstruowania i usuwania instancji klas.

Zagadnienia związane z konstruowaniem i usuwaniem obiektów są złożone i zostaną omówione dokładnie w rozdziale 9 Klasy: Tworzenie i niszczenie obiektów. Poniżej wprowadzono jedynie najbardziej podstawowe pojęcia oraz omówiono najbardziej podstawowe mechanizmy, aby umożliwić Ci rozpoczęcie przygody z obiektowością w języku C++.

2.7.1 Konstruktor

Konstruktor (ang. constructor) to metoda zapewniająca, że nowo utworzony obiekt spełnia niezmienniki gwarantowane przez klasę (w tym celu np. odpowiednio zainicjalizuje pola, przydzieli dynamicznie pamięć dla obiektów będących składowymi klasy). Sam konstruktor nie przydziela pamięci do przechowywania obiektu, który konstruuje – może jednak przydzielać pamięć dla jego składowych.

Aby zilustrować działanie konstruktora, zdefiniujmy klasę `Circle` reprezentującą okrąg jak w przykładzie 2.7.

[8] Podział ten został opisany w rozdziale 3 Organizacja programu

[9] np. trywialne „gettery" i „settery" – czyli metody mające umożliwić jedynie odpowiednio prosty odczyt albo proste ustawienie wartości odpowiedniego pola klasy

[10] zob. When do compilers inline C++ code? (Stack Overflow)

Listing 2.7. Klasa posiadająca wyłącznie konstruktor domyślny.

```
class Circle {
public:
  void set_radius(double r) { radius_ = r; }
  double area() { return 3.14 * radius_ * radius_; }
private:
  double radius_;
};
```

W przypadku tak zdefiniowanej klasy musimy pamiętać o tym, aby każdorazowo po utworzeniu jej instancji ustawić obiektowi odpowiedni promień:

```
Circle circ;
circ.set_radius(2.0);
std::cout << circ.area() << std::endl;
```

W powyższym kodzie w chwili tworzenia obiektu `circ` wywołany zostanie **konstruktor domyślny**[11], który w tym przypadku pozostawi pole `radius_` z wartością niezdefiniowaną. Dopiero po wywołaniu metody `set_radius()` pole `radius_` zostanie zainicjalizowane wartością `2.0`. Zwykle jednak chcemy od razu w chwili tworzenia instancji klasy `Circle` podać promień takiego okręgu, który to promień w dodatku (w naszym hipotetycznym przypadku) nie powinien się zmieniać w trakcie cyklu życia obiektu. W takiej sytuacji zamiast definiować metodę `set_radius()` możemy zdefiniować konstruktor z jednym parametrem a jednocześnie uniemożliwić późniejszą zmianę wartości promienia usuwając metodę `set_radius()` (zob. przykład 2.8).

Listing 2.8. Klasa posiadająca konstruktor domyślny i konstruktor jednoargumentowy.

```
class Circle {
public:
  // Oto 1-argumentowy konstruktor klasy `Circle`.
  // UWAGA! W poniższej sytuacji do inicjalizacji pola
  // powinno używać się listy inicjalizacyjnej konstruktora,
  // która zostanie omówiona w kolejnych rozdziałach.
  Circle(double r) { radius_ = r; };

  double area() { return 3.14 * radius_ * radius_; }
private:
  double radius_;
};
```

Korzystanie z obecnej wersji klasy wygląda następująco:

```
Circle circ(2.0);
std::cout << circ.area() << std::endl;
```

Podobnie jak każda „zwykła” metoda, konstruktory posiadają listę parametrów (która może być pusta) oraz ciało funkcji (które również może być puste). W klasie może być zdefiniowanych kilka konstruktorów, pod warunkiem że różnią się listą parametrów (posiadają różną liczbę parametrów i/lub parametry mają różne typy) – podczas tworzenia obiektu kompilator sam wybierze odpowiedni konstruktor, w zależności od podanych argumentów[12]. Przykładowo, zmodyfikowana klasa `Circle` z przykładu 2.8 posiada wyłącznie jeden konstruktor – jednoargumentowy konstruktor `Circle(double r)`.

Choć konstruktory zachowaniem przypominają zwykłe metody, mają kilka cech wyróżniających je, w szczególności:

- Nazwa konstruktora jest identyczna z nazwą klasy.
- Konstruktory nie posiadają typu zwracanego (nawet `void`) – nie mogą zatem zwracać wartości.
- Konstruktory są wywoływane przez kompilator tylko raz, w chwili tworzenia obiektu. Nie ma możliwości wywołania konstruktora ponownie, na późniejszym etapie, w celu ponownej inicjalizacji obiektu.

[11] Konstruktor domyślny zostanie omówiony w rozdziale 9.2.1 Konstruktor domyślny.

[12] Mechanizm ten, zwany *przeciążaniem funkcji*, został dokładniej omówiony w rozdziale 17.1 Przeciążanie funkcji.

2.7.2 Destruktor

Destruktor (ang. destructor) to metoda zapewniająca, że obiekt po sobie „posprząta” (np. zwolni zasoby przydzielone obiektowi, takie jak pamięć przydzieloną dynamicznie dla jego składowych). Destruktor wywoływany jest automatycznie w chwili, gdy kończy się cykl życia obiektu – gdy program opuszcza zakres, w którym obiekt był zadeklarowany statycznie albo gdy pamięć dynamicznie zaalokowanego obiektu jest zwalniana.

Choć destruktory zachowaniem przypominają zwykłe metody, mają kilka cech wyróżniających je, w szczególności:

- Nazwa destruktora jest identyczna z nazwą klasy, z tym że jest ona poprzedzona tyldą (~).
- Destruktory ani nie posiadają typu zwracanego (nawet `void`), ani nie mogą przyjmować argumentów.
- Nie należy wywoływać destruktora „ręcznie” – w stosownym momencie kompilator sam wywoła destruktor.

Ponieważ standardowo każda klasa posiada domyślny destruktor (generowany przez kompilator), który zapewnia poprawne usunięcie obiektów dla klas nie wymagających specjalnej procedury usuwania (np. zaalokowanych dynamicznie), w najbliższym czasie nie musisz sobie zaprzątać głowy zagadnieniem definiowania destruktorów.

2.8 Strukturalne a obiektowe podejście do programowania: przykład

Jak wspomniano w rozdziale 1 Wprowadzenie do języka C++, w podejściu strukturalnym dane w programie są tylko luźno powiązane z funkcjami operującymi na tych danych i podobnie luźne są powiązania między funkcjami. Oto przykład fragmentu kodu strukturalnego (napisanego w języku C):

```
int area(int width, int height) {
  return width * height;
}

// ...

int my_rect_width = 100;
int my_rect_height = 50;

printf("Area is %d\n", area(my_rect_width, my_rect_height));
```

Takie podejście utrudnia tworzenie złożonych systemów – w szczególności dlatego, że brakuje logicznego i funkcjonalnego grupowania elementów (co wprowadza chaos w kodzie), a dodatkowo brak kontroli dostępu do danych i operacji znacząco zmniejsza bezpieczeństwo kodu.

Oto przykład kodu programu napisanego w języku C++ realizującego analogiczną funkcjonalność co powyższy program, tyle że zorientowanego obiektowego:

```
class Rectangle {
  public:
    int width_;
    int height_;
    Rectangle(int width, int height) {
      width_ = width; height_ = height;
    }
    int area() { return width_ * height_; }
};

// ...

Rectangle rect(100, 50);

std::cout << "Area is " << rect.area() << std::endl;
```

Jak widać, „opakowanie” danych oraz operujących na nich funkcji we wspólną klasę poprawia czytelność kodu i ułatwia korzystanie z takiej abstrakcji.

Rozdział 3

Organizacja programu

W tym rozdziale zapoznasz się ze sposobem podziału programu na pliki nagłówkowe i źródłowe w języku C++ tak, aby poprawić jego czytelność i umożliwić wielokrotne wykorzystywanie kodu.

W znakomitej większości sposób ten jest identyczny, co w przypadku języka C – natomiast poniżej omówiono najważniejsze różnice oraz nowe zagadnienia wynikające ze specyfiki języka C++:

- Język C++ nie precyzuje sposobu nazywania plików źródłowych i nagłówkowych, nie obowiązuje również w tej kwestii spójna konwencja. Jednak pisząc program w języku C++ *nie należy* stosować rozszerzeń „`.c`” dla plików źródłowych i „`.h`” dla plików nagłówkowych, tylko np. odpowiednio „`.cpp`” i „`.hpp`”[1], gdyż przykładowo to na podstawie rozszerzenia pliku źródłowego narzędzia w stylu CMake określają, czy skompilować go z użyciem kompilatora języka C czy C++.
- W plikach nagłówkowych umieszczamy definicje klas, natomiast implementacje ich metod powinny się co do zasady znaleźć w plikach źródłowych (pewne odstępstwa od tej konwencji przedstawiono w rozdziale 2.6.1 Definiowanie metod).
- W przeciwieństwie do języka C, w języku C++ globalne identyfikatory w przestrzeni nazw[2] (w tym w globalnej przestrzeni nazw) zadeklarowane ze specyfikatorem **`const`** i jednocześnie bez specyfikatora **`extern`** mają *łączność wewnętrzną* – nie dochodzi zatem to konfliktów podczas konsolidacji różnych jednostek translacji deklarujących ten sam identyfikator. W związku z tym można definiować (i jednocześnie zainicjalizować) zmienne z kwalifikatorem **`const`** bezpośrednio w pliku nagłówkowym.

[1] Rozszerzenie `.hpp` pochodzi od angielskiego terminu *header file* oznaczającego plik nagłówkowy oraz ++ (plus-plus) z nazwy języka C++.

[2] zob. rozdz. 16 Przestrzenie nazw

Rozdział 4

Biblioteka standardowa – wprowadzenie

Funkcjonalność udostępniana przez język C++ to głównie mechanizmy niskiego poziomu – na przykład sposoby organizacji kodu czy zarządzania pamięcią – które nie przystają do stopnia złożoności tworzonych obecnie systemów informatycznych.

W związku z tym wraz z „jądrem" języka kompilatory C++ udostępniają **bibliotekę standardową** (ang. standard library), rozszerzającą podstawową funkcjonalność języka o często stosowane klasy i funkcje – zaimplementowane właśnie z użyciem podstawowych mechanizmów C++. Co ważne, ta biblioteka programistyczna posiada przymiotnik „standardowa", gdyż stanowi ona część standardu ISO języka C++. Mamy więc gwarancję, że biblioteka standardowa będzie miała taką samą funkcjonalność niezależnie od użytego kompilatora, pod warunkiem jego zgodności ze standardem ISO C++.

Biblioteka standardowa definiuje m.in. kilka podstawowych kontenerów[1] (tj. obiektów służących do przechowywania kolekcji innych obiektów) oraz funkcje do manipulowania ich zawartością i efektywnego przetwarzania ich zawartości, strumienie ułatwiające obsługę operacji wejścia/wyjścia, a także inne często stosowane funkcje (np. operacje matematyczne). Warto podkreślić, że w bibliotece standardowej stosowane są pewne konwencje odnośnie interfejsów klas i funkcji, dzięki czemu tworzą one spójną całość, a korzystanie z nich jest wygodne i intuicyjne.

Stosując funkcjonalności udostępniane przez bibliotekę standardową unikamy „wyważania otwartych drzwi" i przykładowo implementowania własnoręcznie takich abstrakcyjnych typów danych jak tablica dynamiczna czy stos – co nie tylko zabiera sporo czasu, ale również obarczone jest ryzykiem popełnienia błędów w implementacji.

Aby zilustrować wygodę korzystania z biblioteki standardowej posłużymy się przykładem kontenera `std::vector`.

4.1 Kontener `std::vector`

Kontener `std::vector` realizuje funkcjonalność inteligentnej tablicy dynamicznej – takiego ciągu elementów tego samego typu, dla którego liczba elementów może zmieniać się w trakcie działania programu.

Kontener `std::vector` to w istocie **szablon klasy**[2], stąd po nazwie szablonu `std::vector` należy podać w nawiasach trójkątnych nazwę typu elementów przechowywanych w danym obiekcie – przykładowo, `std::vector<int> vi` oznacza, że `vi` to kontener `std::vector` przechowujący obiekty typu `int`. Listing 4.1 demonstruje sposób korzystania z kontenera `std::vector`.

Listing 4.1. Podstawowa funkcjonalność kontenera `std::vector`.

```
#include <cstdlib>
// Aby korzystać z kontenera `std::vector` trzeba dołączyć
//   odpowiedni (standardowy) plik nagłówkowy.
#include <vector>

#include <iostream>

int main() {
```

[1] Wymogi, jakie musi spełniać dany typ, aby mógł być nazwany *kontenerem*, znajdziesz tu.

[2] Szablony zostaną omówione dokładniej w rozdziale 5 Szablony.

```
    // Utwórz pusty kontener umożliwiający przechowywanie elementów typu `int`.
    std::vector<int> vec;

    // Wstawiaj elementy na koniec kontenera `std::vector`.
    vec.push_back(1);
    vec.push_back(5);
    vec.push_back(2);

    // Wyświetl liczbę elementów przechowywanych aktualnie
    //   w kontenerze (metoda niezależna od rodzaju kontenera).
    std::cout << "vec size = " << vec.size() << std::endl;

    // Wyświetl drugi w kolejności element przechowywany
    //   w kontenerze `std::vector`.
    std::cout << vec[1] << std::endl;

    // Wyświetl ostatni element kontenera `std::vector`.
    std::cout << vec.back() << std::endl;

    // Wyświetl wszystkie elementy kontenera z użyciem
    //   konstrukcji "range-based for loop" (rozwiązanie uniwersalne)
    std::cout << "Elements: ";
    for (auto e : vec) {      // równoważne: `for (int e : vec)`
        std::cout << e << " ";
    }
    std::cout << std::endl;

    // Usuń z kontenera wszystkie elementy (metoda uniwersalna).
    vec.clear();

    // Zainicjalizuj kontener `std::vector` umożliwiający przechowywanie
    //   elementów typu `int` listą obiektów stanowiących początkową
    //   zawartość kontenera, tzw. list initialization (rozwiązanie
    //   uniwersalne).
    std::vector<int> vec2{1, 2};

    return EXIT_SUCCESS;
}
```

W powyższym przykładzie wykorzystaliśmy konstrukcję pętli `for` opartej o zasięg (ang. range-based `for` loop) w połączeniu z tzw. zastępczym symbolem specyfikatora typu **`auto`** (omówionym w rozdziale 7.6.1 Zastępczy symbol specyfikatora typu: `auto`), która umożliwia wygodne iterowanie po elementach pewnego zakresu wartości (np. kontenera). Konstrukcja ta została omówiona w rozdziale 7.6.2 Range-based `for` loop. Z kolei obiekt `vec2` został zainicjalizowany z użyciem tzw. **list initialization**[3] (elementy, którymi ma zostać zainicjalizowany obiekt `vec2`, zostały przekazane wewnątrz nawiasów klamrowych w postaci listy obiektów oddzielonych od siebie przecinkami).

Oprócz zademonstrowanej funkcjonalności kontener `std::vector` pozwala m.in. na wstawianie i usuwanie elementów na dowolnej pozycji oraz leksykograficzne porównywanie ze sobą dwóch obiektów typu `std::vector<T>` (gdzie `T` oznacza dowolny wybrany typ elementów) – pełną funkcjonalność kontenera `std::vector` znajdziesz w jego dokumentacji.

W powyższych opisach konsekwentnie stosujemy termin „kontener `std::vector`”, a nie „klasa `std::vector`”, gdyż `std::vector` to *szablon klasy*, a nie zwykła klasa. Funkcjonalność kontenera `std::vector` powinna być identyczna niezależnie od typu przechowywanych w nim elementów. Ponieważ język C++ wymaga określenia konkretnego typu danych w programie, bez użycia szablonów należałoby zdefiniować osobną klasę `std::vector` dla każdego typu elementów – zarówno typów prostych (np. **`int`**, **`double`**), jak i typów złożonych (np. innych klas).

[3] zob. list initialization

Informacja

`std::vector` *to* szablon klasy, *a nie* typ. *Typ otrzymujemy dopiero po podaniu konkretnego parametru szablonu, np.* `std::vector<int>`. *Proces zastępowania parametrów szablonu konkretnymi typami nazywamy* ***konkretyzacją*** *(ang. instantiation) szablonu.*

4.2 Dokumentacja biblioteki standardowej

Podstawą efektywnego posługiwania się elementami biblioteki standardowej jest umiejętność korzystania z jej dokumentacji[4], która zawiera wyczerpujący opis pełnej funkcjonalności udostępnianej przez bibliotekę standardową wraz z przykładami ich użycia.

[4]Na przykład na stronie cppreference.com bądź cplusplus.com.

Rozdział 5

Szablony

Choć w większości podręczników do języka C++ zagadnienie szablonów omawiane jest pod koniec kursu programowania, w niniejszym skrypcie szablony zostały omówione niemal na samym początku, gdyż to na nich opiera się większość podstawowej funkcjonalności biblioteki standardowej (z którą zapoznasz się już w kolejnym rozdziale).

Szablon (ang. template) to element języka C++[1], umożliwiający programowanie generyczne – tworzenie kodu niezależnego od typów, algorytmów oraz struktur danych.

5.1 Programowanie generyczne

Programowanie generyczne (ang. generic programming), inaczej **programowanie uogólnione** paradygmat programowania zgodnie z którym powinna istnieć możliwość pisania fragmentów kodu programu bez wcześniejszej znajomości typów danych wejściowych, na których kod ten będzie operował – np. z wykorzystaniem swego rodzaju *szablonów*. Kod programu stosującego elementy generyczne nie jest kompilowany w dokładnie takiej formie, jak został napisany kod źródłowy, lecz kompilator przekształca szablony do kodu źródłowego (szablon to swego rodzaju „przepis" dla kompilatora służący generowaniu funkcji lub klas o pewnej określonej funkcjonalności).

Prostymi przykładami zastosowania programowania generycznego są kontenery biblioteki standardowej. W statycznie typowanym języku programowania, takim jak C++, normalnie musielibyśmy albo zadeklarować osobne typy kontenerów do przechowywania elementów każdego poszczególnego typu (czyli osobny typ kontenera dla elementów typu `int`, osobny dla elementów `unsigned int` itd.), albo korzystać z kontenera wskaźników na typ `void` (co prowadziłoby do utraty wszelkiej informacji o typie na poziomie kontenera i konieczności jej „ręcznego" odtwarzania po uzyskaniu dostępu do elementów takiego kontenera). Szablony pozwalają uniknąć tego problemu – umożliwiają one definiowanie funkcji i klas w oparciu o parametry, których typ nie jest określony w miejscu definicji. Dopiero w momencie **konkretyzacji szablonu** (ang. template instantiation) kompilator otrzymuje informację, jaki typ został faktycznie użyty jako parametr szablonu – i generuje odpowiednią klasę.

5.2 Szablony funkcji

Nim omówione zostaną szablony funkcji, zwróć uwagę, że język C++ umożliwia dokonywanie **przeciążania funkcji** – czyli deklarowania w tym samym zakresie funkcji o tej samej nazwie, lecz różniących się listą parametrów (czyli liczbą parametrów i/lub ich typem), przy czym przeciążone funkcje mogą mieć identyczne definicje[2] – ponieważ kompilator sam dobierze najlepiej pasującą wersję funkcji o danej nazwie do przekazanych argumentów, stanowi to rodzaj programowania generycznego:

```
#include <cstdlib>
#include <iostream>

int sum(int a, int b) {
    return a + b;
}
```

[1] Inne języki udostępniają podobny mechanizm, lecz pod innymi nazwami – np. *generics* w języku Java.

[2] zob. rozdz. 17.1 Przeciążanie funkcji

```
double sum(double a, double b) {
    return a + b;
}

int main() {

    // Dwa argumenty typu `int` -- wywołaj int sum(int, int).
    std::cout << sum(10, 20) << std::endl;

    // Jeden z argumentów typu `double` -- wywołaj double sum(double, double),
    //   aby uniknąć (stratnej) konwersji.
    std::cout << sum(1, 1.5) << std::endl;

    return EXIT_SUCCESS;
}
```

Funkcja `sum()` mogłaby być przeciążana dla różnych typów i w każdym przypadku miałaby dokładnie takie samo ciało. W takich przypadkach warto skorzystać z **szablonów funkcji** (ang. function templates).

Definiowanie szablonu funkcji różni się tylko tym od definiowania zwykłej funkcji, że umieszczamy na początku definicji słowo kluczowe **`template`**, po którym w nawiasach trójkątnych określamy parametry szablonu:

```
template <template-parameters> function-declaration
```

gdzie `template-parameters` to lista parametrów rozdzielonych przecinkami. Każdy z parametrów składa się ze słowa kluczowego **`class`** albo **`typename`**[3], po którym występuje identyfikator – z identyfikatora tego możemy korzystać w definicji funkcji jak ze zwykłego typu.

Generyczna funkcja `sum()` mogłaby być zdefiniowana następująco:

```
template <class T>
T sum(T a, T b) {
  return a + b;
}
```

przy czym parametru `T` możemy użyć w dowolnym miejscu w definicji szablonu funkcji – jako pełnoprawnego typu.

Aby dokonać konkretyzacji szablonu funkcji, należy wywołać taki szablon z użyciem następującej składni:

```
name <template-arguments> (function-arguments)
```

gdzie `template-arguments` oznacza typy danych użyte do konkretyzacji. Przykładowo, poniższa konkretyzacja

```
// konkretyzacja szablonu funkcji `sum` typem `float`
x = sum<float>(10.0f, 20.0f);
```

spowoduje wygenerowanie przez kompilator poniższej definicji funkcji:

```
float sum(float a, float b) {
  return a + b;
}
```

5.3 Szablony klas

Analogicznie do szablonów funkcji możemy definiować **szablony klas** (ang. class templates), przykładowo:

```
// definicja szablonu klasy
template <class T>
class MyPair {
```

[3] W podstawowych przypadkach możemy stosować te słowa kluczowe zamiennie, w pewnych szczególnych sytuacjach – nie (zob. Stack Overflow).

```
public:
  MyPair(T first, T second) {
    values[0] = first;
    values[1] = second;
  }

  T get_max();
private:
  T values[2];
};

// definiowanie metody poza klasą
template<class T>
T MyPair<T>::get_max() {
  return (values[0] > values[1]) ? values[0] : values[1];
}

// ...

// konkretyzacja szablonu klasy `MyPair` typem `int`
//   => mówimy o typie `MyPair<int>`
MyPair<int> myobject(115, 36);
```

przy czym w definicji metody `get_max()` zapis `MyPair<T>` mówi, że parametr szablonu funkcji (`T`) jest jednocześnie parametrem szablonu klasy `MyPair`.

5.4 W którym miejscu w kodzie definiować funkcje i metody szablonowe?

Kompilator dokonuje konkretyzacji szablonów przed faktyczną kompilacją kodu, przy czym może on dokonać konkretyzacji wyłącznie znając implementację funkcji oraz metod szablonowych (gdyż w ich ciele również trzeba będzie dokonać podstawienia parametru szablonu). Jedynym wygodnym sposobem na spełnienie tego wymogu jest zamieszczenie implementacji funkcji i metod szablonowych w pliku nagłówkowym. W praktyce, dla zwiększenia czytelności, często stosuje się rozbicie kodu na dwa pliki nagłówkowe: jeden zawierający deklarację szablonu, a drugi – definicję (przy czym na końcu pliku z deklaracją dołącza się plik z definicją):

```
// foo.hpp
template <typename T>
struct Foo {
    void do_something(T param);
};

#include "foo.tpp"
```

```
// foo.tpp
template <typename T>
void Foo<T>::do_something(T param) {
    // implementacja
}
```

5.5 Zalety i wady szablonów

Wykorzystanie szablonów w programach pozwala skupić się bardziej na implementacji ogólnej logiki algorytmu niż na rozpatrywaniu tego, „jakiego konkretnie typu dane wejściowe ten algorytm będzie potencjalnie przetwarzał” – w szczególności szablony pozwalają uniknąć duplikacji kodu (w zasadzie identycznego dla różnych typów danych).

Z drugiej strony jednak działanie szablonów w języku C++ przypomina bardzo zaawansowane makra preprocesora – w efekcie kompilator ma duże trudności z wygenerowaniem czytelnych komunikatów dia-

gnostycznych w przypadku błędnego użycia poprawnego szablonu. Większość z nich dotyczy bowiem (poprawnego!) kodu bibliotecznego[4], co wprowadza programistę w konsternację.

[4]Problem ten wynika z faktu, że błędy z reguły dotyczą sposobu użycia szablonu, natomiast diagnostyka włączana jest dopiero po jego konkretyzacji – a więc zasadniczo w innym miejscu niż szablon ten jest faktycznie używany. Stąd wrażenie, że błędy znajdują się w bibliotekach, a nie w programie, który ich błędnie używa.

Rozdział 6

Semantyka wartości a semantyka referencji

Język C++ rozszerza znaną z języka C semantykę wartości i wskaźników o semantykę referencji, a dodatkowo od samego początku swego istnienia wprowadza do wspomnianych semantyk informację o „stałości” (tj. czy możemy daną wartość modyfikować, czy też nie)[1].

Zrozumienie, jak działają wspomniane semantyki, jest kluczowe dla pisania poprawnego i *bezpiecznego* kodu w języku C++.

6.1 Semantyka referencji (do l-wartości)

Referencja (ang. reference) definiuje alternatywną nazwę dla obiektu – czyli **alias** tego obiektu. Mówimy inaczej, że „typ referencyjny «odnosi się» do innego typu”.

Pierwotnie w języku C++ można było utworzyć wyłącznie referencje do nazwanych obiektów, czyli referencje do l-wartości. Standard C++11 wprowadził jednak nowy typ referencji – referencje do obiektów nienazwanych, czyli do r-wartości[2]. W związku z tym obecnie każdorazowo należy określać, czy mówiąc „referencja” mamy na myśli referencję do l-wartości, czy do r-wartości.

Referencję do l-wartości definiujemy poprzez zapisanie deklaratora w formie `T& d`, gdzie `d` to deklarowana nazwa „referencji do l-wartości typu `T`” – przy czym referencję (każdego typu) zawsze należy zainicjalizować w momencie definicji (w języku C++ nie istnieją „puste” referencje):

```
int ival = 1024;
int& ref_val = ival;  // `ref_val` odnosi się do `ival`
int& ref_val2;        // BŁĄD: referencja musi zostać zainicjalizowana
```

Zwróć uwagę, że podczas inicjalizacji „zwykłej” zmiennej (`T t`), wartość obiektu inicjalizującego zostaje skopiowana do tworzonego obiektu. W przypadku inicjalizacji referencji do l-wartości (`T& t_lref`), trwale **wiążemy** (ang. bind) referencję z jej inicjalizatorem – nie ma możliwości „przepięcia” referencji do l-wartości z jednego obiektu na inny.

Informacja

Utworzenie referencji na typ `T` nie tworzy nowego obiektu typu `T`, a jedynie powoduje utworzenie zbliżonego funkcjonalnie do wskaźnika aliasu *na istniejący obiekt typu `T`.*

Różnica względem wskaźników polega na tym, że wskaźnik może być „pusty” (czyli może nie wskazywać na żaden konkretny obiekt), a referencja – nie.

Co więcej, ponieważ referencje nie są obiektami, nie można utworzyć „referencji do referencji” (zapis `T&&` oznacza wspomnianą wcześniej referencję do r-wartości).

Po zdefiniowaniu referencji, wszystkie wykonywane na niej operacje w rzeczywistości odnoszą się do powiązanego obiektu:

```
ref_val = 2;  // przypisz 2 do obiektu powiązanego z `ref_val` (tj. do `ival`)
```

W praktyce semantyka referencji bardzo przypomina semantykę wskaźników[3].

[1] W języku C informację o „stałości” wprowadził dopiero standard C99.

[2] Referencje do r-wartości zostaną dokładniej omówione w rozdziale 12.2 Referencje do r-wartości.

[3] zob. dyskusję Pointers And References Are The Same Thing (wiki.c2.com)

Referencje a funkcje

Referencje często stosuje się jako typ parametrów funkcji lub typ wartości zwracanej – dzięki temu można uniknąć kopiowania (nierzadko dużych) obiektów. Parametry referencyjne pozwalają również modyfikować obiekty przekazane jako argumenty do funkcji – w języku C takie rozwiązanie było możliwe wyłącznie z użyciem wskaźników jako parametrów.

Kiedy przekazywać argument przez wskaźnik, a kiedy przez referencję? W praktyce kluczowa różnica polega na tym, że wskaźnik posiada specjalny „niepoprawny" stan – wskaźnik pusty – podczas gdy referencja nie posiada takiego stanu. Jeśli w Twoim problemie występuje taki „niepoprawny" stan – zastosuj wskaźnik, w przeciwnym razie zastosuj referencję.

6.2 Semantyka wartości

W języku C++, gdy przekazujemy obiekty poprzez wartość, są one *kopiowane*. Jeśli chcemy przechować w programie wartość pola obiektu, powinniśmy przekazać ją przez wartość – gdybyśmy przekazali wskaźnik na takie pole (z pozoru rozwiązanie wydajniejsze, bo unikamy „niepotrzebnego" kopiowania), moglibyśmy w pewnym momencie zostać z „wiszącym" wskaźnikiem (gdy obiekt zawierający wskazywane pole zostanie usunięty).

```
int* foo() {
    int i;
    return &i;  // W praktyce zwraca "wiszący" wskaźnik, gdyż w momencie
                //   opuszczania funkcji lokalna zmienna `i` jest usuwana.
}
```

Podobnie jeśli wewnątrz funkcji zamierzamy modyfikować obiekt przekazany jako argument, przy czym zmiany te nie powinny być widoczne poza funkcją, powinniśmy przekazywać go poprzez wartość.

Dla zainteresowanych...

Korzystanie z semantyki wartości jest kluczowe dla mechanizmu RAII[4], gdyż kopiując obiekt nie tworzymy w programie zależności – w przeciwieństwie do definiowania wskaźników na obiekt albo referencji do obiektu.

6.3 *const correctness*

Źródło: ISO C++: Const Correctness

Termin „const correctness"[5] oznacza umiejętne stosowanie kwalifikatora `const` w celu uniemożliwienia modyfikacji stanu obiektów. Umiejętność tworzenia bezpiecznego kodu – odpornego na przypadkowe modyfikacje obiektów – to jeden z wyznaczników dobrego programisty.

Stosowanie *const correctness* wymaga od Ciebie częstego dekorowania kodu słowami kluczowymi `const`, lecz Twój wysiłek się opłaci – w ten sposób przekazujesz kompilatorowi oraz innym programistom istotne informacje semantyczne odnośnie zasady działania poszczególnych fragmentów kodu. Informacje te pomagają kompilatorowi wychwycić potencjalne błędy, a innym programistom służą za dokumentację.

Najlepiej stosować *const correctness* od samego początku pisania kodu danego programu, gdyż próba jej wprowadzania do istniejącego kodu powoduje efekt kuli śnieżnej: każde dodanie `const` w jednym miejscu powoduje konieczność dodania go w czterech innych miejscach...

6.3.1 *const correctness* a zwykłe bezpieczeństwo typów

Deklarowanie parametrów z kwalifikatorem `const` to po prostu inna forma mechanizmu bezpieczeństwa typów. Możesz traktować typ `const std::string` jako inny typ niż `std::string`, gdyż wariant z `const` nie posiada pewnych operacji zmieniających stan obiektu dostępnych w wariancie bez `const` (np. operatora += oraz każdej innej operacji powodującej zmianę stanu obiektu).

[4] zob. rozdz. 9.6 RAII i cykl życia obiektów

[5] Brak dla niego dobrego polskiego odpowiednika.

6.3.2 Stałe wskaźniki

Informacja

Semantyka wskaźników w języku C++ jest identyczna jak w języku C w standardzie C99 – jednak ponieważ zagadnienie const correctness *jest w języku C++ niezmiernie istotne, raz jeszcze przytoczono stosowne zasady.*

W przypadku wskaźników mamy do czynienia z dwoma elementami, które mogą być stałe (wskazywany obiekt oraz sam wskaźnik), a zatem istnieją cztery możliwe kombinacje użycia **const**:

```
T* p;              // "zwykły" wskaźnik:
                   //   można zarówno zmienić stan wskazywanego obiektu,
                   //   jak i przepiąć wskaźnik na inny obiekt

const T* p;        // wskaźnik na stałą wartość:
                   //   nie można zmienić stanu wskazywanego obiektu,
                   //   natomiast można przepiąć wskaźnik na inny obiekt

T* const p;        // stały wskaźnik:
                   //   można zmienić stan wskazywanego obiektu,
                   //   lecz nie można przepiąć wskaźnika na inny obiekt

const T* const p;  // stały wskaźnik na stałą wartość:
                   //   nie można zmienić stanu wskazywanego obiektu,
                   //   oraz nie można przepiąć wskaźnika na inny obiekt
```

Pamiętaj, że deklaracje wskaźników czytamy od prawej do lewej: **const** `T*` `p` oznacza, że „`p` to wskaźnik na stały typ `T`”.

6.3.3 Stałe referencje

Aby uniemożliwić modyfikację obiektu typu `T` powiązanego z referencją `x`, możemy zadeklarować **stałą referencję do l-wartości** (ang. const l-value reference) na typ `T` w następujący sposób: **const** `T& x`.

Informacja

Programiści C++ zwykli określać „referencję do typu `const` *” w skrócie jako „stałą referencję”. Takie skracanie ma sens, jeśli pamiętasz, że to tylko skrót myślowy.*

Tak naprawdę nie istnieje coś takiego, jak „stała referencja”. Sama referencja nie jest obiektem, zatem nie może być typu **const**. *W szczególności ponieważ „z definicji” nie jesteśmy w stanie powiązać istniejącej referencji z innym obiektem, wszystkie referencje są domyślnie* w pewnym sensie *typu* **const**. *Określenie „stałości” referencji odnosi się zatem wyłącznie do obiektu powiązanego, a nie do samej zmiennej referencyjnej.*

6.3.4 Stałe metody

Stałą metodą (ang. const method) nazywamy taką metodę, która jedynie odczytuje stan obiektu (nie dokonuje żadnych zmian stanu obiektu). Stałą metodę wyróżniamy poprzez dodanie sufiksu **const** tuż po liście parametrów takiej metody:

```
class MyClass {
public:
  void inspect() const;    // ta metoda przyrzeka NIE modyfikować obiektu
  void mutate();           // ta metoda może modyfikować obiekt
};

void user_code(MyClass& changeable, const MyClass& unchangeable)
{
  changeable.inspect();     // OK: nie zmienia obiektu,
                            //   który może być zmieniony
```

```
  changeable.mutate();      // OK: zmienia obiekt,
                            //   który może być zmieniony
  unchangeable.inspect();   // OK: nie zmienia obiektu,
                            //   który nie może być zmieniony
  unchangeable.mutate();    // BŁĄD kompilacji: próba zmiany obiektu,
                            //   który nie może być zmieniony
}
```

Jeśli chcesz posiadać taką metodę, która nie zmienia stanu klasy, a jednocześnie zwraca pole klasy poprzez referencję, najlepiej zwróć wartość tego pola albo poprzez referencję do niemodyfikowalnej l-wartości (`const X& inspect() const`), albo przez wartość (`X inspect() const`):

```
class Person {
public:
  const std::string& get_name() const; // Dobrze: wywołujący nie może
                                        //   zmienić pola `name`
  int get_age() const;                  // Również dobrze: wywołujący nie
                                        //   może zmienić pola `age'
  // ...
};
```

Zastanawiając się nad wyborem sposobu zwracania wartości przez taką metodę „tylko do odczytu” – przez referencję czy przez wartość – pamiętaj, że:

- Pod względem wydajności, zwracanie przez referencję jest (zwykle) znacznie efektywniejsze – nie zachodzi potrzeba tworzenia nowego obiektu. Ma to znaczenie głównie w przypadku złożonych obiektów, o długim czasie konstruowania. Jednak w przypadku prostych typów bądź typów wbudowanych zwracanie przez wartość może okazać równie wydajne, a czasem wręcz wydajniejsze.
- Z punktu widzenia bezpieczeństwa zwracanie kopii jest lepszym rozwiązaniem, gdyż kwalifikator `const` może zostać usunięty przez klienta naszego kodu[6]. Należy mieć to na względzie zwłaszcza tworząc publiczne API, tj. w sytuacji gdy nie mamy pełnej kontroli nad sposobem korzystania ze zwróconego obiektu.
- Zwracając przez referencję bądź wskaźnik musisz brać pod uwagę kwestię cyklu życia [7] powiązanego z nimi obiektu. Przykładowo, jeśli trzymasz zapamiętany zwrócony wskaźnik, a w międzyczasie obiekt wskazywany przestanie istnieć, wskaźnik przestanie być poprawny – co w przypadku odwołania się do niego doprowadzi do niezdefiniowanego działania programu. Zwracając przez wartość nie musisz przejmować się kwestiami cyklu życia obiektu.

6.3.5 Przeciążanie `const`

Przeciążanie `const` (ang. const-overloading) pomaga osiągnąć *const correctness*. O takim rodzaju przeciążeniu mówimy w sytuacji, gdy obiekt posiada dwie (przeciążone) metody – jedną wyłącznie do odczytywania stanu, a drugą również do modyfikacji stanu obiektu – które mają identyczną nazwę oraz identyczny zestaw parametrów, lecz metoda do odczytu jest dodatkowo stała (a metoda do modyfikacji – nie).

Jednym z częstszych przypadków stosowania przeciążania `const` jest operator indeksowania:

```
class T {
public:
  void inspect() const;
  void mutate();
// ...
};

class MyList {
public:
  // ta wersja umożliwia wyłącznie odczyt
  const T& operator[] (std::size_t index) const;
```

[6] W tym celu dokonania takiej konwersji używa się jawnego rzutowania z użyciem operatora `const_cast`.

[7] zob. rozdz. 9.6 RAII i cykl życia obiektów

```
  // ta wersja umożliwia zarówno odczyt, jak i modyfikację
  T&       operator[] (std::size_t index);
// ...
};
```

Wybór wersji operatora przy jego wywołaniu zależy od stałości obiektu klasy `MyList`. Jeśli został on wywołany dla obiektu `MyList` nieposiadającego kwalifikatora **`const`** albo dla referencji do modyfikowalnej l-wartości typu `Mylist`, wówczas wykonana zostanie wersja operatora bez **`const`**:

```
void f(MyList& list) {
  // We wszystkich poniższych przypadkach wywołana zostanie wersja
  //   T& operator[] (unsigned index)

  T x = list[3];      // brak zmiany list[3] - jedynie tworzona jest kopia
  list[3].inspect(); // brak zmiany list[3] - metoda inspect() jest stała

  // Referencja zwracana przez operator[] NIE jest stała, zatem
  //   MOŻNA wywołać metodę modyfikującą tę referencję.
  list[3].mutate();  // OK
}
```

W przypadku, gdy wspomniany obiekt lub referencja posiadają kwalifikator **`const`**, wybrany zostanie wariant operatora posiadający kwalifikator **`const`**:

```
void fc(const MyList& c_list) {
  // We wszystkich poniższych przypadkach wywołana zostanie wersja
  //   const T& operator[] (unsigned index) const

  T x = c_list[3];      // brak zmiany list[3] - jedynie tworzona jest kopia
  c_list[3].inspect(); // brak zmiany list[3] - metoda inspect() jest stała

  // Referencja zwracana przez operator[] jest stała, zatem
  //  NIE MOŻNA wywołać metody modyfikującej tę referencję.
  c_list[3].mutate();  // BŁĄD (na szczęście...)
}
```

Rozdział 7

Biblioteka standardowa

Jak wspomniano w rozdziale 4 Biblioteka standardowa – wprowadzenie, biblioteka standardowa oferuje bogatą funkcjonalność dostępną „od ręki”. W tym rozdziale omówione będą najczęściej stosowane podstawowe elementy biblioteki standardowej.

7.1 Klasa `std::string`

Klasa `std::string` służy do obsługi ciągu jednobajtowych znaków[1].
Oto krótki program demonstrujący sposób korzystania z klasy `std::string`:

Listing 7.1. Podstawowa funkcjonalność klasy `std::string`.

```
#include <cstdlib>
// Pamiętaj o dołączeniu odpowiedniego pliku nagłówkowego
// zawierającego pożądaną funkcjonalność biblioteki standardowej!
#include <string>

#include <iostream>

void old_c_function(const char* str) {
    printf("OLD: %s\n", str);
}

int main() {

    // Zainicjalizuj string literałem łańcuchowym.
    std::string str("Ala");

    // Dodaj elementy na koniec stringa.
    str += " ma " + std::to_string(3);
    str += " kota";

    // Wyświetl string.
    std::cout << str << std::endl;

    // Przekaż do funkcji łańcuch znaków (w stylu C).
    old_c_function(str.c_str());

    // Przypisz do stringa literał łańcuchowy.
    std::string str_c = "Ala";

    return EXIT_SUCCESS;
}
```

Oprócz zademonstrowanej funkcjonalności klasa `std::string` pozwala m.in. wstawianie i usuwanie znaków na dowolnej pozycji, uzyskiwanie podciągu złożonego ze znaków z zadanego zakresu, wyszukiwanie

[1] Na przykład znaków w kodowaniu ASCII.

podciągu, zastępowanie podciągu innym podciągiem – pełną funkcjonalność klasy `std::string` znajdziesz w jej dokumentacji.

7.1.1 Surowe literały łańcuchowe

Czasem przydaje się możliwość uniknięcia interpretowania literałów łańcuchowych (takich jak `\`) lub znaków specjalnych (np. `"`). C++11 umożliwia więc utworzenie **surowych literałów łańcuchowych** (ang. raw string literal):

```
R"(The newline symbol is \n)"
```

W powyższym przykładzie wszystko w obrębie kombinacji nawiasów okrągłych z cudzysłowami `"( )"` jest częścią napisu. Znak „`\`” nie musi być poprzedzony drugim znakiem „`\`” – kombinacja „`\n`” zostanie zapisana z użyciem dwóch znaków, a nie pojedynczego znaku nowej linii (o kodzie ASCII 12). Jedyne ograniczenie w tym przypadku polega na tym, że łańcuch nie może zawierać kombinacji „`)"`”.

Dla zainteresowanych...

Można korzystać z dowolnej niestandardowej kombinacji symboli ograniczających surowy literał łańcuchowy:

```
R"delimiter(The parenthesis is ) )delimiter"
```

W powyższym przykładzie, ciąg „`delimiter(`” zaczyna napis, który kończy się dopiero z chwilą napotkania ciągu „`)delimiter`” – dzięki temu wewnątrz surowego literału możemy korzystać z symboli „`(`” i „`)`”. Zamiast `delimiter` można użyć innego ciągu znaków.

Więcej o surowych literałach łańcuchowych przeczytasz w dokumentacji.

7.2 Iteratory

Choć możemy odwoływać się do elementów pewnych typów obiektów (np. elementów kontenera `std::vector` danego typu lub znaków w obiekcie klasy `std::string`) z użyciem indeksowania:

```
std::string s = "Abc";
char c = s[1];  // dostęp z użyciem indeksowania
```

to biblioteka standardowa udostępnia znacznie bardziej ogólny mechanizm dostępu do elementów zwany **iteratorami**.

W ogólnym ujęciu, **iteratorem** (ang. iterator) jest każdy taki obiekt, który wskazuje na pewien element z *zakresu* elementów (np. z tablicy lub kontenera), i który posiada zdolność *iterowania* po elementach takiego zakresu (czyli „odwiedzania” tych elementów) z użyciem pewnego zbioru operacji – przy czym zbiór taki musi obejmować przynajmniej dwie operacje:

- inkrementacji (`++`) – służy do odwiedzania kolejnego elementu, oraz
- dereferencji (`*`) – służy do uzyskiwania wartości elementu wskazywanego przez iterator.

(Najbardziej oczywistym przykładem realizacji idei iteratora jest wskaźnik udostępniany przez sam język C++.)

Podobnie jak wskaźnik, iterator może być **poprawny** (ang. valid) albo **niepoprawny** (ang. invalid). Poprawny iterator wskazuje na element z zakresu albo na pozycję „tuż za ostatnim elementem”; wszelkie inne iteratory są niepoprawne. *Wszystkie* kontenery oraz klasa `std::string` posiadają zdefiniowane swoje iteratory, podczas gdy indeksowanie wspierane jest tylko przez niektóre z kontenerów.

7.2.1 Typy iteratorów

Każdy kontener `C` posiada własne dwa typy iteratorów:

- `C::iterator` – umożliwia zarówno odczyt, jak i modyfikację wskazywanego elementu
- `C::const_iterator` – umożliwia wyłącznie odczyt wskazywanego elementu

przykładowo

```
// Iterator umożliwiający zarówno odczyt, jak i modyfikację elementów
//   obiektu typu `vector<int>`.
vector<int>::iterator it_vi;
```

```
// Iterator umożliwiający zarówno odczyt, jak i modyfikację znaków
//   w obiekcie typu `string`.
string::iterator it_s;

// Iterator umożliwiający wyłącznie odczyt elementów - "wskazuje" na
//   obiekt typu `const vector<int>`.
vector<int>::const_iterator it_cvi;
```

7.2.2 Uzyskiwanie iteratorów

Aby otrzymać iterator dla danego obiektu, korzystamy zwykle z jego specjalnych metod – `begin()` i `end()`, służących do otrzymania iteratora odpowiednio na początek bądź koniec kolekcji. Metody `begin()` i `end()` dla standardowych kontenerów są przeciążone *względem kwalifikatora* **`const`**, przykładowo (dla kontenera typu `C`):

```
C::iterator       C::begin();
C::const_iterator C::begin() const;
```

Należy zatem pamiętać, że iterator uzyskany za pomocą metod `begin()` i `end()` dla obiektu typu **`const`** nie umożliwia modyfikacji elementów tego obiektu – podobnie jak wskaźnik typu **`const`** nie pozwala na modyfikację obiektu wskazywanego.

Aby zwiększyć bezpieczeństwo kodu, gdy potrzebujemy jedynie odczytywać wartości elementów kolekcji (a nie modyfikować je), rozsądniej jest operować na iteratorach typu `const_iterator` – nawet w przypadku kolekcji zadeklarowanych bez kwalifikatora **`const`**. Dzięki temu kompilator będzie w stanie wychwycić błędy nieuprawnionej modyfikacji. W tym celu korzystamy z metod `cbegin()` i `cend()`, które zwracają iteratory typu `const_iterator`.

Oto przykłady uzyskiwania iteratorów[2]:

```
std::vector<int> v = {1, 2, 3};

std::vector<int>::iterator it_begin = v.begin();
std::vector<int>::iterator it_end = v.end();

std::vector<int>::const_iterator it_cbegin1 = s.begin();
std::vector<int>::const_iterator it_cbegin2 = s.cbegin();
std::vector<int>::const_iterator it_cend1 = s.end();
std::vector<int>::const_iterator it_cend2 = s.cend();
```

Co ważne, w przypadku kontenerów standardowych metoda `end()` zwraca zawsze iterator na fikcyjny element „tuż za końcem zakresu” (ang. „past-the-end” element) – dzięki takiemu rozwiązaniu jesteśmy w stanie wygodnie sprawdzić np. czy przetworzyliśmy już wszystkie elementy.

W przypadku pustego kontenera metody `begin()` i `end()` zwracają ten sam iterator – na fikcyjny element „tuż za końcem zakresu”.

Począwszy od standardu C++11 możliwe jest uzyskiwanie iteratorów w spójny sposób – niezależnie, czy mamy do czynienia z kontenerem, czy z tablicą – za pomocą funkcji zadeklarowanych w pliku nagłówkowym `<iterator>`: `std::begin()` i `std::end()`. Dla instancji klasy lub tablicy `c` funkcje te zwracają odpowiednio:

- `std::begin(c)` – wynik wywołania `c.begin()` w przypadku instancji klasy albo wskaźnik na początek tablicy (w przypadku tablic)
- `std::end(c)` – wynik wywołania `c.end()` w przypadku instancji klasy albo wskaźnik na element tuż za końcem tablicy (w przypadku tablic)

Analogicznie, począwszy od standardu C++14 możliwe jest uzyskiwanie stałych iteratorów w spójny sposób za pomocą funkcji `std::cbegin()` i `std::cend()` zadeklarowanych w pliku nagłówkowym `<iterator>`.

[2] W rozdziale 7.6.1 Zastępczy symbol specyfikatora typu: `auto` poznasz sposób na zapisanie przytoczonych definicji połączonych z inicjalizacją w skróconej formie.

Listing 7.2. Przykład wykorzystania iteratorów.

```
#include <cstdlib>
#include <string>
#include <cctype>

#include <iostream>

int main() {
  std::string s = "abc def";

  for (std::string::iterator it = std::begin(s);
      it != std::end(s) && !isspace(*it); ++it) {
    *it = static_cast<char>(std::toupper(*it));
    // static_cast<char>(...) odpowiada znanemu z języka C
    // rzutowaniu: (char) ...
  }

  std::cout << s << std::endl;

  return EXIT_SUCCESS;
}
```

W dalszej części skryptu do uzyskiwania iteratorów będziemy korzystać właśnie z funkcji `std::begin()`, `std::cbegin()`, `std::end()` i `std::cend()`.

7.2.3 Operacje na iteratorach

Iteratory domyślnie wspierają zaledwie kilka operacji, które zebrano w tabeli 7.1.

Tablica 7.1. Standardowe operacje na iteratorach.

`*iter`	Dereferencja – zwrócenie elementu wskazywanego.[a]
`iter->mem`	Uzyskanie dostępu do składowej `mem` obiektu wskazywanego, równoważne `(*iter).mem`.
`++iter`	Inkrementacja iteratora, aby wskazywał na następny element.[b]
`--iter`	Dekrementacja iteratora, aby wskazywał na poprzedni element.
`iter1 == iter2` `iter1 != iter2`	Porównuje dwa iteratory. Iteratory są równe, jeśli oba wskazują na ten sam element bądź oba wskazują na element „tuż za końcem zakresu”.

[a] Dokonanie dereferencji na niepoprawnym iteratorze lub iteratorze na element „tuż za końcem zakresu” ma niezdefiniowane zachowanie.
[b] Ponieważ iterator „tuż za końcem zakresu” nie wskazuje istniejącego elementu, nie można go inkrementować.

Program 7.2 korzysta z operacji na iteratorach, aby zamienić wszystkie litery w łańcuchu `s` na duże litery – do wystąpienia pierwszego znaku białego bądź końca łańcucha. (Operator **`static_cast`** został omówiony w rozdziale 11.2.1 Operator `static_cast`, natomiast w rozdziale 7.6.2 Range-based `for` loop poznasz sposób na zapisanie użytej w programie pętli w formie skróconej.)

7.2.4 Terminologia: Iteratory a typy iteratorów

Termin „iterator” może być używany w trzech różnych kontekstach:

- jako pewien koncept,
- jako typ iteratora definiowanego przez kontener, lub

- w odniesieniu do konkretnego obiektu.

7.2.5 Operacje powodujące unieważnienie iteratorów

Niektóre operacje na obiekcie, dla którego uzyskaliśmy iterator, powodują jego **unieważnienie** (ang. invalidation) – tak dzieje się przykładowo przy zmianie liczby elementów obiektu typu `std::vector` (np. z użyciem metody `push_back()`).

Dobre praktyki

W przypadku iterowania po elementach kontenera w pętli z użyciem iteratorów nie należy zmieniać liczby elementów w trakcie wykonywania pętli.

7.2.6 Arytmetyka iteratorów

Iteratory dla klasy `std::string` i typu `std::vector` wspierają dodatkowe operacje służące do przemieszczania takiego iteratora o kilka elementów naraz oraz służące do relacyjnego porównywania dwóch iteratorów (zob. tabela 7.2).

Tablica 7.2. Operacje wspierane przez iteratory klasy `std::string` i typu `std::vector`.

`iter + n` `iter - n`	Przemieszczenie iteratora o n pozycji w prawo (lewo).[a] Nowy iterator wskazuje na element w kontenerze albo element „tuż za końcem zakresu”.
`iter1 - iter2`	Zwraca taką liczbę, która po dodaniu do `iter2` spowoduje otrzymanie `iter1`. Iteratory muszą wskazywać na element kontenera bądź na element „tuż za końcem zakresu”.
`>, >=, <, <=`	Operatory relacyjne. Jeden iterator jest mniejszy od drugiego, jeśli wskazuje na obiekt znajdujący się wcześniej w kolekcji. Iteratory muszą wskazywać na element kontenera bądź na element „tuż za końcem zakresu”.

[a] Podobnie wspierane są operacje `iter += n` oraz `iter -= n`.

Poniższy program korzysta z operacji na iteratorach dla typu `std::string`, aby wypisać tylko pierwszą połowę elementów[3]:

Listing 7.3. Przykład wykorzystania arytmetyki iteratorów dla typu `std::string`.

```
#include <cstdlib>
#include <string>

#include <iostream>

int main() {
  std::string s = "abcdef";

  auto it_mid = std::cbegin(s) + s.size() / 2;
  for (auto it = std::cbegin(s); it != it_mid; ++it) {
    std::cout << *it;
  }
  std::cout << std::endl;

  return EXIT_SUCCESS;
}
```

[3] Ściślej, jeśli łańcuch zawiera n znaków wypisanych zostanie $\lfloor n/2 \rfloor$ pierwszych jego znaków.

7.3 Kontenery

Kontener to obiekt–pojemnik przechowujący zbiór innych obiektów (jego elementów). Kontenery są zaimplementowane jako szablony klas, dzięki czemu oferują dużą elastyczność w zakresie wspieranych typów elementów. Kontener zarządza pamięcią dla przechowywanych elementów oraz zapewnia metody służące do uzyskiwania dostępu do elementów – bezpośrednio lub z użyciem iteratorów[4].

Kontenery udostępniają funkcjonalność abstrakcyjnych struktur danych często używanych w programowaniu: tablic dynamicznych (`std::vector`), kolejek (`std::queue`), stosów (`std::stack`), list wiązanych (`std::list`), tablic asocjacyjnych (`std::map`) itd.

Wiele kontenerów oferuje zbliżoną funkcjonalność, a wybór konkretnego typu często zależy głównie od aspektu wydajności pewnych operacji (zwłaszcza w przypadku kontenerów sekwencyjnych). Przykładowo, złożoność obliczeniowa operacji wstawiania elementu w środek ciągu wartości wynosi:

- $O(1)$ przy wstawianiu elementu do obiektu typu `std::list` z użyciem iteratora,
- $O(n)$ przy wstawianiu elementu do obiektu typu `std::vector`.

Pełne zestawienie dostępnych kontenerów i ich funkcjonalności znajdziesz w dokumentacji, poniżej omówiono jedynie wybrane elementy – te szczególnie przydatne i często spotykane.

7.3.1 Kontener `std::array`

Kontener `std::array` realizuje funkcjonalność inteligentnej tablicy statycznej – ciągu elementów tego samego typu, dla którego liczba elementów pozostaje stała w trakcie działania programu.

Oto krótki program demonstrujący sposób korzystania z kontenera `std::array`:

Listing 7.4. Podstawowa funkcjonalność kontenera `std::array`.

```
#include <cstdlib>
// Pamiętaj o dołączeniu odpowiedniego pliku nagłówkowego
// zawierającego pożądaną funkcjonalność biblioteki standardowej!
#include <array>

#include <iostream>

int main() {

  // Zainicjalizuj tablicę 3 elementów typu całkowitego listą wartości.
  std::array<int, 3> arr = { 1, 2, 3 };

  // Zmień wartość 3. elementu.
  arr[2] = 6;

  // Wyświetl liczbę elementów tablicy.
  std::cout << "arr size = " << arr.size() << std::endl;


  // Wyświetl elementy tablicy.
  for (int e : arr) {
    std::cout << e << " ";
  }
  std::cout << std::endl;

  // Utwórz tablicę dwuwymiarową elementów typu całkowitego.
  std::array<std::array<int, 2>, 2> arr2d = {{{8, 2}, {3, 1}}};

  return EXIT_SUCCESS;
}
```

Pełną funkcjonalność kontenera `std::array` znajdziesz w jego dokumentacji.

[4] Wymogi, jakie musi spełniać dany typ, aby mógł być nazwany *kontenerem*, znajdziesz tu.

7.3.2 Kontenery asocjacyjne

W odróżnieniu od kontenerów sekwencyjnych (np. `std::vector`, `std::array`), kontenery asocjacyjne przechowują i udostępniają elementy w oparciu o **klucz** (ang. key). W praktyce najczęściej stosowane klucze to liczby całkowite albo łańcuchy znaków (równoważnie – obiekty klasy `std::string`).

Dla zainteresowanych...

Kluczem w uporządkowanych kontenerach asocjacyjnych (np. `std::map`, `std::set`) może być dowolny typ K, dla którego określona jest relacja „mniejszy niż" i dla którego zachodzą własności:

1. $\forall k_1, k_2 \in K: k_1 < k_2 \Rightarrow \neg k_2 < k_1$
2. $\forall k_1, k_2, k_3 \in K: k_1 < k_2 \wedge k_2 < k_3 \Rightarrow k_1 < k_3$
3. $\forall k_1, k_2 \in K: k_1 \not< k_2 \wedge k_2 \not< k_1 \Rightarrow k_1 \equiv k_2$

Kontener `std::map`

Mapa (in. słownik, tablica asocjacyjna; ang. map, dictionary, associative array) to abstrakcyjny typ danych, który przechowuje pary `(unikatowy klucz, wartość)` i umożliwia dostęp do skojarzonej z kluczem wartości poprzez podanie klucza.

Przykładem wykorzystania mapy będzie przechowywanie par następującej postaci: klucz – numer PESEL, wartość – dane kontaktowe osoby o takim numerze PESEL. O takiej strukturze danych powiemy, że „mapuje numer PESEL na dane kontaktowe".

Do przechowywania par klucz–wartość kontener `std::map` korzysta z szablonu klasy `std::pair` zdefiniowanego w pliku nagłówkowym `<utility>`. Szablon `std::pair` zawiera dwa publiczne pola, `first` oraz `second`, przechowujące dane tworzące parę. Oto przykład tworzenia i korzystania z par:

```
#include <utility>
#include <string>

// ...

std::pair<std::string, std::string> author1("Jan", "Nowak");
auto author2 = std::make_pair("Adam", "Abacki");
author1.first = "Piotr";
```

Pełną funkcjonalność szablonu klasy `std::pair` znajdziesz w jego dokumentacji.

Oto krótki program demonstrujący sposób korzystania z kontenera `std::map`:

Listing 7.5. Podstawowa funkcjonalność kontenera `std::map`.

```
#include <cstdlib>
#include <map>

#include <iostream>

int main() {
  // Mapowanie PESEL -> nr kom.
  //   (obiekt `contact_book` tworzony z użyciem tzw. list initialization)
  std::map<std::string, std::string> contact_book{
      {"85010102756", "+48 600 123 456"},
      {"83010102721", "+48 600 123 789"}
  };

  // Dostęp do wartości poprzez klucz.
  std::cout << contact_book["83010102721"] << std::endl;

  // Zmiana istniejącej wartości.
  contact_book["85010102756"] = "+48 600 123 456";

  // Wstawianie nowej wartości.
  contact_book["75010102721"] = "+48 500 123 456";
```

```
    std::cout << "Po modyfikacjach:\n";
    for (const auto &pair : contact_book) {
        // Składowa `first` pary odpowiada kluczowi, a `second` - wartości.
        std::cout << pair.first << " : " << pair.second << std::endl;
    }

    return EXIT_SUCCESS;
}
```

Pełną funkcjonalność kontenera `std::map` znajdziesz w jego dokumentacji.

Kontener `std::set`

Kontener `std::set` przechowuje uporządkowany zbiór unikalnych kluczy. Operacje przeszukiwania, usuwania i wstawiania mają złożoność logarytmiczną. Kontener `std::set` przydaje się przykładowo w sytuacji, gdy chcemy utworzyć zbiór wszystkich unikalnych słów występujących w danym tekście.

Pełną funkcjonalność kontenera `std::set` znajdziesz w jego dokumentacji.

7.3.3 Inne kontenery

Inne często stosowane typy kontenerów zostały zebrane w tabeli 7.3.

Tablica 7.3. Inne często spotykane typy standardowych kontenerów.

`std::list`	Kontener sekwencyjny realizujący funkcjonalność listy podwójnie wiązanej – umożliwia wstawianie i usuwanie elementów na dowolnym miejscu w stałym czasie oraz pozwala na iterację w obu kierunkach.
`std::stack`	Adapter kontenera[5] (ang. container adaptor) udostępniający funkcjonalność kolejki typu LIFO (ang. last-in first-out), w której elementy są dodawane i usuwane tylko z jednej strony kontenera.
`std::queue`	Adapter kontenera udostępniający funkcjonalność kolejki FIFO (ang. first-in first-out), w której elementy wstawiane są na jeden koniec kontenera, a pobierane z drugiego końca.
`std::tuple`	Kontener realizujący funkcjonalność **krotki**, czyli uporządkowanego ciągu wartości, którego rozmiar nie ulega zmianie. Stanowi uogólnienie szablonu `std::pair` na dowolną liczbę elementów.

Kontener `std::tuple`

Kontener `std::tuple` realizuje funkcjonalność **krotki**, czyli uporządkowanego ciągu wartości, którego rozmiar nie ulega zmianie. Stanowi uogólnienie szablonu `std::pair` na dowolną liczbę elementów.

Do tworzenia krotki służy funkcja `std::make_tuple()`, a wartość i-tego elementu krotki `t` jest uzyskiwania za pomocą funkcji `get<i>(t)`. Do tzw. **rozpakowania krotki** (ang. tuple unpacking) z użyciem pojedynczej instrukcji należy użyć funkcji `std::tie()` (zob. listing 7.6).

Oto krótki program demonstrujący sposób korzystania z kontenera `std::tuple`:

Listing 7.6. Podstawowa funkcjonalność kontenera `std::tuple`.

```
#include <cstdlib>
#include <tuple>

#include <iostream>
```

[5] Adaptery kontenerów to klasy, które „opakowują” podstawowe kontenery w interfejs charakterystyczny dla danego abstrakcyjnego typu danych. Przykładowo, adapter stosu może wykorzystywać obiekt typu `std::vector` i wtedy jego metoda `push()` służąca odkładaniu elementu na stos będzie zaimplementowana z użyciem metody `push_back()` kontenera `std::vector`.

```
int main() {
  // Tworzenie krotki za pomocą `std::make_tuple()`.
  auto t = std::make_tuple(0.5, 1, 'x');

  // Dostęp do wartości poprzez `std::get<i>()`.
  std::cout << std::get<0>(t) << " " << std::get<1>(t) << std::endl;

  double d;
  int i;
  char c;

  std::tie(d, i, c) = t;  // Rozpakowanie krotki z użyciem `std::tie()`.
  std::cout << d << " " << i << " " << c << std::endl;

  return EXIT_SUCCESS;
}
```

Pełną funkcjonalność kontenera `std::tuple` znajdziesz w jego dokumentacji.

Dobre praktyki

*Choć niektórzy korzystają z szablonu klasy `std::tuple` do zwracania kilku wartości (dzięki temu unikają stosowania zwracania wartości z funkcji poprzez **parametry referencyjne**), to jednak zazwyczaj lepszym sposobem na to jest utworzenie pomocniczej struktury danych – w strukturze danych poszczególne pola są nazwane, dzięki czemu wiadomo co one wyrażają (elementy krotki są anonimowe).*

Krotki można stosować w tym kontekście wówczas, gdy zachodzi potrzeba pisania generycznego kodu (np. gdy w hipotetycznym programie mamy kilka funkcji zwracających różne ciągi trzech wartości różnego typu i zachodzi później potrzeba odwrócenia kolejności elementów w ramach każdego takiego ciągu – wówczas nie mamy możliwości zastosowania struktur danych, gdyż nie bylibyśmy w stanie napisać uniwersalnego algorytmu odwracającego).

7.3.4 Kontenery biblioteki standardowej a wskaźniki i referencje

Korzystając z kontenerów biblioteki standardowej należy zwracać uwagę, w jaki sposób przechowują one elementy. Wykonaj kod z przykładu 7.7.

Listing 7.7. Kontener `std::vector` a iterator

```
#include <cstdlib>
#include <vector>
#include <iostream>

int main() {
  std::vector<int> v{1, 2};
  std::vector<int>::iterator it = std::begin(v);

  std::cout << "p = " << *it << std::endl;
  v.push_back(3);
  std::cout << "p = " << *it << std::endl;

  return EXIT_SUCCESS;
}
```

Dlaczego wartość wskazywana przez iterator `it` uległa zmianie – mimo, że nowe elementy dodawane są na koniec wektora?

Kontener standardowy `std::vector` przechowuje elementy w pamięci komputera w postaci tablicy dynamicznej – jako *spójny ciąg*[6]. Ponieważ przekroczony został limit elementów w dotychczas zaalokowanym obiekcie tego kontenera, konieczne było wykonanie *realokacji* – należało przydzielić nowy, większy blok pamięci i *skopiować* do niego elementy ze starego bloku (a następnie zwolnić stary blok). Taka operacja

[6] zob. `std::vector`

sprawia, że wskaźniki, referencje i iteratory odwołujące się do starego miejsca w pamięci *przestają być poprawne*. Podobne problemy wystąpią podczas usuwania elementów z obiektu kontenera `std::vector` – ponieważ kolejne elementy w tym kontenerze są wówczas przesuwane o jedno miejsce „w lewo”, wskaźniki i referencje do elementów znajdujących się wcześniej po elemencie usuwanym przestają być poprawne.

Aby uniknąć podobnych niespodzianek można zastosować kontener standardowy `std::list`, który przechowuje elementy w pamięci komputera w postaci *listy wiązanej* pojedynczo, przy czym każdy element posiada skojarzony wskaźnik do kolejnego elementu, dzięki czemu nie ma potrzeby wykonywania realokacji a jednocześnie usunięcie elementu wymaga jedynie „przepięcia” wskaźników u sąsiadów tego elementu (zob. przykład 7.8).

Listing 7.8. Kontener `std::list` a iterator

```
#include <cstdlib>
#include <list>
#include <iostream>

int main() {
  std::list<int> v{1, 2};
  std::list<int>::iterator it = std::begin(v);

  std::cout << "p = " << *it << std::endl;
  v.push_back(3);
  std::cout << "p = " << *it << std::endl;

  return EXIT_SUCCESS;
}
```

7.4 Algorytmy

Biblioteka standardowa oferuje ponad 100 algorytmów, z których większość służy do wykonywania operacji na zakresach elementów. Podobnie jak kontenery, algorytmy mają spójną architekturę, co ułatwia ich efektywne stosowanie.

Algorytmy nie operują bezpośrednio na kontenerze, zamiast tego operują na zakresie elementów określonym przez dwa iteratory[7] – jeden na początek zakresu, a drugi na element „tuż za końcem”. W związku z tym umożliwiają realizację tylko takich operacji, na jakie pozwala funkcjonalność iteratorów.

Większość algorytmów zdefiniowanych jest w pliku nagłówkowym `<algorithm>`, choć duży zbiór algorytmów numerycznych zdefiniowany jest też w pliku nagłówkowym `<numeric>`.

7.4.1 Przykłady algorytmów

Aby zilustrować działanie algorytmów, w przykładzie 7.9 pokazano sposób korzystania z trzech często stosowanych algorytmów:

- `std::accumulate` służy do sumowania wszystkich elementów z zakresu (można przyjąć arbitralną początkową wartość sumy),
- `std::equal` porównuje parami kolejne elementy z dwóch zakresów i zwraca wartość *prawda*, gdy w obrębie każdej pary elementy są sobie równe,
- `std::find` służy do znajdowania wartości w zakresie elementów, zwraca iterator na pozycję ze znalezioną wartością bądź na element „tuż za końcem”, gdy wartość nie została znaleziona.

Zwróć uwagę, że ponieważ algorytm `std::equal` posługuje się iteratorami, możemy użyć go do porównywania elementów z kontenerów różnych typów. Co więcej, porównywane elementy również nie muszą być tego samego typu – wystarczy, że da się je ze sobą porównywać za pomocą operatora ==.

7.5 Operacje wejścia/wyjścia w oparciu o strumienie

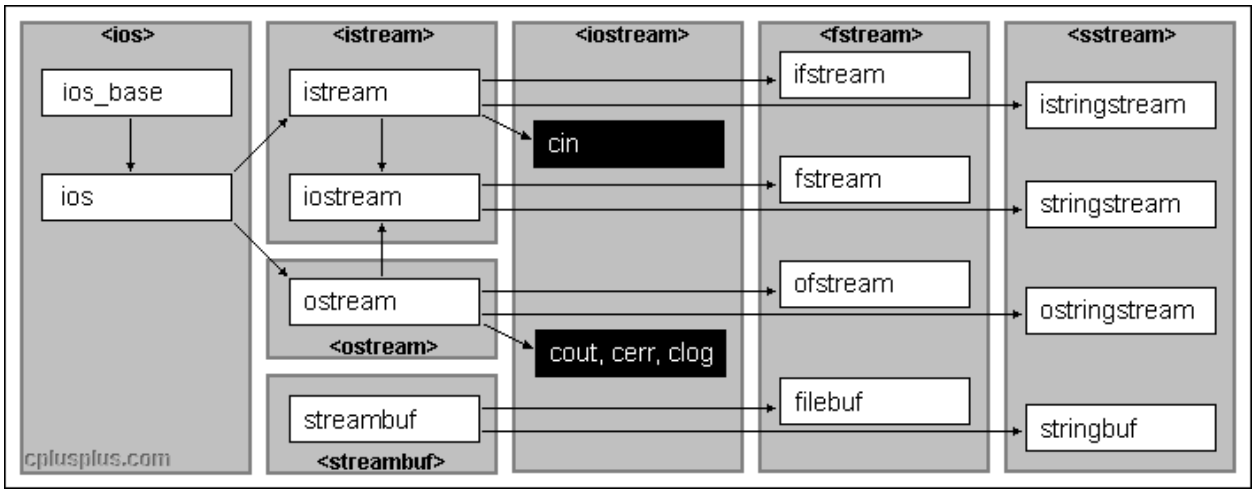
Jądro języka C++ nie posiada wbudowanej obsługi wejścia/wyjścia (we/wy; ang. input/output, I/O) – taką funkcjonalność udostępnia jednak standardowa biblioteka strumieni wejścia/wyjścia.

[7] zob. rozdz. 7.2 Iteratory

Listing 7.9. Przykład użycia algorytmów.

```
#include <algorithm>
// ...

// Posumuj wszystkie elementy wektora, początkowa wartość sumy wynosi 0.
std::vector<int> vec1{1,2,3};
int sum = std::accumulate(std::cbegin(vec1), std::cend(vec1), 0);


// (obiekty `str1` i `str2` tworzone z użyciem tzw. list initialization)
std::vector<std::string> str1{"cat", "ham", "house"};
std::list<const char*> str2{"dog", "pet"};

// Kolekcja str2 powinna zawierać co najmniej tyle elementów, co str1.
bool is_equal = std::equal(std::cbegin(str1), std::cend(str1),
                    std::cbegin(str2));

std::vector<int> vec2{0, 1, 2, 3, 4};
int n = 5;
auto result = std::find(std::begin(vec2), vec2.end(), n);

if (result != vec2.end()) {
    std::cout << "v contains: " << n << '\n';
} else {
    std::cout << "v does not contain: " << n << '\n';
}
```

Biblioteka ta opiera się na pojęciu **strumienia** (ang. stream) – „abstrakcyjnego urządzenia wejścia/wyjścia", który stanowi odpowiednio źródło bądź miejsce docelowe ciągu bajtów. Zastosowanie wspomnianej abstrakcji oraz technik programowania obiektowego umożliwiło opracowanie spójnego interfejsu biblioteki strumieni, pozwalającego na wykonywanie operacji zapisu bądź odczytu z użyciem tego samego zestawu wysokopoziomowych operacji zarówno dla plików na dysku (w tym urządzeń peryferyjnych, np. klawiatury czy „konsoli")[8], jak i danych znajdujących się w pamięci komputera.



Rysunek 7.1. Hierarchia klas biblioteki `iostream`. (Źródło: cplusplus.com)

Najistotniejsze dla nas klasy w hierarchii klas biblioteki we/wy (Rys. 7.1), służące do obsługi strumieniu jednobajtowych znaków typu `char` (np. w kodowaniu ASCII), to:

- `std::istream` – służy do obsługi strumieni wejściowych, definiuje przeciążony operator `>>`, umożliwia odczytywanie danych ze strumienia zgodnie z zadanym formatowaniem;

[8] Dzięki zainstalowanym na komputerze sterownikom urządzeń system operacyjny komunikuje się z urządzeniami peryferyjnymi (np. klawiaturą, konsolą) i ogólnie portami komputera jak przez zwykły plik.

- `std::ostream` – służy do obsługi strumieni wyjściowych, definiuje przeciążony operator <<, umożliwia formatowanie danych umieszczanych w strumieniu.

Klasy znajdujące się niżej w hierarchii udostępniają pełną funkcjonalność „rodziców", a dodatkowo ją rozszerzają o własne pola i metody[9].

7.5.1 Operator << i operator >>

Operator << i operator >> służą odpowiednio do umieszczania danych w strumieniu oraz pobierania danych ze strumienia.

Ponieważ zasada działania obu operatorów jest bardzo podobna, omówiona zostanie na przykładzie operatora <<, gdyż będziesz się z nim stykać o wiele częściej (w swoich programach będziesz głównie wypisywać na konsolę, a nie wczytywać z niej).

Operator << przyjmuje dwa operandy – lewym operandem musi być strumień wyjściowy (obiekt klasy `std::ostream`), prawy operand to wartość do umieszczenia w strumieniu bądź manipulator strumienia[10] – i zwraca referencję na obiekt będący jego lewym operandem.

Rozważ poniższy przypadek użycia operatora <<:

```
std::cout << "Hello world!" << std::endl;
```

który jest równoważny następującym dwóm instrukcjom:

```
std::cout << "Hello world!";
std::cout << std::endl;
```

Takie **łączenie operatorów** (ang. operator chaining) jest możliwe wyłącznie dlatego, że operator << zwraca referencję na strumień wyjściowy – instrukcję z dwoma operatorami << możemy zapisać równoważnie jako:

```
(std::cout << "Hello world!") << std::endl;
```

Operator >> działa analogicznie, z tym że lewym operandem jest obiekt klasy `std::istream`, a prawy określa miejsce umieszczenia wartości pobranej ze strumienia (bądź manipulator).

7.5.2 Co to jest bufor?

Składową każdej klasy operującej na strumieniach jest **bufor** danych (ang. buffer), zapewniający wydajną realizację operacji we/wy niskiego poziomu dla niesformatowanego we/wy do lub z danego urządzenia we/wy – strumień stanowi rodzaj inteligentnego opakowania na bufor. Z buforem wiąże się pojęcie mechanizmu **buforowania** (ang. buffering): przy operacjach zapisu do urządzenia dane najpierw są zbierane w buforze wyjściowym, a dopiero gdy uzbiera się ich odpowiednia ilość są zbiorczo przesyłane do urządzenia. Podobnie w przypadku operacji odczytu zwykle pobierana jest większa porcja danych (np. linia tekstu), które to dane umieszczane są w buforze wejściowym.

Gdy podczas wypisywania danych na standardowe wyjście chcemy wyświetlać dane w konsoli, zwykle zależy nam, aby efekt był od razu widoczny – w związku z tym chcemy uniknąć sytuacji, gdy wypisywany komunikat „utknie" w buforze. W tym celu po umieszczeniu komunikatu w strumieniu możemy **opróżnić** (ang. flush) bufor wywołując dla obiektu strumienia metodę `flush()`. Co istotne, wywołanie dla strumienia funkcji `std::endl` powoduje nie tylko umieszczenie w strumieniu znaku nowej linii (`\n`), lecz również jednoczesne opróżnienie bufora. Z kolei w przypadku zapisu do strumieni skojarzonych z plikami na dysku zwykle chcemy korzystać z buforowania, ze względu na większą wydajność takiego rozwiązania[11] – w tym przypadku należy unikać przechodzenia do nowej linii za pomocą wywoływania funkcji `std::endl`, a zamiast tego powinno się umieszczać w strumieniu po prostu znak nowej linii (`\n`).

[9] Sposoby rozszerzania funkcjonalności klasy – w tym tworzenie podobnych hierarchii z użyciem mechanizmu dziedziczenia – poznasz w rozdziale 8 Klasy: Rozszerzanie funkcjonalności.

[10] Manipulatory zostały omówione w rozdziale 7.5.4 Manipulatory strumieni

[11] Operacje dostępu do dysku twardego magnetycznego są wolne w porównaniu do szybkości wykonywania operacji na danych w pamięci RAM.

7.5.3 Standardowe strumienie wejścia i wyjścia

Biblioteka standardowa definiuje dwa globalne obiekty służące do obsługi odpowiednio bufora wejściowego i wyjściowego:

- `std::cin` – kontroluje dane spływające z bufora standardowego strumienia wejściowego (domyślnie bufor ten powiązany jest z klawiaturą), oraz
- `std::cout` – kontroluje dane napływające do bufora standardowego strumienia wyjściowego (domyślnie bufor ten powiązany jest z konsolą programu).

7.5.4 Manipulatory strumieni

Manipulatory to funkcje pomocnicze, które umożliwiają kontrolę strumieni wejścia/wyjścia za pomocą operatorów << i >>. Kilka najczęściej stosowanych manipulatorów zostało zebranych w tabeli 7.4, a ich użycie ilustruje przykład 7.10. Zestawienie wszystkich manipulatorów znajdziesz w dokumentacji.

Tablica 7.4. Przykłady często stosowanych manipulatorów strumieni wejścia/wyjścia.

`std::setprecision(int n)`	Ustawia precyzję wstawianych/odczytywanych liczb na n pozycji.
`std::setw(int n)`	Ustawia szerokość pola na n pozycji.
`std::hex`	Ustawia typ reprezentacji liczb całkowitych na reprezentację szesnastkową.

Listing 7.10. Przykład użycia manipulatorów strumieni wejścia/wyjścia.

```
#include <cstdlib>
#include <iostream>
#include <sstream>
#include <iomanip>

int main() {
  // Ustaw precyzję liczby na dwie pozycje.
  std::cout << std::setprecision(2) << 5.25 << std::endl << std::endl;

  // Wypisz liczbę w polu o stałej szerokości (wyrównanie do prawej).
  for (int i = 9; i <= 10; i++) {
    std::cout << std::setw(3) << i << std::endl;
  }
  std::cout << std::endl;

  std::istringstream iss("FF");
  int i_hex;
  // Odczytaj liczbę w postaci szesnastkowej.
  iss >> std::hex >> i_hex;
  std::cout << "Odczytana wartosc (dziesietnie): " << i_hex;

  return EXIT_SUCCESS;
}
```

Wynik działania powyższego programu:

```
5.2

  9
 10

Odczytana wartosc (dziesietnie): 255
```

Przywracanie stanu formatowania strumienia

Niektóre manipulatory strumieni (np. `std::hex`) odnoszą się nie do najbliższej wartości, a do wszystkich kolejnych wartości wstawianych bądź pobieranych ze strumienia. Ponieważ kolejni użytkownicy strumienia mogą nie być świadomi takich „trwałych" zmian stanu formatowania strumienia, do dobrych praktyk programistycznych należy zostawienie strumienia w zastanym stanie:

```
#include <cstdlib>
#include <iostream>
#include <iomanip>

int main()
{
   int i = 1000;

   std::cout << "Print int in original format: " << i << std::endl;

   // Zapamiętaj początkowy stan flag formatowania strumienia std::cout.
   std::ios_base::fmtflags original_format = std::cout.flags();

   std::cout << std::hex << std::showbase;

   std::cout << "Print int in a modified format: " << i << std::endl;

   // Przywróć początkowy stan flag formatowania strumienia std::cout.
   std::cout.flags(original_format);

   std::cout << "Print int in original format again: " << i << std::endl;

   return EXIT_SUCCESS;
}
```

7.5.5 Strumienie dla łańcuchów znaków

Operacje na strumieniach dla łańcuchów znaków przydają się w dwóch sytuacjach:

- gdy chcemy otrzymać sformatowany łańcuch znaków,
- gdy chcemy dokonać parsowania łańcucha znaków.

W programie często przydaje się możliwość wypisania tekstowej reprezentacji pewnej struktury danych. Przykładowo, tekstowa reprezentacja struktury wyrażającej punkt na płaszczyźnie o współrzędnych będących liczbami całkowitymi, tj. punkt o współrzędnych $(x, y) \in \mathbb{C}^2$, może mieć postać

```
(x=5, y=3)
```

dla punktu o współrzędnych $(5, 3)$.

Funkcjonalność taka powinna być zawarta w osobnej funkcji, która otrzymuje obiekt albo uchwyt do obiektu (wskaźnik albo referencję) i zwraca obiekt typu `std::string` zawierający tekstową reprezentację argumentu tej funkcji:

Listing 7.11. Prototyp funkcji zwracającej tekstową reprezentację obiektu.

```
#include <string>

struct Point2d {
  int x;
  int y;
};

std::string to_string(const Point2d& p);
```

W jaki sposób zaimplementować funkcję `to_string()`? Najwygodniej – z użyciem klasy `std::ostringstream`.
Typowy cykl pracy z klasą `std::ostringstream` wygląda następująco:

(1) Utwórz obiekt klasy `std::ostringstream`.
(2) Umieść dane w buforze obiektu klasy `std::ostringstream` korzystając z przeciążonego operatora `<<` (analogicznie do umieszczania danych w obiekcie `cout`).
(3) Wywołaj na obiekcie klasy `std::ostringstream` metodę `str()`, aby pobrać reprezentację jego bufora w postaci obiektu klasy `std::string`.

W związku z tym implementacja funkcji `to_string()` może mieć poniższą postać:

Listing 7.12. Implementacja funkcji zwracającej tekstową reprezentację obiektu.

```
#include <sstream>

// ...

std::string to_string(const Point2d& p) {
  // (1) Utwórz obiekt klasy `std::ostringstream`.
  std::ostringstream oss;
  // (2) Umieść dane w buforze obiektu klasy `std::ostringstream`.
  oss << "(x=" << p.x << ", y=" << p.y << ")";
  // (3) Pobierz reprezentację bufora obiektu klasy `std::ostringstream`
  //     w postaci obiektu klasy `std::string`.
  return oss.str();
}
```

a jej użycie może być następujące:

Listing 7.13. Przykład użycia funkcji `to_string(Point2d)`.

```
#include <iostream>

// ...

int main() {
  std::cout << to_string(Point2d{-1, 2}) << std::endl;

  return EXIT_SUCCESS;
}
```

Analogicznie, do parsowania łańcucha znaków służy klasa `std::istringstream` – po wstawieniu łańcucha znaków do bufora obiektu klasy `std::istringstream` można dokonać jego parsowania za pomocą przeciążonego operatora `>>`.

Klasa `std::stringstream` łączy funkcjonalność obu powyższych klas.

Listing 7.14. Przykład użycia manipulatorów strumieni w połączeniu z obiektami z pliku nagłówkowego `sstream`.

```
#include <cstdlib>
#include <iostream>
#include <sstream>
#include <iomanip>

int main() {

  std::ostringstream oss;
  // Umieść sformatowane dane w buforze strumienia.
  oss << "Wynik wynosi " << std::fixed << std::setprecision(3) << 5.25;
  std::string s = oss.str();
  std::cout << s << std::endl;

  std::istringstream iss("Czy 30.70 = 30.7?");
  double x1 = 0, x2 = 0;
  // Pomiń znaki w buforze do wystąpienia spacji (ale nie
  // więcej niż pięć znaków).
  iss.ignore(5, ' ');
```

```
  iss >> x1;
  iss.ignore(5, '=');
  iss >> x2;
  std::cout << ((x1 == x2) ? "Tak." : "Nie") << std::endl;

  return EXIT_SUCCESS;
}
```

Teoretycznie podobne operacje można realizować bez użycia strumieni (np. dodając kolejne sformatowane elementy bezpośrednio do obiektu klasy `std::string`), lecz strumienie zapewniają znacznie lepszą wydajność i większą funkcjonalność takich operacji.

7.5.6 Strumienie dla plików

Wsparcie dla wysokopoziomowych operacji obsługi strumieni jednobajtowych znaków (np. w kodowaniu ASCII) zapewniają klasy: `std::ifstream` (tylko odczyt), `std::ofstream` (tylko zapis), oraz `std::fstream` (zapis i odczyt).

7.5.7 Stan strumienia

Z pracą z wejściem/wyjściem nieodłącznie związane jest ryzyko wystąpienia błędów (np. żądany plik nie istnieje na dysku), dlatego klasy strumieni definiują metody i flagi, które pozwalają sygnalizować i zmieniać stan strumienia.

Jako przykład błędu wejścia, rozważ poniższy przykład:

```
int ival;
std::cin >> ival;
```

Jeśli na standardowe wejście podamy ciąg znaków „Boo”, odczyt się nie powiedzie, gdyż operator wejścia oczekuje ciągu cyfr, a otrzymuje znak „B”. W efekcie obiekt `std::cin` zostanie wprowadzony w stan błędu. Podobny stan błędu wystąpi, gdy podczas odczytu ze strumienia natrafimy na znak **końca pliku** (ang. end-of-file).

Gdy błąd wystąpi, wszelkie dalsze operacje we/wy na takim strumieniu zakończą się niepowodzeniem – dlatego dobrze napisane programy powinny sprawdzać, w jakim stanie znajduje się strumień, przed próbą skorzystania z takiego strumienia. Najprostszy sposób na sprawdzenie stanu strumienia, to użycie go jako warunku:

```
while (std::cin >> word) {
  // ok: operacja odczytu zakończyła się powodzeniem...
}
```

Pętla while sprawdza, czy stan strumienia zwróconego przez wyrażenie >> wciąż jest poprawny.

Powyższa konstrukcja pozwala stwierdzić jedynie ogólnie „czy wystąpił *jakiś* błąd”, natomiast nie umożliwia uzyskania informacji o tym, jaki to był dokładnie błąd – strumienie zawierają jednak szereg wyspecjalizowanych metod, pozwalających na taką szczegółową diagnostykę (zob. std::ios: State flag functions (cplusplus.com)).

Aby po wystąpieniu błędów móc znów korzystać ze strumienia, musimy „wyczyścić” jego stan – np. za pomocą metody `clear()`, która wyłącza wszystkie flagi błędów[12] strumienia:

```
std::cin.clear();          // wyczyść flagi błędów
process_input(std::cin);   // ponownie użyj strumienia
```

7.6 C++11 a biblioteka standardowa

Opisane niżej konstrukcje wprowadzone w standardzie C++11 nie są ściśle powiązane z biblioteką standardową, jednak są szczególnie użyteczne podczas korzystania z funkcjonalności udostępnianych przez bibliotekę standardową.

[12] zob. `std::ios::rdstate`

7.6.1 Zastępczy symbol specyfikatora typu: `auto`

Począwszy od C++11, jeżeli kompilator jest w stanie określić „podstawowy” typ zmiennej w trakcie jej inicjalizacji – nie musimy go jawnie określać, możemy skorzystać z **zastępczego symbolu specyfikatora typu** (ang. placeholder type specifier) stosując słowo kluczowe `auto`:

```
int x = 1;
auto y = x; // `y` będzie typu int
auto c = 'a'; // `c` będzie typu char
```

Mechanizm ten w znaczny sposób ułatwia używanie klas szablonowych – zamiast używać poniższej konstrukcji:

```
const std::vector<int> vi = {1, 2, 3};
std::vector<int>::const_iterator ci = std::cbegin(vi);
```

możemy zadeklarować iterator w następujący sposób:

```
const std::vector<int> vi = {1, 2, 3};
auto ci = std::cbegin(vi);
```

Ważne

Specyfikator `auto` nie uwzględnia referencji oraz kwalifikatora `const` przed typem (kwalifikator `const` w przypadku wskaźnika na typ `const` – `const T*` zostanie zachowany), a w pewnych przypadkach także wskaźników[a] (dlatego, dla uniknięcia błędów, chcąc otrzymać wskaźnik lepiej zawsze pisać `auto*`):

```
int x = 1;

int& y = x;
auto r1 = y;     // `r1` będzie typu `int` (a nie `int&`)
auto& r2 = y;    // `r2` będzie typu `int&`

auto p1 = &x;    // `p1` będzie typu `int*` - NIEZALECANE
auto* p2 = &x;   // `p2` będzie typu `int*` - zalecane

const int z = 0;
auto q = z;    // `q` będzie typu `int` (a nie `const int`)
```

[a] zob. przykład takiej sytuacji

Korzystanie ze specyfikatora `auto` wymaga szczególnej ostrożności w przypadku pracy z iteratorami – ze względu na przeciążenie metod `begin()` i `end()` (względem kwalifikatora `const` oraz typu zwracanego):

```
#include <cstdlib>
#include <iostream>
#include <vector>
#include <algorithm>

void foo(int& i) { ++i; }
void cfoo(const int& i) { std::cout << i << std::endl; }

int main(void) {
   std::vector<int> v = {1, 2, 3};

   auto it1_begin = std::begin(v);
   auto it1_end = std::end(v);
   // Iteratory `it1_begin` i `it1_end` są typu `iterator`, gdyż
   //   są potem wykorzystywane w wywołaniu funkcji foo(), która
   //   modyfikuje argument.
```

```
    std::for_each(it1_begin, it1_end, foo);

    auto it2_begin = std::begin(v);
    auto it2_end = std::end(v);
    // Iteratory `it2_begin` i `it2_end` są typu `const_iterator`,
    //   gdyż są potem wykorzystywane w wywołaniu funkcji cfoo(),
    //   która nie modyfikuje argumentu.
    std::for_each(it2_begin, it2_end, cfoo);

    return EXIT_SUCCESS;
}
```

Aby mieć gwarancję, że dany iterator będzie typu `const_iterator`, lepiej jest jawnie używać metod `cbegin()` i `cend()` (tam, gdzie to konieczne):

```
std::vector<int> v = {1, 2, 3};

auto it_cbegin = std::cbegin(v);
auto it_cend = std::cend(v);
// Iteratory `it_cbegin` i `it_cend` są ZAWSZE typu `const_iterator`.
```

Ważne

Słowo kluczowe **auto** funkcjonowało w języku C++ od samego początku istnienia języka[13], jednak C++11 zmienił jego znaczenie – **auto** przestało oznaczać obiekt o automatycznym typie przechowywania, natomiast zaczęło oznaczać obiekt o typie *wywnioskowanym przy jego inicjalizacji*. W celu uniknięcia nieporozumień stare znaczenie **auto** zostało usunięte z C++11.

7.6.2 Range-based `for` loop

Konstrukcja **range-for** pozwala na iterowanie po obiektach realizujących koncepcję tzw. **zakresu** (ang. range), czyli wszystkich obiektach „iterowalnych” – w szczególności po tablicach, obiektach `std::string`, kontenerach biblioteki standardowej itp.:

```
int tab[3] = {1, 2, 3};
for (auto x : tab) {
  std::cout << x << std::endl;
}

void print_all(const std::vector<T>& vi) {
  for (const auto& elem : vi) {
    std::cout << elem << std::endl;
  }
}
```

Pierwsza sekcja takiej pętli **for** (przed dwukropkiem) definiuje zmienną, która będzie użyta do iterowania po zakresie. Zmienna ta, tak jak zmienne w zwykłej pętli **for**, ma zasięg ograniczony do zasięgu pętli. Druga sekcja (po dwukropku), reprezentuje zakres, po którym iterujemy.

Przykładowo, wynikiem wykonania poniższego programu:

```
#include <cstdlib>
#include <iostream>
#include <vector>

template<class T>
void print_all(const std::vector<T>& vi) {
    for (const auto& elem : vi) {
        std::cout << elem << ' ';
    }
```

[13] Słowo kluczowe **auto** wywodzi się z czasów jeszcze przed wprowadzeniem standardu ANSI C

```
}


int main() {
    int tab[3] = {1, 2, 3};
    for (auto x : tab) {
        std::cout << x << ' ';
    }
    std::cout << std::endl;

    std::vector v = {5, 6, 7};
    print_all(v);

    return EXIT_SUCCESS;
}
```

będzie

```
1 2 3
5 6 7
```

Zwróć uwagę, że zmienna użyta w pętli wyraża kolejne *elementy* z zakresu (a nie indeksy, wskaźniki, czy iteratory).

Logika poniższej schematycznej konstrukcji *range-for*:

```
for (E elem : c) {
  /* ... */
}
```

jest wewnętrznie realizowana z użyciem iteratorów `begin()` i `end()` – *w przybliżeniu*[14] można ją zapisać w poniższej formie:

```
{
  for (auto __begin = std::begin(c), __end = std::end(c);
       __begin != __end; ++__begin) {
    E elem = *__begin;
    /* ... */
  }
}
```

Teraz widzisz, dlaczego podczas definiowania własnych klas zawierających metody `begin()` i `end()` tak ważne jest ich przeciążanie względem **`const`** – umieszczenie jedynie wersji bez **`const`** (i z iteratorem typu `iterator`) spowoduje błąd kompilacji poniższego programu:

```
#include <cstdlib>
#include <cstdlib>
#include <vector>

class Cls {
public:
   std::vector<int>::iterator begin() { return std::begin(v_); }
   std::vector<int>::iterator end() { return std::end(v_); }

private:
   std::vector<int> v_;
};

int main() {
   const Cls c;

   for (int& elem : c) {  // BŁĄD: Klasa `Cls` posiada wyłącznie metody
```

[14] Dokładniejszą implementację znajdziesz tu.

```
                        //   `begin()` i `end()` zwracającą iterator
                        //   umożliwiający modyfikację elementów.
        /* ...*/
    }

    return EXIT_SUCCESS;
}
```

7.6.3 Funkcje wyższego rzędu

Zdarza się, że potrzebujemy być w stanie zmienić fragment zachowania funkcji, bądź chcielibyśmy móc „wygenerować" funkcję. W takich przypadkach korzystamy z **funkcji wyższego rzędu** (ang. higher order functions), czyli funkcji przyjmujących inne funkcje jako parametry, bądź zwracających funkcję.

Często stosowanymi funkcjami wyższego rzędu są takie algorytmy biblioteki standardowej, jak przykładowo `std::find_if()` lub `std::transform()` (zob. przykład 7.15).

Listing 7.15. Przykłady funkcji wyższego rzędu z biblioteki standardowej.

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

bool is_odd(int i) {
  return ((i % 2) == 1);
}

char all_upper(char c) {
  return static_cast<char>(std::toupper(c));
}

int main () {
  std::vector<int> v1 = {1, 2, 3};
  // Ostatni parametr `find_if` to funkcja.
  auto it = std::find_if(std::begin(v1), std::end(v1), is_odd);
  std::cout << "The first odd value is " << *it << '\n';

  std::string s("hello");
  // Wywołaj funkcję `all_upper()` z każdym elementem z zakresu
  // od `s.begin()` do `s.end()`, kolejne wyniki zapisz w zakresie
  // zaczynającym się od `s.begin()` ("w miejscu").
  // Ostatni parametr `transform` to uchwyt do funkcji!
  std::transform(std::begin(s), std::end(s), std::begin(s), all_upper);
  std::cout << s << ':';

  // Wywołaj funkcję `all_upper()` z każdym elementem z zakresu
  // od `s.begin()` do `s.end()`, kolejne wyniki zapisz w zakresie
  // zaczynającym się od `s.begin()` ("w miejscu").
  // Ostatni parametr `transform` to uchwyt do funkcji!
  std::transform(std::begin(s), std::end(s), std::begin(s), all_upper);

  std::vector<int> v2 = {4, 1, 8};
  std::vector<int> vm(v1.size());
  // Wywołaj funkcję zwracającą większą z dwóch wartości dla każdej
  // pary odpowiadających sobie elementów z zakresów:
  //   (1) od `v1.begin()` do `v1.end()` oraz
  //   (2) od `v2.begin()` [do `v2.begin() + (s.end() - s.begin())].
  // Kolejne wyniki zapisz w zakresie zaczynającym się od `vm.begin()`.
  std::transform(std::begin(v1), std::end(v1), std::begin(v2), std::begin(vm),
    [](int a, int b){ return std::max(a, b); });
```

```
  return EXIT_SUCCESS;
}
```

Wyrażenie lambda

Wyrażenie lambda (ang. lambda expression), w skrócie **lambda**, to wygodny sposób na definiowanie anonimowych obiektów funkcyjnych w miejscu, gdzie są od razu wywoływane bądź przekazywane jako argument do funkcji. Lambdy zwykle składają się z jednej instrukcji, a ich użycie pozwala zwiększyć czytelność kodu (nie trzeba niepotrzebnie definiować funkcji w innym – często odległym – miejscu).

Ogólna postać lambdy jest następująca:
`[capture_list](parameter_list) { lambda_body }`
gdzie:

- `capture_list` – lista zmiennych zdefiniowanych w zakresie zawierającym lambdę, do których lambda potrzebuje dostępu (zob. niżej)
- `parameter_list` – lista parametrów lambdy (analogicznie jak w przypadku „zwykłych” funkcji)
- `lambda_body` – ciało lambdy złożone z sekwencji instrukcji (analogicznie jak w przypadku „zwykłych” funkcji)

Zwróć uwagę, że nie musisz określać wprost typu wartości zwracanej.

Korzystając z notacji lambda, przykład 7.15 możemy zapisać następująco:

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

int main () {
  std::vector<int> myvector = {1, 2, 3};
  auto it = std::find_if(std::begin(myvector), myvector.end(),
      [](int i) { return ((i % 2) == 1); });
  std::cout << "The first odd value is " << *it << '\n';

  std::string s("hello");
  std::transform(std::begin(s), s.end(), std::begin(s),
      [](char c) { return std::toupper(c); });
  std::cout << s << ':';

  return EXIT_SUCCESS;
}
```

W powyższym przykładzie zapis:
`[](int i) { return ((i % 2) == 1); }`
oznacza, że:

- `[]` – lambda nie wymaga dostępu do zmiennych z zakresu (zob. niżej)
- `(int i)` – lambda wymaga podania jednego argumentu typu `int`
- `{ return ((i % 2) == 1); }` – ciało lambdy składa się z jednej instrukcji (`return`)

Aby wewnątrz lambdy korzystać ze zmiennych lokalnych zadeklarowanych w tym samym zakresie, zamiast zapisu `[] ...` należy użyć odpowiednio zapisu:

- `[var_name] ...` dla przekazywania przez wartość albo
- `[&var_name] ...` dla przekazywania przez referencję.

Obie formy ilustruje przykład 7.6.3.

```
#include <cstdlib>
#include <vector>
#include <algorithm>
#include <iostream>
```

```
class DummyCls {
public:
    int v_min = 6;
};

int main() {
    std::vector<int> c = {1, 2, 5, 8, 9, 10};
    int v_min = 4;
    // Usuń elementy mniejsze od `x` (oraz powstałe puste miejsca
    //   na końcu kontenera).
    // (`v_min` jest przekazywane przez wartość)
    c.erase(std::remove_if(std::begin(c), std::end(c),
        [v_min](int n) { return n < v_min; }), std::end(c));

    DummyCls dc;
    int v_max = 9;
    // Usuń elementy spoza zakresu [dc.v_min, v_max] (oraz powstałe puste
    //   miejsca na końcu kontenera).
    // (`v_max` jest przekazywane przez wartość, `dc` - przez referencję)
    c.erase(std::remove_if(std::begin(c), std::end(c),
        [v_max, &dc](int n) { return n < dc.v_min || n > v_max; }), c.end());

    std::cout << "c: ";
    std::for_each(std::begin(c), std::end(c),
        [](int i){ std::cout << i << ' '; });
    std::cout << '\n';

    return EXIT_SUCCESS;
}
```

Więcej informacji i przykładów znajdziesz na stronie C++11 FAQ: lambda oraz Lambda Expressions in C++ (MSDN).

Rozdział 8

Klasy: Rozszerzanie funkcjonalności

Podczas tworzenia oprogramowania spotykamy się zwykle z dwoma sprzecznymi dążeniami:

- Raz napisany, uruchomiony i przetestowany program powinien zostać w niezmienionej postaci (gdyż jego modyfikacje stwarzają ryzyko wprowadzenia błędu).
- Programy wymagają stałego dostosowywania do zmieniających się wymagań użytkownika, wymagań sprzętowych itp.

W tym rozdziale omówiono dwie z trzech kluczowych koncepcji programowania obiektowego, pozwalające w dużym stopniu pogodzić wspomniane dążenia: dziedziczenie i polimorfizm. Trzecia kluczowa koncepcja, abstrakcja danych, została już omówiona w rozdziale 2 Obiekty i klasy – podstawy.

8.1 Kompozycja i dziedziczenie

Często podczas tworzenia systemu napotykamy na sytuację, w której nowa klasa rozszerza możliwości innej, istniejącej już, klasy (np. posiadamy już klasę „samochód", a potrzebujemy klasy „samochód ze wspomaganiem kierownicy"). W takim przypadku zamiast implementować nową klasę w całości „od zera" warto wykorzystać istniejący już kod. W języku C++ istnieją dwie podstawowe techniki takiego rozszerzania funkcjonalności: kompozycja i dziedziczenie.

Zwróć szczególną uwagę na kwestię dokonywania wyboru między wspomnianymi technikami, opisaną w rozdziale 8.4 Kiedy kompozycja, a kiedy dziedziczenie?.

8.1.1 Kompozycja

Relacja **kompozycji** (ang. composition) to najprostszy rodzaj powiązania między dwiema klasami – polega po prostu na dodaniu do klasy `B` pola będącego typu `A` (zob. przykład 8.1). Oznacza to, że obiekt typu `B` nie może istnieć w oderwaniu od obiektu typu `A`, oraz że klasa `B` może korzystać wyłącznie z publicznego interfejsu udostępnianego przez klasę `A`.

Listing 8.1. Przykład relacji kompozycji.

```
class A {};

class B {
  A a_;  // relacja kompozycji
};
```

8.1.2 Dziedziczenie

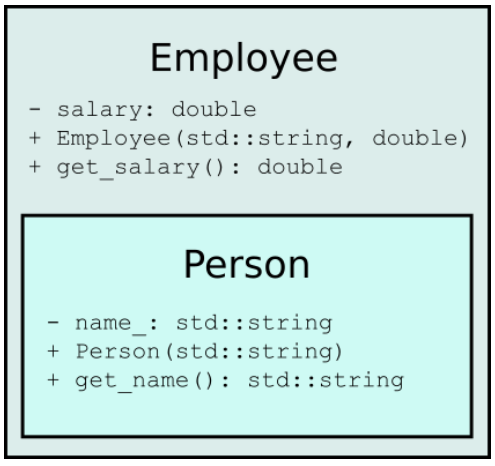
Dziedziczenie to jeden z najistotniejszych elementów obiektowości. Klasy powiązane relacją dziedziczenia tworzą hierarchię: **klasa macierzysta** (ang. base class) definiuje funkcjonalność wspólną dla obu klas, natomiast **klasa pochodna** (ang. derived class), zwana też **klasą potomną**, rozszerza tą funkcjonalność o właściwe sobie składowe. Mówimy, że „*klasa pochodna dziedziczy po klasie macierzystej*". Oczywiście nic nie stoi na przeszkodzie, aby klasa pochodna była równocześnie klasą macierzystą dla innej klasy, lecz generalnie należy unikać tworzenia nadmiernie głębokich hierarchii dziedziczenia.

W języku C++ istnieją różne typy dziedziczenia[1], lecz w praktyce stosuje się przede wszystkim dziedziczenie publiczne – klasa pochodna udostępnia wówczas publiczny interfejs klasy macierzystej, dzięki czemu możemy stosować klasę pochodną w miejsce klasy macierzystej. Przykład 8.2 pokazuje sposób definiowania publicznego dziedziczenia, a na rysunku 8.1 zilustrowano fakt rozszerzania funkcjonalności klasy macierzystej przez klasę pochodną.

Listing 8.2. Przykład relacji dziedziczenia (publicznego).

```
class Person {
private:
  std::string name_;
public:
  Person(std::string name) : name_(name) {}
  std::string get_name() { return name_; }
};

class Employee : public Person {
private:
  double salary_;
public:
  Employee(std::string name, double salary) : Person(name), salary_(salary) {}
  double get_salary() { return salary_; }
};

// ...

Employee employer("John Smith", 1000.0);
std::cout << employer.get_name() << std::endl;
```

Zwróć uwagę, że jako pierwszy element na liście inicjalizacyjnej konstruktora klasy pochodnej `Employee` pojawił się konstruktor klasy macierzystej `Person` – to ważne, aby przed przystąpieniem do konstruowania obiektu klasy pochodnej upewnić się, że składowe odziedziczone po klasie macierzystej zostały poprawnie zainicjalizowane (klasa pochodna tak naprawdę jest jednocześnie klasą macierzystą, dlatego nie można utworzyć jej obiektu bez wcześniejszego utworzenia obiektu klasy bazowej). Konstruktora klasy macierzystej nie można wywołać w ciele konstruktora klasy pochodnej, ponieważ klasy pochodne dziedziczą pola klasy podstawowej, a wszystkie pola klasy muszą zostać utworzone przed utworzeniem danej klasy.



Rysunek 8.1. W przypadku relacji dziedziczenia klasa pochodna (`Employee`) *jest* jednocześnie klasą macierzystą (`Person`) – zawiera komplet pól i metod odziedziczonych po klasie macierzystej. Nie ma zatem żadnych różnic pomiędzy używaniem metod klasy macierzystej i pochodnej w klasie pochodnej (czy też na obiekcie klasy pochodnej) – oczywiście mając na uwadze typy dziedziczenia i wynikające z nich ograniczenia.

W przypadku gdy w klasie pochodnej zdefiniowana zostanie ponownie taka sama metoda jak w klasie macierzystej, nastąpi tak zwane **przesłonienie** (ang. hiding) tej metody – wywołując tą metodę na obiekcie

[1]Język C++ dopuszcza dziedziczenie następujących typów: publiczne, chronione i prywatne.

klasy pochodnej będzie domyślnie wykonywana implementacja tej metody zawarta w klasie pochodnej (zob. przykład 8.3).

Listing 8.3. Przykład przesłaniania metod

```
#include <cstdlib>
#include <iostream>
#include <string>

struct Foo {
    std::string do_something() { return "Foo"; }
};

struct Bar : public Foo {
    std::string do_something() { return "Bar"; }
};

int main() {
    Bar b;

    // dostęp do "domyślnej" wersji metody (zgodnie z typem `b`)
    std::cout << b.do_something() << std::endl;

  // dostęp do wersji metody odziedziczonej po klasie `Foo`
    std::cout << b.Foo::do_something() << std::endl;

    return EXIT_SUCCESS;
}
```

Składowe chronione

Oprócz wprowadzonych w rozdziale 2.5.1 Kontrola dostępu składowych publicznych i prywatnych istnieją jeszcze składowe **chronione** (ang. protected), które mają ścisły związek z mechanizmem dziedziczenia i enkapsulacją – klasy pochodne mają bezpośredni dostęp do składowych chronionych, natomiast nie mają do nich dostępu *użytkownicy* klas pochodnych. Do definiowania składowych chronionych służy specyfikator dostępu `protected`.

Ważne

W dzisiejszych czasach stosowanie składowych chronionych jest (co do zasady) przejawem złego stylu programistycznego, gdyż twórcy klas pochodnych mogą nie w pełni rozumieć zasadę działania wewnętrznych mechanizmów klasy macierzystej (np. w kwestii zapewnienia jej niezmienników) – nadanie im bezpośredniego dostępu do składowych klasy macierzystej to łatwa droga do błędów[2].

Oczywiście w klasach „wewnętrznych" dla danego modułu programu, nieprzeznaczonych do użycia np. jako fragment biblioteki programistycznej, użycie składowych chronionych jest *dopuszczalne* (choć i tak warto je stosować tylko w ostateczności).

Dla zainteresowanych...

Oto przykład scenariusza, w którym użycie dostępu chronionego jest uzasadnione:
W hipotetycznym modelu sieci komunikacyjnej istnieją dwa rodzaje węzłów – nadawcy i odbiorcy – przy czym dany konkretny węzeł może w szczególności być zarówno nadawcą, jak i odbiorcą. Każdy węzeł (niezależnie od rodzaju) posiada kolejkę zadań do przetworzenia, natomiast nadawca posiada dodatkowe dwa elementy – bufor wysyłanej wiadomości oraz metodę „pobierz zadanie z kolejki, przetwórz i umieść wiadomość o statusie wykonania zadania w buforze". Taką funkcjonalność nadawcy można zrealizować jako klasę z prywatnym polem bufora oraz *chronioną* metodą umieszczania elementu w buforze, po której będzie dziedziczyć klasa węzła-nadawcy (zob. przykład 8.4). Dzięki temu będzie możliwość polimorficznego stosowania klasy węzła-nadawcy przy jednoczesnym ukryciu przed użytkownikami klasy jej szczegółów implementacyjnych (tj. sposobu implementacji bufora oraz jego możliwości dokonywania operacji na buforze).

[2] Zagadnienie to zostało szerzej omówione w dyskusji Should you ever use protected member variables? (Stack Overflow).

Listing 8.4. Przykład metody chronionej

```
class Node {
 /* ... */
};

class Sender {
public:
 void send_message() {  /* ... */ }

protected:
 void push_message(Message) { /* ... */ }  // metoda chroniona

private:
 Buffer buffer_;
};

// Klasa `SenderNode` dziedziczy publicznie po dwóch klasach.
class SenderNode : public Node, public Sender {
public:
 SenderNode(/* ... */) : Node(), Sender() { /* ... */ }
 void process_task() {
     Message m = tasks_.back();
     /* ... */
     push_message(m);
 }

private:
 TaskQueue tasks_;
};
```

Konstruktory odziedziczone

Konstruktory *nie są* dziedziczone. Jednak w przypadku, gdy klasa pochodna nie zawiera dodatkowych pól (albo gdy inicjalizacja takich pól domyślnymi wartościami będzie wystarczająca), możemy użyć konstrukcji `using BaseClass::BaseClass;` w celu *wygenerowania* konstruktorów klasy pochodnej na podstawie konstruktorów klasy macierzystej – swoistego ich „dziedziczenia” – przez co nazywa się je **konstruktorami odziedziczonymi** (ang. inherited constructors). Przykład 8.5 pokazuje zastosowanie wspomnianej konstrukcji.

Listing 8.5. „Dziedziczenie” konstruktorów.

```
class A {
private:
  int i_;
public:
  A(int i) : i_(i) {}
};

class B : public A {
public:
  using A::A;   // "dziedziczenie" konstruktorów
  void foo() {}
};
```

8.2 Polimorfizm

Co do zasady klasy pochodne dziedziczą składowe swej klasy macierzystej. Zdarza się jednak, że *działanie* (implementacja) pewnych odziedziczonych składowych powinno się różnić od ich działania określonego w

klasie macierzystej, przy czym zależy nam, aby kompilator sam dokonywał właściwego wyboru – w takiej sytuacji stosuje się polimorfizm.

W przypadku klas polimorfizm to mechanizm umożliwiający współistnienie wielu metod o tych samych nagłówkach (lecz różnych implementacjach), które mogą zostać wykonane w odpowiedzi na komunikat odebrany przez obiekt, przy czym wybór konkretnej metody dokonywany jest dynamicznie (w trakcie działania programu), na podstawie typu obiektu odbiorcy[3]. Polimorfizm pozwala zatem traktować odbiorcę komunikatu w sposób abstrakcyjny, bez znajomości jego typu, ponieważ właściwa wersja metody polimorficznej zostanie wybrana automatycznie, bez wiedzy wywołującego ją obiektu.

Rozważmy przypadek funkcji, która dla zadanego samochodu (obiektu klasy `Car`) obliczy minimalny czas podróży do celu odległego o s kilometrów na podstawie maksymalnej możliwej szybkości samochodu. Podobną operację możemy chcieć wykonać dla samochodu wyposażonego w tempomat (obiekt klasy `TempomatCar`). Musimy jednak pamiętać, że implementacja metody obliczającej maksymalną możliwą szybkość samochodu będzie różniła się pomiędzy klasami `Car` i `TempomatCar`. Przykład 8.6 przedstawia pierwszą (nieudaną) próbę realizacji zadania.

Listing 8.6. Przykład sytuacji, w której chcemy stosować polimorfizm.

```
#include <cstdlib>
#include <iostream>

class Car {
protected:
  double engine_horse_power_ = 100.0;
public:
  double compute_max_speed() const { return engine_horse_power_; }
};

class TempomatCar : public Car {
private:
  double tempomat_max_speed_ = 50.0;
public:
  double compute_max_speed() const {  // PRZESŁANIA metodę z klasy
                                      //   macierzystej
    return std::min(engine_horse_power_, tempomat_max_speed_);
  }
};

double compute_min_journey_time(const Car& car, double distance) {
  return distance / car.compute_max_speed();
}

int main() {
  Car car;
  TempomatCar tempomat_car;

  std::cout << compute_min_journey_time(car, 100.0) << std::endl;
  std::cout << compute_min_journey_time(tempomat_car, 100.0) << std::endl;

  return EXIT_SUCCESS;
}
```

Zwróć uwagę, że choć funkcja `compute_min_journey_time()` przyjmuje referencję na typ `Car`, możemy wywołać tę funkcję przekazując argument typu `TempomatCar` – wynika to z faktu, że obiekt klasy pochodnej zawiera w sobie wszystkie składowe odziedziczone po klasie macierzystej, zatem możemy z niego korzystać tak jak gdyby był on obiektem klasy macierzystej (zagadnienie to zostanie omówione dokładniej w kolejnych rozdziałach).

Po uruchomieniu powyższego programu na konsoli pojawi się następujący komunikat:

```
1
```

[3]Ogólne pojęcie polimorfizmu zostało omówione w rozdziale 1.1 Programowanie zorientowane na obiekty.

```
1
```

Przyczyną takiego zachowania jest to, że jeszcze na etapie kompilacji do wywołania wybrana zostanie metoda zdefiniowana w klasie macierzystej – zgodnie z typem parametru `car` umieszczonym w kodzie funkcji (czyli `const Car&`). Taki proces wyboru konkretnej wersji metody do wywołania nazywa się **wiązaniem statycznym** (ang. static linkage). Aby uzyskać pożądane zachowanie, musimy zadeklarować metodę `compute_max_speed()` jako *metodę wirtualną*.

Metoda wirtualna (ang. virtual method) to metoda w klasie bazowej zadeklarowana z użyciem słowa kluczowego `virtual`. Słowo kluczowe `virtual` sygnalizuje kompilatorowi, że nie chcemy, aby taka funkcja miała wiązanie statyczne. Zamiast tego oczekujemy, że wybór właściwej wersji metody odbędzie się w trakcie wykonywania programu, w zależności od rzeczywistego typu obiektu (określonego na podstawie zawartości pamięci, a nie na podstawie kodu źródłowego) dla którego wywoływana jest metoda wirtualna. Taki mechanizm nazywamy **wiązaniem dynamicznym** (ang. dynamic linkage, late binding).

Przykład 8.7 prezentuje zmodyfikowaną klasę `Car`, która obecnie wykorzystuje polimorfizm (pozostała część kodu pozostaje bez zmian). Zwróć uwagę na specyfikator `override` dodany na końcu nagłówka metody `compute_max_speed()` w klasie `TempomatCar`. Sygnalizuje on, że metoda ta **nadpisuje** (ang. override) metodę `compute_max_speed()` odziedziczoną po klasie `Car` (o takim samym prototypie) – czyli oznacza, że metoda `compute_max_speed()` jest polimorficzna. Umieszczenie specyfikatora `override` nie tylko poprawia czytelność kodu (gdyż pokazuje intencję programisty), lecz również umożliwia efektywniejszą diagnostykę potencjalnych błędów przez kompilator.

Metoda zadeklarowana jako wirtualna w klasie macierzystej jest automatycznie wirtualna w klasach pochodnych.

Listing 8.7. Przykład polimorfizmu.

```
class Car {
protected:
  double engine_horse_power_ = 100.0;
public:
  virtual double compute_max_speed() const { return engine_horse_power_; }

  // Potrzeba stosowania wirtualnych destruktorów w przypadku
  // dziedziczenia została uzasadniona w rozdziale `Wirtualne destruktory`.
  virtual ~Car() {}
};

class TempomatCar : public Car {
private:
  double tempomat_max_speed_ = 50.0;
public:
  double compute_max_speed() const override {  // NADPISUJE metodę z klasy
                                               //   macierzystej
    return std::min(engine_horse_power_, tempomat_max_speed_);
  }
};
```

Po uruchomieniu programu z tak zmienioną klasą `Car` na konsoli pojawi się następujący komunikat:

```
1
2
```

Tym razem kompilator wybrał właściwą wersję metody po zweryfikowaniu dynamicznego typu referencji. Identyczny mechanizm zadziała dla wskaźników na klasę macierzystą (stosowne przykłady pojawią się w kolejnych podrozdziałach).

8.2.1 Konwersja typu a dziedziczenie

Aby efektywnie korzystać z mechanizmów udostępnianych przez dziedziczenie należy mieć świadomość reguł rządzących konwersjami między typami obiektów powiązanych relacją dziedziczenia oraz między wskaźnikami i referencjami powiązanymi z obiektami klas macierzystych i pochodnych.

Konwersja „w górę”

Ponieważ obiekt klasy pochodnej zawiera w sobie wszystkie składowe odziedziczone po klasie macierzystej, możemy korzystać z takiego obiektu klasy pochodnej tak jak gdyby był on obiektem klasy macierzystej. W szczególności możemy powiązać referencję lub wskaźnik na klasę macierzystą z obiektem klasy pochodnej (zob. przykład 8.8). Zauważ, że w związku z tym otrzymując referencję lub wskaźnik na klasę macierzystą nie jesteśmy w stanie stwierdzić, jaki jest typ dynamiczny wskazywanego obiektu.

Listing 8.8. Konwersja typu pochodnego na typ bazowy (tzw. „w górę”).

```
class Person {
  // ...
};

class Employee : public Person {
  // ...
};

Person person;          // obiekt klasy macierzystej
Employee employer;      // obiekt klasy pochodnej

Person* p = &person;    // `p` wskazuje na obiekt typu `Person`

p = &employer;          // `p` wskazuje na fragment obiektu `employer`
                        // zawierający składowe odziedziczone po klasie Person

Person& r = employer;   // `r` jest powiązane z fragmentem obiektu
                        // `employer` zawierającym składowe odziedziczone
                        // po klasie `Person`
```

Taki typ konwersji nazywamy **konwersją „w górę”** (ang. derived-to-base conversion). Konwersja „w górę” dokonywana jest niejawnie przez kompilator, który w razie potrzeby generuje konstruktor kopiujący oraz operator przypisania[4]. Zauważ, że w przypadku typów danych nie powiązanych relacją dziedziczenia możemy powiązać referencję do danego typu lub wskaźnik na dany typ wyłącznie z obiektem tego samego typu statycznego[5].

Ważne

Fakt, że obiekt klasy pochodnej zawiera podobiekty dla każdej z klas macierzystych, jest kluczowy dla zrozumienia działania dziedziczenia.

Typ dynamiczny wyrażeń nie będących ani wskaźnikami, ani referencjami, jest zawsze taki sam, jak ich typ statyczny (np. zmienna typu `Person` będzie *zawsze* obiektem klasy `Person`).

Zasada „nie ma niejawnej konwersji typu macierzystego do typu pochodnego”

Konwersja „w górę” jest zawsze możliwa, gdyż obiekt klasy pochodnej zawiera w sobie wszystkie składowe odziedziczone po klasie macierzystej. Odwrotna zależność oczywiście nie jest prawdziwa, gdyż obiekt klasy macierzystej może istnieć zarówno jako fragment obiektu klasy pochodnej, jak i jako niezależny obiekt – w tym drugim przypadku nie posiada on składowych klasy pochodnej.

```
Person person;

Employee* e_ptr = &person;    // BŁĄD
Employee& e_ref = person;     // BŁĄD
```

Co ważne, taka niejawna konwersja nie jest możliwa nawet w sytuacji, gdy po uprzedniej konwersji typu potomnego na typ macierzysty chcemy z powrotem przekonwertować referencję lub wskaźnik na typ macierzysty na oryginalny typ potomny:

[4] Zagadnienie generowania metod specjalnych przez kompilator zostało omówione w rozdziale 9 Klasy: Tworzenie i niszczenie obiektów.

[5] Dopuszczalna jest również konwersja dodająca do typu kwalifikator `const`.

```
Employee employer;

Person& p_ref = employer;    // poprawne - konwersja "w górę"
Employee& e_ptr = p_ref;     // BŁĄD
```

Wynika to z faktu, że (w ogólnym przypadku) na etapie kompilacji nie jesteśmy w stanie stwierdzić, jaki jest dynamiczny typ obiektu – w związku z tym taki rodzaj domyślnej konwersji nie byłby bezpieczny. W rozdziale 11.2 Operatory rzutowania poznasz sposoby na radzenie sobie w takich sytuacjach.

Zasada „nie ma niejawnej konwersji między obiektami”

Automatyczna konwersja „w górę” działa jedynie dla wskaźników i referencji, nie dla obiektów. Owszem, przypisanie obiektu klasy pochodnej do obiektu klasy macierzystej (lub inicjalizowanie obiektu klasy macierzystej obiektem klasy pochodnej) jest możliwe, lecz nie zawsze ma pożądane przez nas działanie.

Zwróć uwagę, że zarówno wspomniana operacja inicjalizacji, jak i przypisania, wiąże się tak naprawdę z wywołaniem odpowiedniej metody specjalnej – konstruktora kopiującego albo operatora przypisania. Ponieważ wspomniane metody przyjmują (stałą) referencję na typ klasowy, konwersja „w górę” umożliwia nam przekazanie do nich obiektu klasy pochodnej. Operacje te nie są jednak wirtualne, dlatego zawsze wywołana zostanie wersja z klasy macierzystej (zob. przykład 8.9). Przekazanie obiektu klasy pochodnej do konstruktora klasy macierzystej spowoduje wykonanie wersji konstruktora właśnie z klasy macierzystej, przy czym konstruktor taki ma dostęp jedynie do składowych zdefiniowanych w klasie macierzystej. Podobnie przypisanie obiektu klasy pochodnej do obiektu klasy macierzystej spowoduje wywołanie operatora przypisania z klasy macierzystej, który nie ma pojęcia o polach zdefiniowanych w klasie pochodnej. W obu przypadkach nowo utworzony obiekt klasy macierzystej zostanie zainicjalizowany jedynie tymi polami klasy pochodnej, które zostały odziedziczone – mówimy, że obiekt klasy pochodnej zostanie **przycięty** (ang. sliced) do zawartego w nim podobiektu klasy macierzystej.

Listing 8.9. Próby konwersji między obiektami.

```
Employee employer;
Person person(employer);  // używa Person::Person(const Person&)
person = employer;        // używa Person::operator=(const Person&)
```

8.2.2 Wirtualne destruktory

Mechanizm dziedziczenia sprawia, że typ dynamiczny wskaźnika może różnić się od jego typu statycznego – wskaźnik na klasę macierzystą może wskazywać obiekt klasy pochodnej. W przypadku usuwania wskazywanego obiektu (np. w momencie zwalniania pamięci) program musi wiedzieć, której wersji destruktora powinien użyć – wersji z klasy macierzystej czy wersji z klasy pochodnej[6]. W związku z tym, aby umożliwić wybór odpowiedniego destruktora, destruktor klasy macierzystej powinien być *zawsze* zdefiniowany jako wirtualny (zob. przykład 8.10). Oczywiście mechanizm dziedziczenia sprawia, że destruktory w klasach pochodnych również będą wirtualne.

Listing 8.10. Definiowanie wirtualnego destruktora.

```
class A {
public:
  virtual ~A() {}
};
```

Warto pamiętać, że definiowanie destruktora jako metody wirtualnej to sygnał dla użytkowników takiej klasy, że klasa ta może być użyta jako klasa macierzysta. Jeśli nie jest to naszą intencją, lepiej powstrzymać się od takiego zabiegu.

Choć destruktory nie są dziedziczone, w praktyce destruktor klasy pochodnej *nadpisuje* destruktor klasy macierzystej (zdefiniowany jako wirtualny) – w związku z tym destruktor w klasie pochodnej powinien być oznaczony słowem kluczowym `override`.

[6]Próba usunięcia obiektu klasy pochodnej poprzez wskaźnik na klasę macierzystą, w przypadku gdy klasa macierzysta nie posiada wirtualnego destruktora, prowadzi do niezdefiniowanego zachowania.

8.2.3 Szablon `std::function`

Rodzaj polimorfizmu możemy również uzyskać korzystając z wprowadzonego w standardzie C++11 szablonu klasy `std::function`. Instancje tego szablonu służą do przechowywania i wywoływania dowolnych elementów „wywoływalnych" – funkcji, wyrażeń lambda, metod klas itp. (zob. przykład 8.11). Polimorfizm *statyczny* dla klas możemy zatem osiągnąć dodając pole będące typu odpowiedniej instancji `std::function`[7].

Listing 8.11. Wykorzystanie szablonu klasy `std::function` do uzyskania statycznego polimorfizmu.

```
#include <cstdlib>
#include <iostream>
#include <functional>

struct Cls {
    Cls(std::function<void()> f) : f_(f) {}
    std::function<void()> f_;
};

int main() {
    Cls c1([](){ std::cout << "X" << std::endl; });
    Cls c2([](){ std::cout << "Y" << std::endl; });

    c1.f_();
    c2.f_();

   return EXIT_SUCCESS;
}
```

8.3 Klasy abstrakcyjne

Zacznijmy omawianie zagadnienia klas abstrakcyjnych od następującego przykładu:
Powiedzmy, że posiadasz klasę macierzystą `Animal` zawierającą m.in. metodę `make_sound()` służącą do wydawania odgłosu danego zwierzęcia (oraz różne klasy potomne – np. `Dog`, `Cat` itd.). Jaką implementację powinna posiadać metoda `make_sound()` w klasie `Animal`? No cóż, nie jesteś w stanie rozsądnie zdefiniować takiej metody wewnątrz klasy `Animal`, gdyż rodzaj wydawanego odgłosu zależy od konkretnego zwierzęcia – pies szczeka, kot miauczy itd.

Język C++ pozwala na zaznaczenie takiej sytuacji poprzez zadeklarowanie metody jako **w pełni wirtualnej** (ang. pure virtual). W przeciwieństwie do „zwykłych" metod wirtualnych, metody w pełni wirtualne nie muszą zostać zdefiniowane w danej klasie. Aby zaznaczyć, że dana metoda jest w pełni wirtualna, dodajemy na końcu jej deklaracji `= 0` (zob. przykład 8.12).

Klasę zawierająca metodę w pełni wirtualną nazywamy **klasą abstrakcyjną** (ang. abstract class). Klasy dziedziczące po **abstrakcyjnej klasie macierzystej** (ang. abstract base class), które nie nadpiszą *wszystkich* odziedziczonych metod w pełni wirtualnych, również stają się klasami abstrakcyjnymi. Kompilator uniemożliwia (bezpośrednie) tworzenie obiektów takiego typu. Klasę, która nie zawiera (niezdefiniowanych) metod w pełni wirtualnych nazywamy **klasą konkretną** (ang. concrete class).

W przykładzie 8.12 zaprezentowano dwie klasy abstrakcyjne, `Animal` oraz `LandAnimal`, oraz klasę konkretną `Dog`.

8.3.1 Interfejsy

Termin **interfejs** (ang. interface) posiada kilka znaczeń[8]. W odniesieniu do relacji dziedziczenia interfejs oznacza taką abstrakcyjną klasę, która nie posiada stanu (nie przechowuje danych), a jedynie posiada same

[7] Przed wprowadzeniem tego szablonu rozwiązanie zbliżone, choć znacznie mniej uniwersalne, można było uzyskać za pomocą znanych z języka C wskaźników do funkcji.

[8] zob. Interface (Wikipedia)

Listing 8.12. Przykład zastosowania metody w pełni wirtualnej.

```
class Animal {  // klasa abstrakcyjna
public:
  virtual std::string make_sound() const = 0;
  virtual ~Animal() {}
};

class LandAnimal : public Animal {};  // klasa abstrakcyjna

class Dog : public Animal {  // klasa konkretna
public:
  std::string make_sound() const override {
    return "Woof!";
  }
};

// ...

Animal animal;           // błąd kompilacji - klasa "Animal" to
                      //   klasa abstrakcyjna
LandAnimal land_animal;  // błąd kompilacji - klasa "LandAnimal" to też
                      //   klasa abstrakcyjna
Dog dog;                 // OK - klasa "Dog" to klasa konkretna
```

metody w pełni wirtualne (zob. przykład 8.13). Kiedy w konkretnej klasie zdefiniowane są wszystkie metody interfejsu mówimy, że klasa **implementuje** (ang. implements) dany interfejs.

Listing 8.13. Przykład interfejsu.

```
class IMovable {
public:
  virtual void move() = 0;
  virtual ~IMovable() {}
};
```

Ponieważ interfejs jedynie *deklaruje* udostępniane operacje, bez ich *definiowania* (tj. interfejs nie zawiera *implementacji* żadnych operacji), klasy mogą implementować wiele interfejsów – bez problemów wynikających z wielokrotnego dziedziczenia. Stosowanie interfejsów umożliwia zatem efektywniejsze wykorzystanie klas implementujących interfejsy, dzięki lepszej kontroli nad udostępnianiem funkcjonalności takich klas (kontrola jest możliwa dzięki konwersji „w górę” do interfejsów).

8.4 Kiedy kompozycja, a kiedy dziedziczenie?

W literaturze oraz w „życiu inżyniera oprogramowania” często można spotkać się z dwoma skrajnymi podejściami: stosowanie dziedziczenia, gdy to tylko możliwe, oraz na odwrót – stosowanie wyłącznie kompozycji. W praktyce ten problem przypomina dylemat „sok czy kanapka”: nie można powiedzieć „zawsze wybiorę sok” albo „zawsze wybiorę kanapkę”, gdyż odczuwając pragnienie chętniej wypijesz sok niż zjesz kanapkę, a odczuwając głód – na odwrót. Podobnie jest z kompozycją i dziedziczeniem – obie techniki mają swoje dobre zastosowania. Jeśli nie wiesz, którą technikę wybrać pracując nad modelowaniem danego zagadnienia, powinieneś poświęcić więcej czasu na dokładniejsze poznanie obu mechanizmów oraz na lepsze zrozumienie istoty modelowanego zagadnienia. Poniżej znajdują się wskazówki, które pomogą Ci podjąć właściwą decyzję.

W podręcznikach programowania często można spotkać następujący opis:

- Kompozycja to relacja typu „posiada” (ang. has-a relationship), np.:
  - samochód „posiada” silnik
  - człowiek „posiada” imię

- Dziedziczenie to relacja typu „jest” (ang. is-a relationship), np.:
  - samochód „jest” pojazdem
  - człowiek „jest” ssakiem

Są to przydatne ogólne formułki („zasady kciuka”), lecz w praktyce warto uwzględnić jeszcze **zasadę podstawienia Liskov**[9] (ang. Liskov substitution principle).

> **Definicja: Zasada podstawienia Liskov**
>
> Funkcje które używają wskaźników lub referencji do klas bazowych, muszą być w stanie używać również obiektów klas dziedziczących po klasach bazowych, bez dokładnej znajomości tych obiektów.

Zasada ta pozwala wychwycić sytuacje, gdy zastosowanie relacji dziedziczenia między dwiema *pozornie* powiązanymi klasami jest błędne. Rozważ następujący przypadek:

Otrzymaliśmy zadanie zaprojektowania frameworku do gier komputerowych rozgrywanych na płaskiej planszy złożonej z kwadratów – stworzyliśmy zatem klasę `Board2d`. Ponieważ później wymagania zmieniły się – nasz framework powinien obsługiwać również gry na planszy przestrzennej, złożonej z sześcianów – wywiedliśmy klasę `Board3d` z klasy `Board2d`, jako rozszerzenie dwuwymiarowej planszy o trzeci wymiar (zob. przykład 8.14).

Listing 8.14. Przykład niepoprawnego zastosowania dziedziczenia.

```
class Board2d {
  public:
    Tile get_tile(int x, int y);
  private:
    std::vector< std::vector<Tile> > board2d_;
};

class Board3d : public Board2d {
  public:
    Tile get_tile(int x, int y, int z);
  // ...
};
```

Na pozór dziedziczenie wygląda jak słuszna decyzja, lecz problem pojawia się w *niezgodności interfejsów*: metody w stylu `get_tile()` potrzebują podania dwóch parametrów (x, y) dla planszy 2D, lecz w przypadku planszy 3D wymagają trzech parametrów (x, y, z). Klasa `Board3d` musi zatem implementować metodę `get_tile(int x, int y, int z)`, lecz jednocześnie – ponieważ dziedziczy metodę `get_tile(int x, int y)` z klasy macierzystej – w kodzie programu może dojść do następującej sytuacji:

```
Board3d board3d;
board3d.get_tile(1, 2);   // to nie ma sensu!
```

W tym przypadku poprawną relacją będzie kompozycja – plansza 3D nie chce udostępniać metody `get_tile(int x, int y)`, ale może chcieć wykorzystać tę funkcjonalność do realizacji własnych operacji (zob. przykład 8.15).

Listing 8.15. Przykład zastosowania kompozycji.

```
class Board2d {
  public:
    Tile get_tile(int x, int y) {
        return board2d_[y][x];
    }
  private:
    std::vector< std::vector<Tile> > board2d_;
};

class Board3d {
```

[9] Dokładniejsze omówienie tej zasady znajdziesz tutaj.

```
  public:
    Tile get_tile(int x, int y, int z) {
      return board3d[z].get_tile(x, y);
    }
  private:
    std::vector<Board2d> board3d;
};
```

W praktyce dobrze sprawdza się następująca zasada:

Dobre praktyki

Domyślnie *wybieraj kompozycję zamiast dziedziczenia (gdyż jest to rozwiązanie bardziej elastyczne), lecz nie stosuj reguły „*zawsze kompozycja*". Po prostu jeśli zamierzasz zastosować dziedziczenie, zastanów się dwukrotnie – być może bardziej odpowiednia będzie kompozycja.*

W przypadku kompozycji łatwiej zmienić zachowanie obiektu danej klasy w trakcie działania programu, na przykład stosując technikę **wstrzykiwania zależności**[10] (ang. dependency injection) albo z użyciem obiektu funkcyjnego `std::function`. Z kolei dziedziczenie jest w tej kwestii znacznie mniej elastyczne, gdyż języki takie jak C++ czy Java nie pozwalają na zmianę typu obiektu po jego utworzeniu, a dodatkowo większość języków programowania nie pozwala na wielokrotne dziedziczenie (tzn. dziedziczenie po więcej niż jednym typie)[11].

Oto dobre wyznaczniki tego, jaki rodzaj powiązania chcemy utworzyć między dwoma abstrakcyjnymi typami danych, Adt_1 i Adt_2, posiadającymi zbliżoną funkcjonalność (przykłady ich zastosowania w praktyce pokazano później). Kluczowe pytanie brzmi: czy abstrakcja Adt_2 udostępnia *pełne* API (każdą jedną publiczną składową) abstrakcji Adt_1 *oraz* czy abstrakcja Adt_2 powinna umożliwiać jej wykorzystanie w miejscu, gdzie oczekiwana jest abstrakcja Adt_1?

1. Jeśli powyższe kryterium nie jest spełnione, lepiej zastosować kompozycję.
2. Jeśli jest spełnione – *można rozważyć* użycie dziedziczenia.
   Musimy sobie jeszcze zadać pytanie, czy faktycznie potrzebujemy *dynamicznego* polimorfizmu za pomocą dziedziczenia, czy wystarczy *statyczny* rodzaj polimorfizmu osiągany np. z użyciem szablonu `std::functional`. W praktyce sprowadza się to do pytania: czy abstrakcja Adt_2 posiada dodatkowy *stan* (w stosunku do abstrakcji Adt_1)?
   (a) Jeśli tak – faktycznie warto użyć dziedziczenia;
   (b) jeśli nie – wystarczy kompozycja.

Oto przykłady wybranych scenariuszy do każdego ze wspomnianych przypadków:

- Wspomnianą planszę 3D można potraktować jako kilka złożonych ze sobą plansz 2D – plansza 3D nie powinna udostępniać metody `Board2d::get_tile(int x, int y)`, ale może chcieć wykorzystać tę funkcjonalność do realizacji własnych operacji. To przypadek 1 – należy zastosować kompozycję.
- Przytoczona w rozdziale 8.2 abstrakcja „samochodu z tempomatem" wyrażona przez klasę `TempomatCar` realizuje w pełni funkcjonalność „zwykłego" samochodu wyrażoną poprzez klasę `Car` – czyli udostępnia pełne jego API – ale posiada inny stan (dodatkowa konfiguracja tempomatu). Jeśli w programie będziemy chcieli przechowywać informację o flocie pojazdów i będziemy chcieli mieć możliwość wygodnego uzyskiwania informacji o minimalnym czasie przejazdu dla *każdego z pojazdów* (trzymanych w pewnej kolekcji), będzie to wówczas przypadek 2a – warto w takiej sytuacji zastosować dziedziczenie.
- Hipotetyczny formularz danych osobowych zawiera m.in. rubrykę „numerem telefonu", którego poprawność powinna zostać zweryfikowana przed wysłaniem formularza. Znakomita większość logiki formularza jest uniwersalna, jednak dopuszczalne formaty zapisu numerów telefonów różnią się pomiędzy krajami. To przypadek 2b – można dodać do klasy reprezentującej formularz pole przechowujące uchwyt do funkcji weryfikującej poprawność numeru telefonicznego (np. z użyciem szablonu `std::functional`) oraz stosowny parametr w konstruktorze tej klasy; właściwy sposób walidacji

[10] Dokładniejsze omówienie techniki wstrzykiwania zależności znajdziesz tutaj.

[11] Język C++ dopuszcza dziedziczenie wielokrotne, jednak jego stosowanie rodzi szereg problemów i zdecydowanie komplikuje kod. Należy je stosować tylko wówczas, gdy to faktycznie niezbędne (a takie sytuacje zachodzą niezmiernie rzadko).

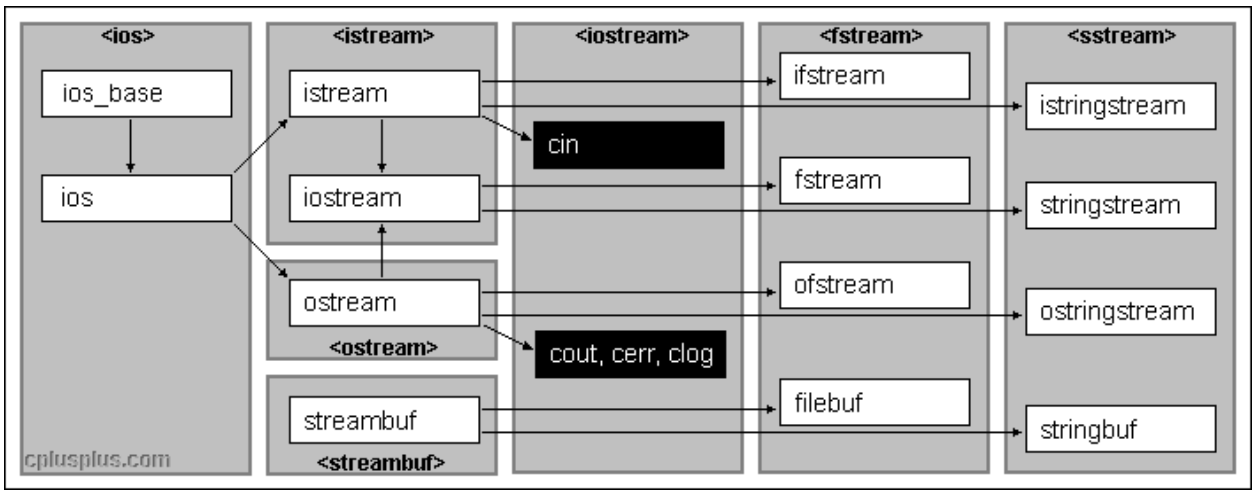
będzie określany w momencie tworzenia instancji formularza (poprzez przekazanie do konstruktora odpowiedniego argumentu).

Dlaczego nie należy nadużywać dziedziczenia, a wręcz stosować je bardzo oszczędnie? Choć posiada ono wiele niezaprzeczalnych zalet, na przykład umożliwia zastosowanie *dynamicznego* polimorfizmu, ma również kilka wad – w szczególności:

- Nie możesz zmienić implementacji metod odziedziczonych po klasach macierzystych w trakcie działania programu (gdyż relacja dziedziczenia jest określana na etapie kompilacji).
- **Ścisłe powiązanie** (ang. tight coupling) między klasami w przypadku dziedziczenia sprawia, że implementacja klas pochodnych zależy w dużym stopniu od implementacji klasy macierzystej – każda zmiana w implementacji klasy macierzystej wymusi zmiany w implementacji klasy pochodnej.
- Nadmierne stosowanie dziedziczenia prowadzi do powstawania wielopoziomowych, nieczytelnych hierarchii klas.

8.4.1 Przykład: Biblioteka standardowa we/wy

Dobrym przykładem wykorzystania dziedziczenia jest standardowa biblioteka wejścia wyjścia. Schemat relacji pomiędzy klasami tej biblioteki przedstawia Rys. 8.2.



Rysunek 8.2. Hierarchia standardowych klas strumieni we/wy (zdefiniowanych w pliku nagłówkowym `<iostream>`). (Źródło: cplusplus.com)

Rozdział 9

Klasy: Tworzenie i niszczenie obiektów

Do poprawnego korzystania z klas niezbędne jest zrozumienie reguł określających zasady tworzenia i usuwania obiektów klas (konstruktorów i destruktorów), inicjalizacji pól klasy, a także operacji przypisywania i kopiowania takich obiektów.

9.1 Inicjalizacja pól klasy

Gdy definiujemy zmienne do dobrej praktyki należy niezwłoczne nadanie im pożądanej wartości – najlepiej obie te operacje zawrzeć w ramach jednej instrukcji, czyli dokonanie inicjalizacji:

```
int i = 1;  // definicja z jawną inicjalizacją
int j;      // definicja z inicjalizacją domyślną -- w przypadku typów
            //   wbudowanych wartość niezdefiniowana
j = 2;      // PRZYPISANIE wartości (to nie jest inicjalizacja!)
```

Podobnie wygląda sytuacja w przypadku pól klas – najlepiej doprowadzić do takiej sytuacji, gdy pola od razu w momencie definicji zostają *jawnie* zainicjalizowane odpowiednią wartością. Owszem, nawet gdy nie dokonamy analogicznej jawnej ich inicjalizacji kompilator mimo to dokona inicjalizacji *niejawnej*[1], jednak nadana wartość może nie odpowiadać niezmiennikom klasy (zob. przykład 9.1).

Listing 9.1. Niezalecane podejście: brak inicjalizacji pól wewnątrz klasy

```
// Dozwolone, ale NIEZALECANE -- brak inicjalizacji wewnątrz klasy.
class Foo {
public:
  Foo(int i) {
    // W tym miejscu `i_` i `v_` mają wartości domyślne (zgodnie z zasadami
    //   inicjalizacji niejawnej), czyli:
    //   * w przypadku np. typów wbudowanych wartość niezdefiniowaną
    //   * w przypadku typów klasowych -- wartość nadaną przez konstruktor
    //     domyślny
    /* ... */
    i_ = i;  // Dopiero tu nadajemy "rozsądną" wartość polu `i_`.
  }
private:
  int i_;
  std::vector<int> v_;
};
```

Dobre praktyki

Aby uniknąć takich sytuacji warto zawsze stosować inicjalizatory wewnątrzklasowe *(a przynajmniej w przypadku pól typów wbudowanych i ich pochodnych), a gdy to rozwiązanie nie wystarcza – dodatkowo skorzystać z* list inicjalizacyjnych konstruktora. *To jedyne dwa sposoby na inicjalizację pól w C++17.*

[1] Inicjalizacja niejawna odbywa się zgodnie z zasadami opisanymi w rozdziale 17.3.2 Inicjalizacja domyślna.

9.1.1 Inicjalizator wewnątrzklasowy

Mechanizm **inicjalizatora wewnątrzklasowego** (ang. in-class initializer), wprowadzony w standardzie C++11, umożliwia *jawne* określenie domyślnej wartości pola w miejscu jego definicji (przy czym faktyczne nadanie polu wartości odbywa się dopiero w momencie wywołania konstruktora klasy). Inicjalizator określa się albo z użyciem operatora =, albo za pomocą nawiasów klamrowych. Nie ma znaczenia, czy typ pola posiada kwalifikator `const`, czy też nie (zob. przykład 9.2).

Listing 9.2. Inicjalizator wewnątrzklasowy

```
struct X {
  X(int i) {}
};

// Wszystkie pola klasy `Foo` inicjalizowane są jawnie.
struct Foo {
  Foo(int i) {
    // W tym miejscu `i_` ma już wartość 10...
    i_ = i;  // ...a tu jedynie nadpisujemy tę wartość.
  }

  // inicjalizacja pola typu wbudowanego
  int i_ = 10;

  // inicjalizacja pola z kwalifikatorem `const` też jest dopuszczalna
  const double x_ = 7.41;

  // inicjalizacja kontenera biblioteki standardowej...
  std::vector<int> vi_{1, 2, 3};   // dla typów wbudowanych
  std::vector<X> vs_{X(1), X(2)};  // dla typów klasowych
};
```

9.1.2 Listy inicjalizacyjne

Drugim sposobem na jawną inicjalizację składowej jest użycie tzw. *listy inicjalizacyjnej konstruktora*, która wykonywana jest jeszcze *przed* rozpoczęciem wykonywania *ciała* konstruktora. Lista inicjalizacyjna składa się z dwukropka i występującej po niej liście inicjalizowanych pól (oddzielonych przecinkami). Po nazwie danego pola podajemy w nawiasach okrągłych[2] wartość, którą chcemy zainicjalizować to pole. W przypadku pól będących typu klasowego w nawiasach podajemy parametry, które zostaną przekazane do *konstruktora* odpowiedniego pola,

Oto zmodyfikowany kod przykładu 9.1, w którym konstruktor klasy `Foo` korzysta z listy inicjalizacyjnej:

```
class Foo {
public:
  Foo(int i) : i_(i), v_(3, 0) {}  // konstruktor z listą inicjalizacyjną:
                                   //   inicjalizuje pole `i_` wartością `i`,
                                   //   oraz wywołuje dla pola `v_`
                                   //   konstruktor std::vector<int>(3, 0)
                                   //   (czyli tzw. konstruktor wypełniający)
private:
  int i_ = 0;
  std::vector<int> v_;
};
```

Dobre praktyki

Konstruktory powinny nadpisywać wartość pola zawierającego inicjalizator w miejscu definicji wyłącznie wówczas, gdy wartość ta różni się od określonej przez inicjalizator wewnątrzklasowy tego pola – w

[2] Albo nawiasach klamrowych (zob. list initialization).

przeciwnym razie mielibyśmy do czynienia z duplikacją kodu.

Dlaczego do dobrych praktyk należy każdorazowe stosowanie inicjalizatorów wewnątrzklasowych dla pól typów wbudowanych (i ich pochodnych), skoro inicjalizacji można dokonać z użyciem listy inicjalizacyjnej konstruktora? Ponieważ w przypadku korzystania z samych list inicjalizacyjnych trzeba pamiętać, aby w *każdym* konstruktorze zainicjalizować *wszystkie* takie pola (zob. przykład 9.3).

Listing 9.3. Niezalecane podejście: konstruktor bez listy inicjalizacyjnej.

```
// BŁĄD: Niezainicjalizowane pola klasy...
class Foo {
public:
  Foo() : i_(0), j_(0) {}  // poprawne -- oba pola jawnie zainicjalizowane
  Foo(int x) : i_(0) {
    // BŁĄD: W tym miejscu `i_` ma wartość 0, ale `j_` ma wartość
    // domyślną - czyli w tym przypadku niezdefiniowaną.
  }
private:
  int i_;
  int j_;
};
```

Listy inicjalizacyjne są czasem niezbędne...

Czasem nie możemy ignorować różnicy między tym, czy zmienna jest od razu inicjalizowana, czy przypisujemy do niej wartość z użyciem osobnej instrukcji. Przykładem takich sytuacji są pola o typie posiadającym kwalifikator `const`, pola będące referencjami, oraz pola będące typu klasowego `T` w sytuacji, gdy klasa `T` nie definiuje konstruktora domyślnego. Poniższy kod nie zadziała:

```
struct ConstRefClass {
  ConstRefClass(int ii)
  // W tym momencie wszystkie pola inicjalizowane są
  // domyślnymi wartościami...
  {
    i_ = ii;     // poprawne
    ci_ = ii;    // BŁĄD: nie można zmienić wartości stałej
    ri_ = ii;    // BŁĄD: referencja `ri_` nie została zainicjalizowana
  }

  int i_;
  const int ci_;
  int& ri_;
};
```

W przypadku takich pól, gdy chcemy nadać im wartość przekazaną jako argument konstruktora, jedynym rozwiązaniem jest zastosowanie listy inicjalizacyjnej:

```
ConstRefClass(int& ii): i_(ii), ci_(ii), ri_(ii) {}
```

Kolejność inicjalizacji pól

Kompilator ignoruje kolejność elementów na liście inicjalizacyjnej – pola będą inicjalizowane w takiej kolejności, w jakiej występują *w definicji klasy*. Związane z tym potencjalne problemy ilustruje przykład 9.4.

Listing 9.4. Niechlujna próba inicjalizacji jednego pola wartością innego pola

```
class Foo {
public:
  // Zgodnie z kolejnością występowania pól w DEFINICJI KLASY
  //   pole `i_` jest inicjalizowane przed `j_` -- zatem
  //   zostanie ono zainicjalizowane niezdefiniowaną wartością!
  Foo(int val): j_(val), i_(j_) {}
```

```
private:
  int i_;
  int j_;
};
```

W przypadku klasy z przykładu 9.4, aby uniknąć problemów z inicjalizacją pól wystarczyłoby zmienić listę inicjalizacyjną konstruktor w następujący sposób:

```
Foo(int val): i_(val), j_(val) {}
```

Dobre praktyki

Z korzystaniem z list inicjalizacyjnych wiążą się dwie dobre praktyki:

- *Umieszczaj pola na liście inicjalizacyjnej w takiej samej kolejności, w jakiej występują one w definicji klasy.*
- *Unikaj (o ile to tylko możliwe) inicjalizowania jednego pola wartością innego pola.*

9.2 Konstruktory

Ogólną ideę konstruktora – specjalnej metody określającej sposób inicjalizacji obiektu klasy – przedstawiono w rozdziale 2.7.1 Konstruktor. W tym rozdziale omówione zostaną szczególne rodzaje konstruktorów oraz elementy języka wspomagające ich efektywne implementowanie.

9.2.1 Konstruktor domyślny

Konstruktor domyślny (ang. default constructor) to konstruktor, który nie przyjmuje żadnych parametrów. Kompilator wywołuje go, gdy obiekt jest definiowany, ale nie jest jednocześnie jawnie inicjalizowany:

```
Rectangle rect_b;  // wywołanie domyślnego konstruktora klasy Rectangle
                   //   (brak nawiasów jest celowy!)
```

Zwróć uwagę, że w powyższym przykładzie nie występują nawiasy, które zwykle umieszczamy w przypadku wywołań funkcji (i konstruktorów) – podanie nawiasów po nazwie tworzonego obiektu tak naprawdę oznaczałoby nie tyle utworzenie obiektu typu `Rectangle`, co deklarację funkcji `rect_c` nie przyjmującej parametrów i zwracającej obiekt typu `Rectangle`:

```
Rectangle rect_c();  // prototyp funkcji rect_c()
```

Jeśli sami nie zdefiniujemy w danej klasie *żadnego* jej konstruktora, kompilator automatycznie wygeneruje tzw. **syntetyzowany konstruktor domyślny** (ang. synthesized default constructor) – język C++ wymaga bowiem, aby *każda* klasa posiadała *choć jeden* konstruktor[3]. Natomiast gdy klasa deklaruje jawnie choć jeden konstruktor, konstruktor domyślny nie zostanie wygenerowany automatycznie (możemy go jednak zdefiniować samemu)[4] – język C++ nie wymaga bowiem, aby klasa posiadała konstruktor domyślny.

Informacja

Kompilator automatycznie wygeneruje konstruktor domyślny tylko w przypadku, gdy klasa nie deklaruje żadnego konstruktora.

Język C++ nie wymaga, aby klasa posiadała konstruktor domyślny – klasa musi posiadać jedynie choć jeden (jakikolwiek) konstruktor.

Dobre praktyki

W praktyce, jeśli klasa definiuje własne konstruktory, powinna *również jawnie definiować konstruktor domyślny.*

[3] To logiczne – w przeciwnym razie nie wiadomo byłoby, jak tworzyć obiekty takiego typu i ew. typów pochodnych.

[4] Uzasadnienie takiej zasady jest następujące: skoro w *niektórych* przypadkach klasa potrzebuje kontroli nad procesem inicjalizacji, to zapewne w istocie potrzebuje go we *wszystkich* przypadkach.

Syntetyzowany konstruktor domyślny inicjalizuje pola w następujący sposób:

- jeśli pole posiada inicjalizator w miejscu definicji[5], użyj go do inicjalizacji pola;
- w przeciwnym razie zainicjalizuj pole wartością domyślną[6].

Jak wspomniano w rozdziale 9.1, dla pewnych typów pól taka strategia inicjalizacji *może* skutkować nadaniem polu o typie wbudowanym (lub pochodnym dla niego) wartości niezdefiniowanej – dlatego najlepiej stosować inicjalizator wewnątrzklasowy dla takiego pola albo chociaż zaimplementować własną wersję konstruktora domyślnego, która nada temu polu wartość za pomocą listy inicjalizacyjnej.

> **Ostrzeżenie**
>
> *Klasy posiadające pola typu wbudowanego lub złożonego mogą w pełni polegać na wynikach działania syntetyzowanego konstruktora domyślnego tylko wtedy, gdy* wszystkie *takie pola są zainicjalizowane w miejscu deklaracji.*

Podobnie w sytuacji, gdy klasa posiada pole typu klasowego `T`, a klasa `T` nie definiuje konstruktora domyślnego, kompilator nie będzie w stanie zainicjalizować takiego pola – w takim przypadku domyślny konstruktor nie zostanie wygenerowany (zob. przykład 9.5).

Listing 9.5. Czasem nie da się wygenerować konstruktora domyślnego. . .

```
#include <cstdlib>

struct X {
    // Klasa zawiera jawny konstruktor, więc konstruktor domyślny nie zostanie
    //   wygenerowany.
    X(int i) {}
};

struct Foo {
    X x;  // Klasa `Foo` posiada pole typu nie defniującego konstruktora
          //   domyślnego, zatem konstruktor domyślny klasy `Foo` nie zostanie
          //   wygenerowany.
};

int main() {
    Foo f;  // BŁĄD: Próba wywołania (nieistniejącego) konstruktora domyślnego
            //   klasy `Foo`.

    return EXIT_SUCCESS;
}
```

9.2.2 Konstruktory a argumenty domyślne

Konstruktor zawierający domyślne wartości dla *wszystkich* przyjmowanych argumentów[7], pełni jednocześnie rolę konstruktora domyślnego, gdyż można go wywołać nie podając ani jednego argumentu:

```
class Foo {
public:
  // Poniższy konstruktor pełni rolę konstruktora domyślnego
  //   gdyż wywołanie Foo() jest poprawne.
  Foo(int val = 0): i_(val), j_(i_) {}
private:
  int i_;
  int j_;
};
```

Należy jednak pamiętać, że mechanizm argumentów domyślnych w odniesieniu do konstruktorów rozsądnie jest stosować wyłącznie w przypadku, gdy konstruktor przyjmuje tylko jeden argument – wówczas unika się

[5] czyli gdy zastosowano inicjalizację pola wewnątrz klasy, zob. rozdz. 9.1 Inicjalizacja pól klasy

[6] zob. rozdz. 17.3 Inicjalizacja danych

[7] Argumenty domyślne zostały omówione w rozdziale 17.2 Argumenty domyślne.

niejednoznaczności i potencjalnych błędów, gdy kompilator próbuje dopasować odpowiedni konstruktor do podanych w wywołaniu (wielu) argumentów.

9.2.3 Konstruktory delegujące

Konstruktory delegujące (ang. delegating constructors) pozwalają uniknąć duplikacji kodu pomiędzy konstruktorami lub definiowania pomocniczych metod (często o nazwach w stylu `init()`) wywoływanych później w poszczególnych konstruktorach.

W poniższym przykładzie hipotetyczna klasa `X` powinna posiadać dwa konstruktory, a każdorazowo (tj. niezależnie od użytego konstruktora) w przypadku pomyślnego zakończenia procesu tworzenia obiektów tej klasy należy wypisywać na standardowe wyjście informację o utworzonym obiekcie – to idealna sytuacja do zastosowania konstruktorów delegujących.

W przypadku konstruktorów delegujących lista inicjalizacyjna zawiera wyłącznie pojedynczy element – odpowiadający konstruktorowi, do którego zostanie wydelegowane zadanie inicjalizacji. Stosowny przykład przedstawia listing 9.6.

Listing 9.6. Delegowanie konstruktorów

```
#include <cstdlib>
#include <iostream>

class X {
public:
  // konstruktor DELEGOWANY
  X(int a, int b) : a_(a), b_(b) {
    std::cout << "Utworzono X(a=" << a_ << ", b=" << b_ << ")" << std::endl;
  }
  // konstruktor DELEGUJĄCY
  //   X(int a) deleguje część odpowiedzialności do X(int a, int b)
  //   Użycie mechanizmu delegacji uniemożliwia umieszczenie
  //   innych elementów na liście inicjalizacyjnej.
  X(int a) : X(a, 0) {}

private:
  int a_;
  int b_;
};

int main() {
  X(1, 2);
  X(1);
  return EXIT_SUCCESS;
}
```

Kolejność wykonywania kodu w przypadku konstruktorów delegujących wygląda następująco:

(1) lista inicjalizacyjna konstruktora delegowanego

(2) ciało konstruktora delegowanego

(3) ciało konstruktora delegującego

Dla zainteresowanych...

Więcej informacji: C++11 FAQ: Delegating constructors

9.2.4 Konstruktor kopiujący

Obiekty są kopiowane w kilku przypadkach, na przykład podczas inicjalizacji obiektu innym obiektem oraz podczas przekazywania obiektu przez wartość:

```
std::string dots(10, '.');
std::string dots2 = dots;  // kopiowanie obiektu `dots` do `dots2
```

```
void foo(std::string s);
foo(dots);  // przekazywanie do funkcji argumentu PRZEZ WARTOŚĆ
            //   -- też wymaga kopiowania (do parametru `s`)
```

Dany konstruktor może pełnić rolę **konstruktora kopiującego** (ang. copy constructor), jeśli jego pierwszym parametrem jest referencja na typ klasowy[8] (zwykle stała referencja), a ewentualne pozostałe parametry mają domyślne wartości:

```
class Foo {
public:
  Foo();             // konstruktor domyślny
  Foo(const Foo&);   // konstruktor kopiujący
  // ...
};
```

Jeśli sami nie zdefiniujemy konstruktora kopiującego, kompilator *w większości przypadków* wygeneruje **syntezowany konstruktor kopiujący** (ang. synthesized copy constructor) za nas – w przeciwieństwie do konstruktora domyślnego, konstruktor kopiujący zostanie wygenerowany nawet wtedy, gdy zdefiniujemy inne konstruktory[9].

Syntezowany konstruktor kopiujący kopiuje pola z obiektu kopiowanego do konstruowanego obiektu jedno po drugim, w kolejności występowania pól w definicji klasy. W zależności od typu pola kopiowanie odbywa się albo bezpośrednio (dla typów wbudowanych i pochodnych dla nich), albo z użyciem stosownego konstruktora kopiującego (dla pól typu klasowego). Pola będące tablicami są kopiowane element po elemencie zgodnie z powyższą regułą.

Jawnie zdefiniowany konstruktor kopiujący (działający w tym przypadku identycznie jak jego syntezowana wersja) ma następującą postać:

```
class Foo {
public:
  Foo(const Foo& orig) : s_(orig.s_), i_(orig.i_) {}
private:
  std::string s_;
  int i_ = 0;
};
```

9.3 Destruktory

Ogólną ideę destruktora – specjalnej metody określającej sposób niszczenia obiektu klasy – przedstawiono w rozdziale 2.7.2 Destruktor. W tym rozdziale omówione zostaną kwestie istotne dla poprawnego korzystania z destruktorów.

Proces niszczenia instancji klasy `X` przebiega odwrotnie do procesu konstruowania tych instancji, w sposób rekursywny:

(1) wykonanie ciała destruktora zdefiniowanego w klasie `X`

(2) zniszczenie pól zdefiniowanych w klasie `X` (w kolejności odwrotnej niż przy inicjalizacji)

(3) wykonanie destruktora klasy macierzystej dla `X` – o ile klasa `X` jest klasą pochodną *oraz* o ile jednocześnie klasa macierzysta definiuje destruktor jako metodę wirtualną[10]

Ciało destruktora zawiera instrukcje zapewniające poprawne usunięcie obiektu. Dla pól będących typu klasowego wywoływane są ich własne destruktory, natomiast pola typów wbudowanych (i pochodnych dla tych typów) nie wymagają wykonywania żadnych specjalnych operacji niszczenia. Należy jedynie zwrócić uwagę, aby zwolnić ewentualne dynamicznie przydzielone danemu obiektowi zasoby[11].

[8] O tym, dlaczego niezbędna jest referencja, przeczytasz w rozdziale 9.5.3 Kopiowanie a przypisanie.

[9] O tym, kiedy konstruktor kopiujący nie zostanie wygenerowany automatycznie przeczytasz w rozdziale 12 Semantyka przeniesienia.

[10] zob. rozdz. 8.2.2 Wirtualne destruktory

[11] Zagadnienie to zostanie omówione w rozdziale 13 Zarządzanie pamięcią.

Informacja

Destruktor nie jest wykonywany, gdy program opuszcza zakres, w którym zdefiniowano referencję do obiektu lub „surowy" wskaźnik[12] na obiekt – dotyczy to także pól będących referencją bądź „surowym" wskaźnikiem.

9.3.1 Destruktor syntezowany

Dla każdej klasy nie posiadającej ręcznie zdefiniowanego destruktora kompilator zdefiniuje **syntezowany destruktor** (ang. synthesized destructor).

Dzięki wprowadzeniu do standardu C++11 inteligentnych wskaźników programistom potrafiącym efektywnie korzystać z biblioteki standardowej odpadło większość problemów związanych z poprawną implementacją destruktorów zwalniających dynamicznie przydzieloną pamięć[13]. Mimo to wciąż zdarzają się sytuacje, gdy trzeba taki destruktor zdefiniować ręcznie – przykładowo, gdy chcemy skorzystać z mechanizmu dziedziczenia należy w klasie macierzystej zdefiniować destruktor jako metodę wirtualną[14].

9.4 Kopiujący operator przypisania

Każda klasa posiada specjalne metody nie tylko określające sposób inicjalizacji obiektów takiej klasy, ale również sposób przypisywania do niego obiektów tego samego typu – tzw. **kopiujący operator przypisania** (ang. copy assignment operator):

```
Foo foo1;
Foo foo2 = foo1;  // wywołanie kopiującego operatora przypisania klasy Foo
```

Jeśli nie zdefiniujemy kopiującego operatora przypisania ręcznie, kompilator w większości przypadków wygeneruje go za nas[15]. Syntezowany kopiujący operator przypisania przypisuje każdemu polu obiektu po lewej stronie znaku = wartość odpowiednika tego pola z obiektu po prawej stronie (czyli w kolejności analogicznej jak w przypadku syntezowanego konstruktora kopiującego). W praktyce takie rozwiązanie sprawdza się w większości przypadków[16].

Prototyp kopiującego operatora przypisania dla hipotetycznego typu klasowego `T` w większości przypadków ma następującą postać[17]:

```
T& operator=(const T&);
```

czyli nie umożliwia on modyfikacji obiektu stojącego po prawej stronie symbolu „=". Kopiujący operator przypisania dla klas zwraca referencję z dwóch powodów. Po pierwsze, aby zapewnić zgodność jego formy z operatorami przypisania dla typów wbudowanych (i pochodnych dla tych typów). Po drugie, kontenery biblioteki standardowej wymagają, aby operator przypisania zdefiniowany dla typu ich elementów zwracał właśnie referencję na ten typ.

Samą operację przypisania

```
foo2 = foo1;
```

można równie dobrze zapisać równoważnie w następujący sposób:

```
foo2.operator=(foo1);
```

dzięki czemu jasne staje się dla Ciebie, który obiekt zostaje przekazany jako argument operatora przypisania (mimo to zapis z użyciem symbolu „=" jest idiomatyczny).

---

[12] „Surowy" wskaźnik to wskaźnik znany Ci z języka C – standard C++11 wprowadza inteligentne wskaźniki będące w istocie zwykłymi klasami, stąd należało je jakoś odróżnić od „klasycznych" wskaźników.

[13] zob. rozdz. 9.5.2 Zasada trzech, zasada pięciu a zasada zera

[14] zob. rozdz. 8.2.2 Wirtualne destruktory

[15] O tym, kiedy konstruktor kopiujący nie zostanie wygenerowany automatycznie przeczytasz w rozdz. 12 Semantyka przeniesienia.

[16] Operator przypisania nie zostanie wygenerowany m.in. dla klasy posiadającej pole typu `std::unique_ptr` (zob. rozdz. 13.3.1 Szablon klasy `std::unique_ptr`).

[17] O tym, kiedy stosować wersję `T& operator=(T);` możesz przeczytać tutaj.

9.5 Operacje specjalne

Poniżej omówiono kwestie związane z grupą operacji wykonywanych na klasie, zwanych operacjami specjalnymi: z kopiowaniem, przypisywaniem, oraz niszczeniem.

9.5.1 Domyślne operacje specjalne a dziedziczenie

Syntetyzowane operacje kopiowania, przypisania i niszczenia odpowiednio inicjalizują, przypisują, albo niszczą jedno po drugim pola danej klasy. Natomiast w przypadku klas powiązanych relacją dziedziczenia te syntetyzowane operacje w klasie pochodnej dodatkowo inicjalizują, przypisują, albo niszczą pola należące do bezpośredniej klasy macierzystej – z użyciem analogicznych operacji udostępnianych przez tę klasę macierzystą (czyli np. konstruktor kopiujący klasy pochodnej na początku swojego działania wywołuje konstruktor kopiujący klasy macierzystej).

Nie ma znaczenia czy operacja specjalna udostępniana przez klasę macierzystą jest sama syntezowana, czy też zdefiniowana ręcznie – ważne, żeby była dostępna dla klasy pochodnej.

9.5.2 Zasada trzech, zasada pięciu a zasada zera

W pewnych sytuacjach okazuje się, że syntetyzowane wersje operacji specjalnych albo nie są odpowiednie dla naszej klasy, albo w ogóle nie są dostępne[18] – w takiej sytuacji, jeśli ich potrzebujemy, musimy je zaimplementować własnoręcznie.

Aby zapewnić spójne działanie klasy, warto stosować się do tzw. **zasady trzech** (ang. Rule of Three):

> **Definicja: Zasada trzech (Rule of Three)**
>
> Jeśli do poprawnego działania klasa potrzebuje zdefiniowanej przez użytkownika wersji konstruktora kopiującego, operatora przypisania, bądź destruktora, to niemal na pewno potrzebuje zdefiniowania przez użytkownika wszystkich tych trzech operacji.

> **Dla zainteresowanych...**
>
> Ponieważ standard C++11 wprowadził referencje do r-wartości oraz semantykę przeniesienia[19], *zasada trzech* została poszerzona o konstruktor przenoszący oraz o przenoszący operator przypisania – stając się **zasadą pięciu** (ang. Rule of Five). Warto jednak pamiętać, że w takich przypadkach poleganie na syntetyzowanych wersjach zamiast na własnej implementacji zwykle nie jest błędem, a jedynie nie pozwala wykorzystać w pełni możliwości optymalizacji kodu przez kompilator.

Jednak począwszy od standardu C++11 mamy też możliwość efektywnego stosowania **zasady zera** (ang. Rule of Zero).

> **Definicja: Zasada zera (Rule of Zero)**
>
> Klasy powinny być projektowane w taki sposób, aby kwestię zarządzania zasobami mogły w całości pozostawić w gestii innych klas gwarantujących poprawne zarządzanie takimi zasobami (najlepiej klas biblioteki standardowej).

Oto przykłady efektywnego wykorzystania klas biblioteki standardowej w celu uniknięcia konieczności ręcznego zarządzania zasobami:

- Korzystaj z klasy `std::string` zamiast z łańcuchów znaków w stylu C.
- Korzystaj z kontenerów zamiast z tablic dynamicznych do trzymania kolekcji elementów.
- Korzystaj z inteligentnych wskaźników zamiast z „surowych wskaźników”[20].

> **Dla zainteresowanych...**
>
> Obszerniejszy opis powyższych zasad znajdziesz na stronie Rule of Zero (Flaming Dangerzone), a przykłady zastosowania każdej z nich na stronie The rule of three/five/zero (cppreference.com).

[18] Konkretne „życiowe” przykłady takich klas zostaną podane m.in. w rozdziale rozdziale 13 Zarządzanie pamięcią.

[19] zob. rozdz. 12 Semantyka przeniesienia

[20] zob. rozdz. 13 Zarządzanie pamięcią

9.5.3 Kopiowanie a przypisanie

Warto uzmysłowić sobie różnicę między **bezpośrednią inicjalizacją**[21] (ang. direct initialization), a **inicjalizacją poprzez kopiowanie**[22] (ang. copy initialization):

```
// operacje inicjalizacji bezpośredniej
//   (wywołanie właściwego konstruktora zgodnie z typem argumentów)
std::string dots(10, '.');
std::string s1("abc");

// operacje inicjalizacji poprzez kopiowanie
//   (wywołanie konstruktora kopiującego)
//   UWAGA: Znak `=` nie oznacza w tym kontekście przypisania!
std::string s2 = dots;
std::string null_book = "9-999-99999-9";
std::string nines = std::string(100, '9');

void foo(std::string s);
foo(s1);  // przekazywanie do funkcji argumentu PRZEZ WARTOŚĆ
          //   - też wymaga kopiowania
```

W przypadku inicjalizacji bezpośredniej wywoływany jest odpowiedni konstruktor. Inicjalizacja poprzez kopiowanie korzysta z konstruktora kopiującego[23]. Choć w kodzie pojawia się symbol „=", to *nie jest* operacja przypisania – gdyż obiekt stojący po lewej stronie symbolu „=" dopiero jest tworzony! Nie są zatem tworzone żadne dodatkowe obiekty tymczasowe, nie ma narzutu wydajnościowego.

Inicjalizacja poprzez kopiowanie odbywa się nie tylko podczas definiowania zmiennych, lecz również m.in. gdy:

- przekazujemy obiekt do funkcji przez wartość, a nie referencję, bądź
- zwracamy z funkcji obiekt przez wartość, a nie przez referencję.

W szczególności kontenery biblioteki standardowej inicjalizują swoje elementy poprzez kopiowanie (np. podczas wstawiania elementów do kontenera za pomocą metody `push_back()`).

W kontekście powyższego akapitu powinno stać się jasne, dlaczego konstruktor kopiujący musi przyjmować argument poprzez referencję – gdyby przyjmował go przez wartość, otrzymalibyśmy błędne koło, ponieważ żeby wywołać konstruktor kopiujący musielibyśmy najpierw skopiować przekazywany argument z użyciem tego właśnie konstruktora kopiującego...

Dobre praktyki

Większość klas powinna definiować – niejawnie albo jawnie – konstruktor domyślny, konstruktor kopiujący, oraz operator przypisania[24].

9.5.4 `= default`

Rozważ następującą sytuację: potrzebujesz zdefiniować własny jednoargumentowy konstruktor klasy `Foo`, natomiast przydałby Ci się również konstruktor domyślny wykonujący takie samo zadanie, jak syntezowany konstruktor domyślny. Ponieważ jednak *musisz* ręcznie zdefiniować ów jednoargumentowy konstruktor, kompilator nie wygeneruje już automatycznie konstruktora domyślnego (ten jest generowany tylko wówczas, gdy klasa nie posiada ani jednego jawnie zdefiniowanego konstruktora). W takim przypadku zamiast własnoręcznie implementować taki konstruktor domyślny, możesz jawnie poprosić kompilator o wygenerowanie *implementacji* syntezowanego konstruktora domyślnego w poniższy sposób, z użyciem składni „`= default`":

```
Foo() = default;
```

[21] zob. direct initialization (cppreference.com)

[22] zob. copy initialization (cppreference.com)

[23] Choć zapis z użyciem znaku = jest analogiczny co w przypadku przenoszącego operatora przypisania (zob. rozdz. 12 Semantyka przeniesienia).

[24] Dwie inne operacje specjalne – przenoszący konstruktor kopiujący oraz przenoszący operator przypisania – zostały omówione w rozdziale 12 Semantyka przeniesienia.

W podobny sposób można poprosić kompilator o wygenerowanie pozostałych metod specjalnych, m.in.:

```
class Foo {
public:
  Foo() = default;
  Foo(const Foo&) = default;
  Foo& operator=(const Foo&) = default;
  ~Foo() = default;
};
```

Ponieważ składnia „= **default**” odnosi się do ciała metody, a nie do jej nagłówka, w analogiczny sposób można oznaczyć nawet destruktor wirtualny – choć jak wspomniano we wcześniejszych rozdziałach, destruktor syntezowany nie posiada specyfikatora **virtual**:

```
class Foo {
public:
  // ...
  virtual ~Foo() = default;  // równoważne: virtual ~Foo() {}
};
```

9.5.5 = delete

Choć znakomita większość klas powinna definiować konstruktor kopiujący oraz operator przypisania, w niektórych przypadkach takie operacje zwyczajnie nie mają sensu[25] – w takich sytuacjach należy za pomocą składni „= **delete**” jawnie zaznaczyć, że nie chcemy otrzymywać wersji syntezowanej danej operacji specjalnej, przykładowo:

```
class NoCopy {
  NoCopy() = default;
  NoCopy(const NoCopy&) = delete;            // brak możliwości kopiowania
  NoCopy& operator=(const NoCopy&) = delete; // brak możliwości przypisania
  ~NoCopy() = default;
};
```

W przeciwieństwie do składni „= **default**”, składnia „= **delete**” musi pojawić się przy pierwszej deklaracji usuwanej metody.

Z oczywistych względów nie należy usuwać destruktora – wtedy kompilator nie pozwoli nam na tworzenie zmiennych oraz obiektów tymczasowych takiego „wybrakowanego” typu, jak również nie będzie w stanie zwolnić dynamicznie zaalokowanego obiektu tego typu.

Jeśli klasa posiada pole, którego typ nie pozwala na jego domyślne konstruowanie, kopiowanie, przypisywanie, albo usuwanie, wówczas odpowiednia metoda takiej klasy będzie automatycznie oznaczona jako usunięta. W szczególności takimi polami są referencje oraz stałe.

Dla zainteresowanych...

Nim wprowadzono składnię „= **delete**”, klasy uniemożliwiały jej zwykłym użytkownikom tworzenia kopii poprzez deklarowanie konstruktora kopiującego oraz operatora przypisania jako składowych prywatnych:

```
class PrivateCopy {
public:
  PrivateCopy() = default;
  ~PrivateCopy();
private:
  // prywatne metody nie są dostępne dla zwykłego użytkownika kodu
  PrivateCopy(const PrivateCopy&);
  PrivateCopy& operator=(const PrivateCopy&);
};
```

[25] zob. np. rozdz. 13.3.1 Szablon klasy `std::unique_ptr`

Takie rozwiązanie nie zabezpieczało jednak przed tworzeniem kopii przez funkcje i klasy zaprzyjaźnione oraz przez inne składowe tej klasy – chyba, że definicja konstruktora kopiującego i operatora przypisania została celowo pominięta.

9.6 RAII i cykl życia obiektów

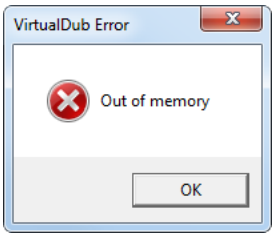
Źródło: RAII (cppreference.com)

Każdy program komputerowy (działający na dowolnym systemie operacyjnym) posiada pewną określoną pulę zasobów przydzielonych przez system operacyjny, które muszą wystarczyć mu na cały okres działania. Zasobami takimi są m.in. obszar w pamięci komputera do przechowywania danych (obszar pamięci programu) oraz pula **uchwytów do pliku**[26] (ang. file handles), umożliwiających otwieranie plików.

Zmorą programistów piszących w języku C++ są tzw. **wycieki zasobów** (ang. resource leaks), czyli bezpowrotna utrata zasobów w trakcie wykonywania danego wykonania programu, gdy przydzielony zasób nie zostanie poprawnie zwolniony, przykładowo:

- Z wyciekiem pamięci mamy do czynienia np. gdy wewnątrz funkcji dokonamy dynamicznej alokacji bloku pamięci, ale jednocześnie ani nie zwolnimy tego bloku wewnątrz funkcji, ani nie zwrócimy wskaźnika na ten blok – wówczas bezpowrotnie tracimy możliwość korzystania z tego bloku pamięci, co w przypadku częstego wywoływania takiej funkcji w pewnym momencie doprowadzi do całkowitego wyczerpania wolnego miejsca w obszarze pamięci programu, a to z kolei spowoduje „ubicie" programu przez system operacyjny (zob. Rys. 9.1).
- Z wyciekiem uchwytów do plików mamy do czynienia, gdy po otwarciu pliku nie zamkniemy go w odpowiedni sposób (np. plik otwarty z użyciem funkcji `std::fopen()` powinien zostać zamknięty z użyciem funkcji `std::fclose()`) – wówczas z czasem możemy całkowicie utracić możliwość otwierania nowych plików.

Język C++ udostępnia mechanizm wyjątków, służących do sygnalizowania anomalii występujących podczas działania programu – zostanie on dokładniej omówiony w rozdziale 15 Wyjątki i ich obsługa, jednak teraz przytoczone zostaną tylko najbardziej elementarne informacje. W skrócie, w momencie zgłoszenia przez program wyjątku („rzucenia" wyjątku) dalsze normalne wykonanie programu zostaje przerwane i program przechodzi do najbliższego miejsca w programie, gdzie znajduje się kod obsługi wyjątku; jeśli takiego kodu nie znajdzie – program zakończy swoje działanie. W związku z tym do wycieku zasobów może dojść także wówczas, gdy pomiędzy kodem odpowiedzialnym za przydział zasobów a kodem odpowiedzialnym za jego (poprawne) zwolnienie zostanie rzucony nieobsłużony wyjątek.



Rysunek 9.1. Przykład komunikatu o błędzie wykonania programu spowodowanym wyciekami pamięci (w systemie operacyjnym Windows 7).

Na listingu 9.7 przedstawiono typowy scenariusz prowadzący do wycieku uchwytów do pliku. W przykładzie tym za bezpośrednią obsługę plików (tekstowych) odpowiada hipotetyczna klasa `File` posiadająca trzy metody: `open()` służącą do otwarcia uchwytu do pliku, `read_line()` odczytującą wiersz z pliku (rzuca wyjątek, gdy nie ma już więcej wierszy do odczytu), oraz `close()` służącą do poprawnego zamknięcia uchwytu pliku.

Listing 9.7. Kod potencjalnie prowadzący do wycieku zasobów.

```
File f;
f.open("foo.txt");

f.read_line();  // NIEBEZPIECZNE - jeśli metoda rzuci wyjątek,
                //   plik nie zostanie nigdy zamknięty!
                //   (gdyż nie ma kodu obsługi wyjątku)
```

[26] zob. File Handles

```
f.close();
```

W przykładzie tym wystąpienie błędu w trakcie działania programu (np. rzucenie wyjątku podczas odczytu z pliku) spowoduje, że uchwyt nie zostanie zamknięty – gdyż instrukcja `f.close();` nie zostanie osiągnięta. Jak uniknąć takiej sytuacji?

Język C++ daje następującą gwarancję: dla każdego obiektu przechowywanego w pamięci na stosie jego destruktor zostanie *zawsze* wywołany – nawet wówczas, gdy rzucony zostanie wyjątek. W związku z tym, aby w przypadku przechowywanych na stosie obiektów danej klasy mieć *gwarancję* zwolnienia przydzielonych w trakcie ich cyklu życia zasobów (np. zwolnienia dynamicznie przydzielonej pamięci, zamknięcia uchwytu do pliku), należy umieścić stosowny kod zwalniający zasób w destruktorze tej klasy.

RAII to akronim techniki programistycznej, w myśl której przetrzymywanie zasobu jest **niezmiennikiem klasy**[27] (ang. class invariant), w dodatku ściśle powiązanym z **cyklem życia** (ang. lifecycle, lifetime) obiektu:

- przydział (*alokacja*) zasobów dokonywana jest podczas tworzenia obiektu (a dokładniej podczas jego inicjalizacji) – odpowiada za to jego konstruktor;
- zwolnienie (*dealokacja*) zasobów następuje podczas niszczenia obiektu – za to odpowiada jego destruktor.

W związku z tym mamy zawsze gwarancję, że:

- zasób będzie przetrzymywany w okresie pomiędzy końcem procesu inicjalizacji a początkiem procesu niszczenia, oraz że
- zasób będzie przetrzymywany tylko wówczas, gdy obiekt jest „żywy".

Ponieważ w takim mechanizmie nie dochodzi do wycieku obiektów, pozwala on również uniknąć wycieku zasobów.

Rozwinięcie akronimu RAII to „*Resource Acquisition Is Initialization*" – zatem przydział zasobów (człon *resource acquisition*) powinien następować w momencie inicjalizacji (człon *is initialization*). Niestety, akronim ten nie informuje o rzeczy kluczowej – że *zwolnienie zasobów powinno odbywać się podczas niszczenia obiektu...*

Poniższy przykład demonstruje definicję hipotetycznej klasy `FileHandler` „opakowującej" użycie klasy `File`, przy czym klasa `FileHandler` została napisana w zgodzie z techniką RAII:

Listing 9.8. Bezpieczny kod, napisany zgodnie z techniką RAII, w którym unikamy wycieku zasobów.

```
class FileHandler {
public:
    FileHandler(const char* filename){
        // Przyjmij, że metoda `File::open()` rzuca wyjątek w przypadku
        //   niemożliwości otwarcia pliku -- dzięki temu błąd otwarcia
        //   pliku powoduje przerwanie wykonania konstruktora obiektu
        //   klasy FileHandler (bo rzucony przez tę metodę wyjątek nie
        //   jest wychwytywany w niniejszym konstruktorze) i w efekcie
        //   zasoby przeznaczone dla nowo tworzonego obiektu klasy
        //   FileHandler zostaną automatycznie zwolnione.
        file_.open(filename);
    }

    ~FileHandler(){
      // Jawne zwolnienie zasobów następuje w destruktorze -- aby uniknąć
      //    ich wycieku. (Destruktor jest wykonywany ZAWSZE na koniec
      //    cyklu życia obiektu.)
        file_.close();
    }

    std::string read_line() {
        return file_.read_line();
    }
```

[27] zob. Class invariant (Wikipedia)

```
private:
    File file_;
};

/* ... */

FileHandler fh("foo.txt");  // alokacja na stosie

// Wywołanie bezpieczne dla wyjątków -- w przypadku rzucenia wyjątku
//   (i opuszczania bieżącego zakresu) zostanie wywołany destruktor
//   klasy FileHandler, który spowoduje zamknięcie uchwytu do pliku.
fh.read_line();
```

Miej na uwadze, że powyższa klasa `FileHandler` została utworzona wyłącznie w celach demonstracyjnych! Współczesna biblioteka standardowa języka C++ zawiera szereg funkcjonalności służących do obsługi zasobów zgodnie z RAII (np. strumienie plikowe do obsługi plików, inteligentne wskaźniki do dynamicznego zarządzania pamięcią) – korzystając z nich umiejętnie unikasz konieczności zadbania samemu o zwalnianie zasobów. Dodatkowo, pisząc kod pamiętaj o odpowiedniej obsłudze wyjątków, tak aby napisany przez Ciebie kod był bezpieczny dla wyjątków (ang. exception-safe).

Zwróć uwagę, że samo korzystanie z klas napisanych zgodnie z RAII nie wystarcza do uniknięcia wycieku zasobów – choć język C++ gwarantuje wywołanie destruktora podczas niszczenia obiektu, to jedynie obiekty zaalokowane w pamięci na stosie są zawsze niszczone! W przypadku obiektów alokowanych na stercie takiej gwarancji nie ma:

Listing 9.9. RAII: alokacja na stosie a na stercie

```
std::string first_line_of__stack(const char* filename){
    FileHandler f("foo.txt");  // alokacja na stosie
    return f.read_line();
    // (w tym miejscu obiekt `f` jest usuwany)
}

std::string first_line_of__heap(const char* filename){
    FileHandler* f_ptr = new FileHandler("foo.txt");  // alokacja na stercie
    return f_ptr->read_line();
    // (w tym miejscu obiekt `f_ptr` jest usuwany)
    //   UWAGA: Obiekt WSKAZYWANY przez `f_ptr` NIE jest w tym miejscu
    //     usuwany - dochodzi do wycieku pamięci!
}
```

Rozdział 10

Klasy: Varia

Niniejszy rozdział omawia zagadnienia, które pozwalają efektywnej korzystać z klas. Zagadnienia te nie są jednak kluczowe dla ogólnego zrozumienia podstawowych mechanizmów obiektowości w języku C++.

10.1 Wskaźnik `this`

Niejawny wskaźnik **`this`** przechowuje adres obiektu, dla którego została wywołana metoda, np.:

```
#include <cstdlib>

class Foo {
private:
  int counter_ = 1;
public:
  Foo& mult2() {
    counter_ *= 2; // Użycie `this` nie jest potrzebne.
    return *this;  // Zwróć referencję do obiektu, dla którego została
                   //   wywołana ta metoda.
  }
};

int main() {
  Foo foo;
  foo.mult2().mult2();  // Zwrócenie referencji do obiektu wywołującego metodę
                        //   pozwala na tworzenie łańcuchów operacji.
  // pole `foo.counter_` ma teraz wartość 4
  return EXIT_SUCCESS;
}
```

„Niejawność” wskaźnika **`this`** polega na tym, że nie musimy go sami deklarować – zgodnie ze standardem języka C++ jest on domyślną składową klasy, zapewnianą przez kompilator.

Zauważ, że nie musimy odwoływać się do składowych klasy z użyciem **`this`** – wskaźnik ten jest niezbędny tylko wtedy, gdy potrzebujemy odwołać się do obiektu wywołującego jako całości[1].

W powyższym przykładzie instrukcja

```
foo.mult2().mult2();
```

to przykład często stosowanego idiomu zwanego **method chaining** – w tym przypadku wspomniana instrukcja jest równoważna następującym dwóm wywołaniom:

```
foo.mult2();
foo.mult2();
```

Użycie *method chaining* pozwala na pisanie bardziej zwięzłego, czytelniejszego kodu.

[1] Inaczej niż w języku Python, gdzie odwołanie się do składowej instancji klasy *musi* odbywać się poprzez `self`.

Dobre praktyki

Stosuj wskaźnik **`this`** wyłącznie *wówczas, gdy musisz odwołać się do obiektu wywołującego daną metodę jako* do całości *– przykładowo do zwrócenia odpowiedniej wartości z kopiującego operatora przypisania:*

```
T& T::operator=(const T&) {
  /* ... */
  return *this;
}
```

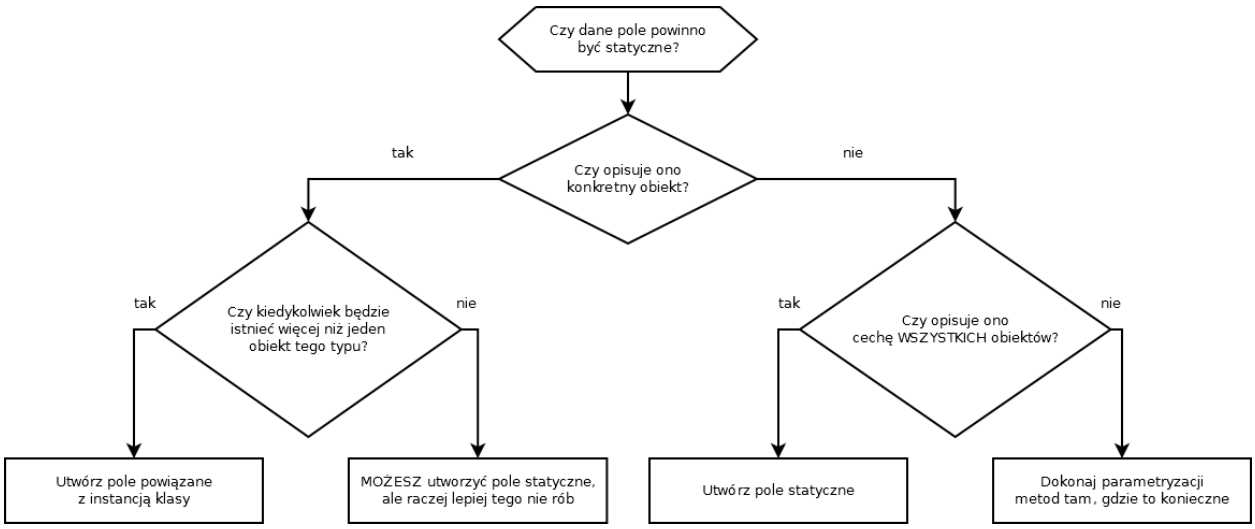
a nie każdorazowo w celu odwołania się do składowej obiektu wywołującego daną metodę:

```
int Foo::get_x() const {
  return this->x;  // ANTY-przykład stosowania `this`
}
```

10.2 Składowe statyczne

Czasem zachodzi potrzeba, aby *wszystkie* obiekty danej klasy korzystały ze wspólnej składowej – przykładowo, w pewnym banku wszystkie obiekty klasy reprezentującej rachunek bankowy powinny odwoływać się do tego samego pola określającego wysokość stopy procentowej. W takim przypadku możemy powiązać wspomniane pole z samą klasą reprezentującą dany typ rachunku bankowego, a nie z pojedynczą instancją tej klasy – pozwala to zwiększyć spójność danych oraz uniknąć ich nadmiarowego przechowywania[2]. Analogiczna sytuacja może się odnosić do metod klasy – metoda statyczna powiązana z daną klasą (a nie z jej instancjami) może m.in. operować na polach statycznych bez konieczności tworzenia instancji tej klasy. Składowe statyczne podlegają normalnym zasadom dziedziczenia, z tym że każda klasa potomna również posiada tylko jeden egzemplarz danej składowej statycznej.

Schematy na rysunkach 10.1 oraz 10.2 przedstawiają *wskazówki* (a nie sztywne wytyczne!), kiedy dana składowa powinna być składową statyczną.



Rysunek 10.1. Kiedy pole powinno być statyczne?

10.2.1 Deklarowanie statycznych składowych

Składową deklarujemy jako statyczną poprzez poprzedzenie jej typu specyfikatorem **`static`**:

```
#include <set>

// Klasa `Account` reprezentuje rachunek bankowy.
```

[2]Wystarczy jeden egzemplarz takiego współdzielonego obiektu danych powiązanego z klasą, nie trzeba przechowywać jego duplikatów w każdej z instancji tej klasy.

Rysunek 10.2. Kiedy metoda powinna być statyczna?

```
class Account {
public:
  // Dolicz odsetki do depozytu.
  void capitalize() { amount_ += amount_ * interest_rate_; }

  // Zwróć wartość stopy oprocentowania.
  static double get_rate() { return interest_rate_; }  // metoda statyczna

  // Ustaw nową wartość stopy oprocentowania.
  static void set_rate(double new_rate);               // metoda statyczna
private:
  // Kwota depozytu.
  double amount_;

  // Stopa procentowa - wspólna dla wszystkich rachunków.
  static double interest_rate_;  // pole statyczne

  // Zbiór identyfikatorów rachunków (tylko w celach poglądowych).
  static set<int> ids_;          // pole statyczne
};
```

(Pole `ids_` zostało dodane wyłącznie po to, aby później pokazać inicjalizację statycznych składowych będących kontenerami.)

Metody statyczne nie mogą posiadać kwalifikatora **`const`**, gdyż kwalifikator ten w kontekście metod oznacza, że „metoda nie modyfikuje stanu obiektu" – a metoda statyczna nie jest wywoływana dla żadnego obiektu (tylko dla pewnego typu)[3].

10.2.2 Definiowanie składowych statycznych

Metody statyczne definiujemy podobnie jak metody niestatyczne – mamy zatem dwie możliwości: albo wewnątrz definicji klasy, albo poza nią. Gdy definiujemy metodę statyczną poza definicją klasy, nie powtarzamy słowa kluczowego **`static`** w definicji:

```
void Account::set_rate(double new_rate) {  // UWAGA: W definicji nie dodajemy
                                           //   `static` -- `static` pojawia
                                           //   się tylko w deklaracjach!
  interest_rate_ = new_rate;
}
```

Ponieważ pola statyczne klasy nie są częścią jej instancji, pola te nie są inicjalizowane podczas tworzenia instancji takiej klasy (przez któryś z jej konstruktorów), a znacznie wcześniej – w momencie uruchamiania programu. W związku z tym aby zainicjalizować pole statyczne należy wybrać jedną z dwóch możliwości:

- Aby zainicjalizować pole statyczne wewnątrz definicji klasy, z użyciem inicjalizatora wewnątrzklasowego[4], należy oznaczyć je z użyciem specyfikatora **`inline`**[5]:

[3] Mimo, że moglibyśmy oczekiwać, że kwalifikator **`const`** dla metody statycznej odnosiłby się do pól statycznych.

[4] zob. rozdz. 9.1.1 Inicjalizator wewnątrzklasowy

[5] Rozwiązanie wprowadzone w standardzie C++17.

```
class Account {
// ...
private:
  inline static double interest_rate_ = 1.0;
  inline static set<int> ids_;  // wywołany zostanie konstruktor
                                //   domyślny: set<int>()
};
```

To rozwiązanie jest zalecane ze względu na przejrzystość kodu.

- Aby zainicjalizować pole statyczne poza definicją klasy należy tego dokonać w pliku **źródłowym**[6], używając nazwy kwalifikowanej pola (tj. z użyciem operatora zakresu):

```
/* plik ŹRÓDŁOWY z dołączoną definicją klasy Account */
// zdefiniuj i zainicjalizuj pola statyczne klasy `Account`
double Account::interest_rate_ = 1.0;
set<int> Account::ids_;  // wywołany zostanie konstruktor domyślny
// UWAGA: Zwróć uwagę, że w definicjach nie występuje `static`!
```

Pole statyczne można zainicjalizować tylko raz!

Każda metoda statyczna jest powiązana z całą klasą, dlatego w jej ciele nie ma możliwości korzystania ze wskaźnika `this` (ani odwoływania się do jakiejkolwiek składowej *niestatycznej*) – można odwoływać się wyłącznie do innych składowych statycznych.

Dla zainteresowanych...

Ponieważ pola statyczne klasy istnieją niezależnie od jej instancji, pole statyczne może być typu klasy, która zawiera to pole:

```
struct Foo {
  static Foo m1; // poprawne - pole statyczne może mieć niekompletny typ
  Foo* m2;       // poprawne - pole będące wskaźnikiem może mieć
                 //   niekompletny typ
  Foo m3;        // BŁĄD: "zwykłe" pole niestatyczne musi mieć
                 //   kompletny typ
};
```

10.2.3 Korzystanie ze składowych statycznych

Co do zasady, dostęp do składowych statycznych uzyskujemy za pomocą operatora zakresu:

```
double r;
r = Account::get_rate();  // dostęp do składowej statycznej
                          //   z użyciem operatora zakresu
```

Choć składowe statyczne klasy `T` nie są częścią instancji tej klasy, dla wygody programisty twórcy języka C++ umożliwili uzyskanie dostępu do składowej statycznej za pomocą obiektów typu `T` oraz referencji i wskaźników na typ `T`, przykładowo:

```
Account ac;
Account& ac_ref = ac;
Account* ac_ptr = &ac;

// Inne, równoważne sposoby dostępu do składowej statycznej:
r = ac.get_rate();       // *) poprzez obiekt typu `Account`
r = ac_ref.get_rate();   // *) poprzez referencję do obiektu typu `Account`
r = ac_ptr->get_rate();  // *) poprzez wskaźnik na obiekt typu `Account`
```

Metody związane z instancją klasy mają bezpośredni dostęp do składowych statycznych (tj. nie muszą korzystać z operatora zakresu):

[6] Inaczej konsolidator zgłosi błąd „`multiple definition of ...`”.

```
class Account {
public:
  void calculate() { amount_ += amount_ * interest_rate_; }
  // ...
private:
  static double interest_rate_;
  // ...
};
```

Poniższy przykład raz jeszcze pokaże Ci, że wewnątrz metod statycznych nie można korzystać ze wskaźnika `this` (ani odwoływać się do jakiejkolwiek składowej *niestatycznej*) – można odwoływać się wyłącznie do innych składowych *statycznych*[7]:

```
class SomeClass {
public:

    SomeClass() : my_id_(next_id_++) {}

    static void some_static_method(int n) {
        n = 5;
        my_id_ = 5;  // NIEDOZWOLONE - dostęp do składowej związanej
                     //   z instancją klasy w metodzie statycznej
        next_id_ = 5;
        some_static_method(5);
        some_instance_method(5);  // NIEDOZWOLONE - dostęp do składowej
                                  //   związanej z instancją klasy
                                  //   w metodzie statycznej
    }

    void some_instance_method(int n) {
        n = 5;
        my_id_ = 5;  // to samo, co: this->my_id_
        next_id_ = 5;
        some_static_method(5);
        some_instance_method(5);
    }

private:
    int my_id_;
    inline static int next_id_ = 0;
};
```

10.3 Dziedziczenie a kontenery biblioteki standardowej

Chcąc przechowywać w jednym kontenerze obiekty z różnych poziomów hierarchii dziedziczenia musimy to robić nie wprost, gdyż przechowywanie obiektów typu macierzystego (a nie referencji do typu macierzystego lub wskaźników na typ macierzysty) doprowadziłoby do obcięcia wstawianych elementów[8]. Rozwiązaniem tego problemu jest przechowywanie wskaźników (najlepiej wskaźników inteligentnych[9]) na typ macierzysty i wykorzystanie mechanizmu polimorfizmu[10]:

Listing 10.1. Przechowywanie w tym samym kontenerze obiektów różnych typów z tej samej hierarchii dziedziczenia

```
#include <cstdlib>
#include <vector>

class Base {};
```

[7] W rzeczywistej implementacji aby uniknąć błędów kompilacji obie metody musiałyby być zdefiniowane poza klasą!
[8] zob. rozdz. 8.2.1 Zasada „nie ma niejawnej konwersji między obiektami”
[9] zob. rozdz. 13.3.1 Szablon klasy `std::unique_ptr`
[10] zob. rozdz. 8.2 Polimorfizm

```
class Derived : public Base {};

int main() {
    // UWAGA: Począwszy od C++11 w tej sytuacji zamiast "surowego" wskaźnika
    //   powinniśmy użyć jak typu elementów albo kontenera `std::unique_ptr`,
    //   albo kontenera `std::reference_wrapper`.
    std::vector<Base*> vb;

    Base b;
    vb.push_back(&b);
    Derived d;
    vb.push_back(&d);

    return EXIT_SUCCESS;
}
```

10.4 enum class

Wyliczenie służy grupowaniu zbioru stałych wartości typu całkowitego, jakie może przyjmować zmienna tego typu. Wartości te reprezentowane są za pomocą literałów wyliczeniowych. Przykładami wyliczeń są: dni tygodnia, miesiące w roku, wartości logiczne (prawda albo fałsz) itp.

Używając „klasycznych” (znanych z języka C) typów wyliczeniowych dochodzi czasem do problemu kolizji nazw wartości – problem ten wynika z faktu, że do zakresu widoczności nazw dodawane są wszystkie wartości ze wszystkich typów wyliczeniowych. Jeśli w tym samym zakresie potrzebujemy inny typ wyliczeniowy i powinien on zawierać pewne wartości o tych samych nazwach, kompilator nie dopuści do tego ze względu na powtórzenia nazw.

Przykładowo, gdybyśmy mieli już w aktualnej przestrzeni nazw zdefiniowany typ `RgbColor` opisujący składowe koloru na obrazie RGB:

```
enum RgbColor {
  RED,
  GREEN,
  BLUE
};
```

a dodatkowo chcieli dodać typ wyliczeniowy `TrafficLightsColor` dla kolorów świateł na skrzyżowaniach:

```
enum TrafficLightsColor {
  RED,
  YELLOW,
  GREEN
};
```

to w momencie próby zdefiniowania typu wyliczeniowego `TrafficLightsColor` pojawi się konflikt, gdyż wartości `RED` i `GREEN` są już zajęte przez typ wyliczeniowy `Color`.

Począwszy od C++11 możemy uniknąć takiego konfliktu nazw poprzez definiowanie „klasowego” typu wyliczeniowego – **`enum class`** – którego podstawowa definicja wygląda następująco:

```
enum class EnumName {
  value1,
  value2,
  ...
  valueN
};
```

Oto przykład użycia powyższego typu:

```
enum class RgbColor {
  RED,
  GREEN,
  BLUE
```

```
};

auto color = RgbColor::RED;
```

Uwagi odnośnie korzystania z **enum class**:

- Nazwy wartości muszą być zawsze kwalifikowane nazwą typu wyliczeniowego, np. `EnumName::valueX`.
- Wartości nie są niejawnie konwertowane na typy całkowite, zatem nie mogą być porównywane z typami całkowitymi (takie porównanie spowoduje błąd kompilacji)[11].
- Można wykorzystać zmienną typu wyliczeniowego jako wyrażenie w instrukcji **switch**, a poszczególne elementy jej typu wyliczeniowego jako etykiety **case**.

Dla zainteresowanych: C++11 FAQ: enum class

[11] To różnica w stosunku do „zwykłych” typów wyliczeniowych, znanych z języka C.

Rozdział 11

System typów

11.1 Niejawne konwersje typów dla klas

Język C++ definiuje kilka rodzajów automatycznych (niejawnych) konwersji między typami wbudowanymi, przykładowo z typu `int` na typ `double`. Klasy mogą definiować podobne niejawne konwersje: każdy konstruktor, który przyjmuje pojedynczy argument, definiuje niejawną konwersję do typu klasowego – traktowany jest on jako **konstruktor konwertujący** (ang. converting constructor). Oto przykład definiowania i (niejawnego) użycia konstruktora konwertującego:

```
#include <cstdlib>
#include <string>

class Foo {
public:
  // konstruktor KONWERTUJĄCY
  //   (bo przyjmuje pojedynczy argument typu innego niż `Foo`)
  Foo(const std::string& s) : n(s.size()) {}
private:
  std::size_t n;
};

void bar(const Foo& f) { /* ... */ }

int main() {
  std::string str = "abc";
  bar(str);  // `bar()` oczekuje argumentu typu `Foo`, zatem przekazując
             //   obiekt typu `std::string` wywołany zostanie niejawnie
             //   konstruktor konwertujący `Foo(const std::string&)`
  return EXIT_SUCCESS;
}
```

Należy jednak pamiętać, że kompilator jest w stanie wykonać tylko jedną niejawną konwersję na raz:

```
// BŁĄD: wymagane dwie zdefiniowane przez użytkownika konwersje:
//  (1) konwertuj "9-999-99999-9" typu `const char[]` na typ `std::string`
//  (2) konwertuj ten (tymczasowy) obiekt typu `std::string` na typ `Foo`
bar("9-999-99999-9");
```

W takich przypadkach musimy jedną z konwersji określić jawnie, z użyciem odpowiedniego konstruktora:

```
// OK: jawna konwersja do std::string, niejawna do Foo
bar(std::string("9-999-99999-9"));
// OK: niejawna konwersja do std::string, jawna do Foo
bar(Foo("9-999-99999-9"));
```

11.2 Operatory rzutowania

Zdarza się, że chcemy wymusić jawną konwersję typu obiektu. Przykładowo, w poniższym przykładzie chcemy użyć dzielenia zgodnego z arytmetyką liczb zmiennoprzecinkowych (a nie całkowitych):

```
int i = 1;
int j = 2;
double slope = i / j;
```

W tym celu musimy wymusić konwersję wartości zmiennej `i` i/lub zmiennej `j` na typ **`double`**.

> **Ostrzeżenie**
>
> *Choć w pewnych przypadkach jawne rzutowanie jest niezbędne, generalnie jest to bardzo niebezpieczna konstrukcja języka C++.*

Wszystkie operatory rzutowania w języku C++ mają następującą postać:

```
XXX_cast<type>(expression);
```

gdzie `XXX_cast` określa rodzaj rzutowania, `type` – typ docelowy, a `expression` – wartość, którą rzutujemy.

Język C++ udostępnia kilka operatorów rzutowania, natomiast w niniejszym skrypcie zostaną omówione tylko dwa z nich (najczęściej spotykane): **`static_cast`** oraz **`dynamic_cast`**.

Dokładniejszy opis działania wspomnianych operatorów (wraz z przykładami) znajdziesz na stronie Type conversions (cplusplusreference.com).

11.2.1 Operator `static_cast`

Operator **`static_cast`** pozwala m.in. na jawne dokonanie tych konwersji, które zwykle są dokonywane niejawnie (np. między typami całkowitymi i zmiennoprzecinkowymi) oraz na konwersję między wskaźnikami na powiązane klasy (i to nie tylko „w górę”, ale także „w dół”).

W przytoczonym wcześniej przykładzie dzielenia dwóch liczb całkowitych możemy zastosować operator **`static_cast`** w następujący sposób:

```
int i = 1;
int j = 2;
double slope = static_cast<double>(i) / j;
```

Kompilator nie weryfikuje, czy żądana konwersja faktycznie ma sens – w rezultacie to na programiście spoczywa odpowiedzialność za zapewnienie bezpieczeństwa danej konwersji. Rozważ poniższy przykład:

```
class Base {};
class Derived: public Base {};
Base* a = new Base;
Derived* b = static_cast<Derived*>(a);
```

Powyższy kod jest poprawny pod względem składniowym, mimo że wskaźnik `b` wskazywałby na niekompletny obiekt klasy, co mogłoby doprowadzić do błędów wykonania programu w przypadku dereferencji takiego wskaźnika.

11.2.2 Operator `dynamic_cast`

Podobnie jak **`static_cast`**, operator **`dynamic_cast`** umożliwia konwersję „w górę” i „w dół” dla wskaźników i referencji na klasy, z tym że dodatkowo gwarantuje, że wynikiem rzutowania będzie poprawny (kompletny) obiekt pożądanego typu.

Podstawowe formy operatora **`dynamic_cast`** to:

```
dynamic_cast<type*>(expression)
dynamic_cast<type&>(expression)
```

gdzie `type` musi być typu klasowego, a `expression` to:

- albo obiekt klasy dziedziczącej *publicznie* po `type`,
- albo *publiczna* klasa macierzysta klasy `type`,
- albo ta sama klasa co `type`.

Jeśli `expression` nie spełni powyższych kryteriów, operator **`dynamic_cast`** odpowiednio:

- zwróci wskaźnik pusty w przypadku rzutowania na wskaźnik, albo

- rzuci wyjątek[1] `bad_cast` w przypadku rzutowania na referencję.

Przykład 11.1 pokazuje przykład użycia operatora `dynamic_cast`.

Listing 11.1. Przykład użycia operatora `dynamic_cast`.

```
#include <cstdlib>
#include <iostream>

class Base {
  public:
    void dummy() { /* ...*/ }
    virtual ~Base() = default;
};

class Derived: public Base {};

int main () {
    Base b;
    Derived d;
    Base* pbd = &d;

    // Ponieważ typ dynamiczny `*pbd` to `Derived`, poniższa konwersja powinna
    //   zakończyć się pomyślnie.
    Derived* pd = dynamic_cast<Derived*>(pbd);
    if (pd == nullptr) {
        std::cout << "(1) Null pointer on first type-cast." << std::endl;
    }

    // Poniższa konwersja nigdy się nie powiedzie, gdyż `Derived` to klasa
    //   pochodna od `Base` -- zatem `pd2` wskazywałby na niekompletny obiekt.
    Derived* pd2 = dynamic_cast<Derived*>(&b);
    if (pd2 == nullptr) {
        std::cout << "(2) Null pointer on first type-cast." << std::endl;
    }

    return EXIT_SUCCESS;
}
```

Oto rezultat wykonania powyższego programu (z wyłączoną flagą `-Werror`):

```
(2) Null pointer on first type-cast.
```

[1] zob. rozdz. 15 Wyjątki i ich obsługa

Rozdział 12

Semantyka przeniesienia

Standard C++11 wprowadził **semantykę przeniesienia** (ang. move semantics), związaną z pojęciem **własności** (ang. ownership) i mechanizmem RAII[1] – *własność* określa, „kto" ma prawo do odwoływania się do danych zasobów (i kto ma być odpowiedzialny za ich późniejsze zwolnienie).

12.1 Po co idea własności?

Wcześniejsze standardy języka C++ zakładały, że obiekt posiada swoje zasoby *na wyłączność* i można go co najwyżej *skopiować*. Zastanów się jednak nad przykładowym scenariuszem – obiekt lokalny ma zostać umieszczony w kontenerze standardowym, przy czym obiekt ten nie będzie więcej wykorzystywany w swoim zasięgu:

```
void add_empty_element(std::vector<T>& v) {
  T obj;
  v.push_back(obj);
}
```

Metoda `std::vector<T>::push_back(const T&)` dokonuje *kopiowania* przekazanego obiektu (czyli stworzenia *nowego* obiektu typu `T` i umieszczeniu go w kontenerze), a obiekt przekazywany jako argument jest później usuwany. Zwróć uwagę, że w powyższej sytuacji oznacza to niepotrzebną utratę wydajności – konieczność dodatkowej alokacji zasobów i wykonania kopiowania (dla nowego obiektu) oraz zwolnienia zasobów (dla starego obiektu). Mechanizmy i elementy języka związane z semantyką wartości pozwalają uniknąć tej niedogodności. Nim jednak przejdziemy do ich omówienia, konieczne będzie wprowadzenie pojęcia *referencji do r-wartości*.

12.2 Referencje do r-wartości

Podobnie jak w języku C, r-wartości to w uproszczeniu[2] obiekty tymczasowe, które *nie mogą* występować po lewej stronie wbudowanego w język C++ operatora przypisania oraz dla których *nie można* zastosować operatora adresu, m.in. literały oraz wyrażenia arytmetyczne, logiczne i porównania.

Klasyczna referencja – czyli referencja do l-wartości, oznaczana z użyciem przyrostka `&` – umożliwia zdefiniowanie aliasu dla „istniejącego" obiektu („zasada kciuka" brzmi: *jeśli coś posiada nazwę – jest l-wartością*):

```
T t;
T& t_ref1 = t;     // OK -- t to istniejący obiekt
T& t_ref2 = T();   // BŁĄD: T() tworzy obiekt TYMCZASOWY
```

Z kolei wprowadzona w standardzie C++11 referencja do r-wartości – oznaczana z użyciem przyrostka `&&` – pozwala na wydłużenie cyklu życia obiektów tymczasowych:

```
T t;
T&& t_ref1 = t;     // BŁĄD: t NIE JEST r-wartością
T&& t_ref2 = T();   // OK -- T() tworzy obiekt tymczasowy
```

[1] zob. rozdz. 9.6 RAII i cykl życia obiektów

[2] Drobiazgowe omówienie tego, które wartości są r-wartościami znajdziesz tu.

Szczegółowe omówienie semantyki r-wartości znajdziesz na stronie C++ Rvalue References Explained (Thomas Becker).

12.3 Zawłaszczanie zasobów

Standard C++11 wprowadził też możliwość dokonania **przeniesienia praw własności** do zasobu (ang. ownership transfer) pomiędzy obiektami, czyli operacji zwanej inaczej **zawłaszczeniem** (ang. ownership claim) – służy do tego standardowa funkcja `std::move()` zadeklarowana w pliku nagłówkowym `<utility>`. Sama funkcja `std::move()` nie tyle dokonuje faktycznego transferu zasobów, co służy do *oznaczenia*, że zasób przekazany jako argument może zostać „przeniesiony” – funkcja ta zwraca referencję do r-wartości[3] odnoszącą się do obiektu przekazanego jako argument.

Aby faktycznie zawłaszczyć zasób, należy skorzystać np. z odpowiednio zaimplementowanego konstruktora przenoszącego lub z przenoszącego operatora przypisania:

- **Konstruktor przenoszący** (ang. move constructor) klasy `T` to konstruktor, którego postać zwykle[4] wygląda następująco:

```
T( T2&& );
```

  przy czym `T2` oznacza dowolny typ, w szczególności może być on tożsamy z `T`.
  Konstruktor przenoszący zwykle nie tyle kopiuje zasoby przetrzymywane przez argument (zasobami tymi mogą być np. wskaźniki na dynamicznie zaalokowane obiekty, deskryptory plików, strumienie I/O itp.), co je „kradnie” pozostawiając argument w pewnym poprawnym, lecz niekoniecznie dokładnie określonym stanie (choć przykładowo konstruktor przenoszący dla szablonu klasy `std::vector` *zawsze* pozostawia argument w stanie „pustym” (tj. usuwa wszystkie elementy z danego kontenera), a w hipotetycznej klasie do obsługi listy pojedynczo wiązanej konstruktor przenoszący mógłby ustawić wartość wskaźnika na „głowę” listy na `nullptr`).
- **Przenoszący operator przypisania** (ang. move assignment operator) klasy `T` zachowuje się podobnie jak konstruktor przenoszący, przy czym jego typowy prototyp wygląda następująco:

```
T& operator= ( T2&& );
```

Zwróć uwagę, że jeśli wszystkie typy pól klasy `T` definiują daną przenoszącą operację specjalną, możliwe jest dokonanie syntetyzacji tej operacji w klasie `T` z użyciem „`= default`”. Podobnie w przypadku każdej z tych przenoszących operacji specjalnych można za pomocą „`= delete`” jawnie określić, że nie powinna być ona syntetyzowana.

Przykładowe użycie funkcji `std::move()` wygląda zatem następująco (oczywiście przy założeniu, że typ `T` definiuje „klasyczny” przenoszący operator przypisania o prototypie `T& operator=(T&&)`):

```
T obj1;
T obj2 = std::move(obj1);  // Zmiana właściciela zasobów: z `obj1` na `obj2`.
// Od tego momentu co do zasady nie powinniśmy więcej korzystać z `obj1`.
```

Dla zainteresowanych...

Przekazywanie do funkcji argumentu przez wartość w przypadku obiektu, który nie umożliwia jego regularnego kopiowania, a jedynie definiuje konstruktor przenoszący (na przykładzie hipotetycznego typu `U` – od „unikalny właściciel”), czyli np. w poniższym przypadku:

```
void foo(U obj_param) {
  /* ... */
}

/* ...*/

U obj;
```

[3] Ściślej, `std::move()` zwraca referencję do x-wartości, będącej jednym ze szczególnych przypadków r-wartości.

[4] O pewnym szczególnym przypadku, gdy konieczne jest zdefiniowanie konstruktora o prototypie `T(const T2&&)`, przeczytasz tu.

```
foo(std::move(obj));
```

przebiega w następujący sposób:

- Przed wykonaniem ciała funkcji `foo()` obliczane są wartości jej argumentów – w tym przypadku wywoływana jest funkcja `std::move(obj)`, która zwraca referencję do r-wartości na obiekt `obj` (dla przypomnienia, funkcja `std::move()` nie dokonuje transferu zasobów, a jedynie jest swego rodzaju operatorem rzutowania).
- Ponieważ funkcja `foo()` przyjmuje argument przez wartość, w momencie jej wywołania utworzony nowy obiekt typu `U` – do jego utworzenia zostaje użyty konstruktor przenoszący `U(U&&)` wywołany z argumentem będącym w istocie r-referencją do obiektu `obj`.

12.4 Semantyka przeniesienia a wydajność programu (i wygoda pisania kodu)

Zdefiniowanie (lub przeciążenie) metody z wykorzystaniem referencji do r-wartości w połączeniu z użyciem funkcji `std::move()` pozwala przede wszystkim poprawić wydajność programu – przykładowo, we wspomnianej wcześniej sytuacji:

```
void add_empty_element(std::vector<T>& v) {
  T obj;
  v.push_back(obj);
}
```

po przeciążeniu metody `push_back()` i zdefiniowaniu jej drugiej wersji o poniższym prototypie:

```
void push_back( T&& value );
```

możliwe staje się uniknięcie niepotrzebnego kopiowania:

```
void add_empty_element(std::vector<T>& v) {
  T obj;
  v.push_back(std::move(obj));  // Uniknij niepotrzebnego kopiowania.
}
```

przy czym powyższy zapis można jeszcze bardziej uprościć – zamiast tworzyć l-wartość i oznaczać ją jako „do przeniesienia” z użyciem funkcji `std::move()` można po prostu utworzyć tymczasowy obiekt `T()` w obrębie wywołania metody `push_back()`:

```
void add_empty_element(std::vector<T>& v) {
  v.push_back(T());  // Uniknij niepotrzebnego wywołania std::move() oraz
                     //   "zaśmiecenia" zasięgu.
}
```

Dodatkową zaletą tworzenia obiektów tymczasowych jest to, że w ten sposób unikamy „zaśmiecania” zasięgu niepotrzebnymi identyfikatorami.

Ważne

Wiele kontenerów z biblioteki standardowej zawiera specjalne metody służące do obsługi r-wartości, zawierające w nazwie `emplace`, pozwalające na efektywniejsze wstawianie elementów do kontenera[5] np. `std::vector<T,A>::emplace_back()`, `std::map<K,T,C,A>::emplace()`, `std::list<T,A>::emplace_front()` itp.

[5] zob. porównanie operacji wykonywanych podczas wstawiania elementu do kontenera `std::vector` z użyciem metod `push_back()` i `emplace_back()` dostępne tutaj

Rozdział 13

Zarządzanie pamięcią

Zarządzanie pamięcią to jeden z kluczowych mechanizmów języka C++, pozwalający na tworzenie obiektów w trakcie działania programu.

13.1 Zarządzanie pamięcią z użyciem „surowych" wskaźników

Język C++ definiuje dwie pary operatorów służących do dynamicznego alokowania i zwalniania pamięci odpowiednio:

- **new** i **delete** – dla pojedynczych obiektów, oraz
- **new**`[]` i **delete**`[]` – dla tablic obiektów.

Operatory **new** i **new**`[]` oraz **delete** i **delete**`[]` to (w przybliżeniu) odpowiedniki znanych Ci z języka C funkcji bibliotecznych `malloc()` i `free()`.

Ważne

Miej świadomość, że „ręczne" zarządzanie pamięcią z użyciem wspomnianych operatorów jest bardzo podatne na błędy, co zostało wyczerpująco omówione w rozdziale 13.2 Po co nam inteligentne wskaźniki?.

13.1.1 Dynamiczna alokacja pamięci

Operator **new** wykonuje trzy operacje: przydziela pamięć dla alokowanego obiektu, (opcjonalnie) dokonuje jego inicjalizacji, oraz zwraca wskaźnik na typ zgodny z typem operandu (czyli alokowanego obiektu). Dla porównania funkcja `malloc()` nie umożliwia inicjalizacji alokowanego obiektu oraz zawsze zwraca wartość typu **void***. W obu przypadkach obiekty alokowane dynamicznie nie są nazwane – są obiektami tymczasowymi.

Dynamicznie zaalokowane obiekty są inicjalizowane w sposób domyślny[1], co oznacza że:

- obiekty typów wbudowanych mają niezdefiniowaną wartość, natomiast
- obiekty klas są inicjalizowane z użyciem właściwego konstruktora (w tym potencjalnie domyślnego).

Oto przykład ilustrujący powyższą zasadę:

```
int* p = new int;  // inicjalizacja "w domyślny sposób" = brak inicjalizacji
                   //   (`p` wskazuje na obiekt niezainicjalizowany)

std::string* ps1 = new std::string;       // wywołanie konstruktora domyślnego
                                          //   (obiekt wskazywany będzie mieć
                                          //     pusty łańcuch - "")
std::string* ps2 = new std::string("X"); // wywołanie konstruktora jednoarg.
                                          //   (inicjalizacja łańcuchem "X")
```

przy czym standard C++03 umożliwił inicjalizację obiektów *każdego* typu (także typu prostego) z użyciem nawiasów okrągłych – jak w przypadku konstruktorów klas[2]:

[1] zob. rozdz. 17.3 Inicjalizacja danych

[2] Konstrukcja **new** `T()` została opisana w punkcie ISO C++03 5.3.4[expr.new]/15).

```
int* x = new int();    // `x` wskazuje na obiekt o "wartości domyślnej"
                       //   dla danego typu (czyli tu: 0)
int* y = new int(2);   // `y` wskazuje na obiekt o wartości 2
```

natomiast standard C++11 umożliwił inicjalizację za pomocą tzw. jednolitej inicjalizacji (zapis z użyciem nawiasów klamrowych):

```
// "uniform initialization" => tablica zawiera 10 elementów
//   o wartościach: 1, 2, 3, 0, 0, ...
int* pi = new int[10]{1,2,3};

// "uniform initialization" => kontener zawiera elementy
//   o wartościach 0, 1 i 2
std::vector<int>* pv = new std::vector<int>{0,1,2};
```

Dobre praktyki

Aby uniknąć niezdefiniowanych wartości zawsze *inicjalizuj nowo tworzony obiekt – także w przypadku alokacji dynamicznej.*

Można również dynamicznie alokować obiekty o typie z kwalifikatorem `const`:

```
const int *pci = new const int(1024);
```

13.1.2 Zwalnianie dynamicznie przydzielonej pamięci

Aby zapobiec wyciekom pamięci, musimy zwalniać dynamicznie przydzieloną pamięć, gdy przestaniemy jej potrzebować – w tym celu korzystamy z operatora `delete`, którego operandem jest wskaźnik na obiekt, który należy usunąć i zwolnić odpowiadającą mu pamięć. Wskaźnik przekazany do `delete` musi wskazywać na obiekt zaalokowany dynamicznie albo mieć wartość `nullptr`:

```
delete p;    // `p` musi wskazywać na obiekt zaalokowany
             //   dynamicznie albo mieć wartość nullptr
```

W języku C++ (podobnie jak w języku C) poniższe scenariusze doprowadzą do niezdefiniowanego zachowania programu:

- próba usunięcia wskaźnika na pamięć, która nie została dynamicznie przydzielona
- próba dwukrotnego zwolnienia tego samego miejsca w pamięci

Analogicznie, aby zwolnić pamięć przydzieloną z użyciem operatora `new[]` należy użyć `delete[]`, przykładowo:

```
int* pi = new int[3]{1,2,3};
delete[] pi;
```

13.2 Po co nam inteligentne wskaźniki?

Źródło: A brief introduction to C++'s model for type- and resource-safety (Bjarne Stroustrup et al.)

Rozważ poniższy kod (napisany w języku C) który korzysta ze szczególnych zasobów systemowych, jakim są uchwyty do plików (w języku C uchwytem tym jest wskaźnik na typ strukturalny `FILE`; użyte funkcje `fopen()`, `ftell()`, `fprintf()` i `fclose()` to funkcje z biblioteki standardowej zadeklarowane w pliku `<stdio.h>`):

Listing 13.1. Przykład złych praktyk w zarządzaniu pamięcią.

```
1 // Pewna funkcja napisana przez innego programistę... :>
2 void misuse_file_handle(FILE* p);
3
4 int some_function(int x) {
5   FILE* f = fopen("foo", "w");  // otwarcie pliku = alokacja struktury FILE
```

```
6   if (f == NULL) {
7     // Zwrócenie wskaźnika pustego przez `fopen()` oznacza
8     //   błąd otwarcia pliku.
9     printf("ERROR: Cannot open file.\n");
10  }
11
12  if (0 < x) {
13    misuse_file_handle(f);
14  } else {
15    return EXIT_FAILURE;
16  }
17
18  /* ... */
19  long pos = ftell(f);  // odczyt danych z pliku z użyciem uchwytu `f`
20  /* ... */
21
22  fprintf(f, "Hello world!\n");  // zapis do pliku z użyciem uchwytu `f`
23  fclose(f);  // zamknięcie pliku = (poprawne) zwolnienie struktury FILE
24
25  return EXIT_SUCCESS;
26 }
```

Na pierwszy rzut oka wygląda on niegroźnie, lecz co w przypadku, gdy funkcja `misuse_file_handle()` wygląda tak:

```
void misuse_file_handle(FILE* p) {
  /* ... */
  free(p);  // BŁĄD: ręczne zwolnienie uchwytu do pliku
  /* ... */
}
```

Powyższy kod to przykład *fatalnego* stylu programistycznego, a mimo to niektóre ze znajdujących się w nim błędów programiści popełniają nieprzerwanie od ponad 40 lat! Problemy te wywodzą jeszcze z języka C (stąd przykład zawiera kod w czystym języku C), a ich usunięcie w przypadku dużych projektów nastręcza olbrzymich trudności.

Oto najważniejsze błędy w powyższym przykładzie:

- **Wyciek zasobów**
  Jeśli funkcja zakończy się przedwcześnie, czyli gdy zostanie wywołana instrukcja

  ```
  return EXIT_FAILURE;
  ```

  w linii 15, wówczas plik w ogóle nie zostanie zamknięty – pamięć i zasoby systemowe nie zostaną zwolnione.
- **Dostęp przez niewłaściwy wskaźnik**
  Jeśli funkcja nie zakończy się, instrukcja `free()` wewnątrz funkcji `misuse_file_handle()` sprawi, że wskaźnik do pliku stanie się nieprawidłowy (przy założeniu, że wskaźnik wskazuje na strukturę `FILE` zaalokowaną przy pomocy `malloc()`). To miejsce w pamięci prawdopodobnie zostanie ponownie wykorzystane gdzieś w kodzie schematycznie oznaczonym jako linia 18. Funkcja `ftell()` w linii 19 będzie zatem próbować odczytać dane z użyciem **„wiszącego" wskaźnika** `FILE*` (ang. dangling pointer) – czyli będzie próbować uzyskać dostęp do pamięci, która już nie przechowuje struktury typu `FILE`.
- **Naruszenie pamięci**
  Analogicznie do powyższej sytuacji, funkcja `fprintf()` w linii 22 będzie usiłować nadpisać pamięć, w której trzymane są już inne (niebędące strukturą `FILE`) obiekty.
- **Niepewność odnośnie typu wiązania obiektu**
  Dokumentacja funkcji `fopen()` zawiera informację o tym, czy zwracany przez tę funkcję wskaźnik `FILE*` wskazuje obiekt zaalokowany statycznie, czy dynamicznie. Jednak kompilator nie czyta dokumentacji, a zarówno wywołanie funkcji `free()` na obiekcie zaalokowanym statycznie, jak i brak wywołania `free()` dla obiektu zaalokowanego dynamicznie prowadzi (zwłaszcza w dużych,

złożonych systemach) do poważnych błędów – odpowiednio do naruszenia ochrony pamięci oraz do wycieku pamięci.

- **Niewłaściwe zwalnianie pamięci**
  W dokumentacji biblioteki standardowej znajduje się wzmianka, że w celu zamknięcia pliku otwartego z użyciem funkcji `fopen()` należy użyć funkcji `fclose()` – a nie `free()`. Lecz kompilator nie czyta dokumentacji i nie istnieje mechanizm chroniący nas przed popełnieniem takiego błędu. Nawet gdyby wskaźnik `FILE*` zwracany przez `fopen()` wskazywał na obiekt `FILE` zaalokowany z użyciem `malloc()`, wywołanie `free()` spowoduje jedynie zwolnienie *pamięci* zajmowanej przez obiekt typu `FILE`, ale nie spowoduje zwolnienia pewnych dodatkowych zasobów systemu operacyjnego używanych przez programy do korzystania z plików.

Doświadczenie pokoleń programistów C, C++ (i innych języków) pokazuje, że – przy odpowiednim wysiłku – tych problemów można często uniknąć „ręcznie” (m.in. poprzez wyczerpujące testowanie kodu). Jednak nawet „bycie ostrożnym” i pisanie wyczerpujących testów nie gwarantuje pełnego bezpieczeństwa kodu, a problemy z bezpieczeństwem pamięci, „wiszącymi” wskaźnikami i bezpieczeństwem cyklu życia obiektów stanowią utrapienie i główne źródło błędów w dużych projektach C++. Inteligentne wskaźniki w sposób kompleksowy radzą sobie z powyższymi problemami.

Warto również zaznaczyć, że stosowanie inteligentnych wskaźników pozwala nie tylko uniknąć wspomnianych problemów, lecz także umożliwia korzystającym z nich klasom na automatyczne generowanie składowych odpowiedzialnych za kopiowanie, przypisywanie, oraz usuwanie obiektów takich klas – co dodatkowo upraszcza proces implementacji takich klas.

13.3 Inteligentne wskaźniki

Inteligentne wskaźniki (ang. smart pointers) to abstrakcyjne typy danych, które zachowują się jak tradycyjne (czyli znane z języka C) **„surowe” wskaźniki** (ang. raw pointers), lecz jednocześnie zapewniają dodatkową funkcjonalność – w szczególności automatyczne zarządzanie pamięcią (tj. automatyczne zwalnianie pamięci użytej do przechowywania wskazywanego obiektu) i/lub sprawdzanie zakresu (tj. czy nie próbujemy uzyskać dostępu do nie swojej pamięci). Ta funkcjonalność ma na celu zmniejszenie liczby błędów spowodowanych niepoprawnym użyciem wskaźników, przy jednoczesnym zachowaniu wydajności programu.
W języku C++:

- inteligentne wskaźniki są zdefiniowane jako szablony klasy[3] o parametrze będącym typem wskazywanego obiektu – możesz zatem utworzyć inteligentny wskaźnik do obiektu dowolnego typu;
- inteligentny wskaźnik przechowuje (jako pole szablonu klasy) „surowy” wskaźnik do obiektu wskazywanego;
- funkcjonalność „surowego” wskaźnika jest symulowana za pomocą przeciążenia odpowiednich operatorów (m.in. operatora dereferencji `*`).

W przypadku obiektów alokowanych dynamicznie szczególnego znaczenia nabiera pojęcie własności. W C++ poprzez **własność** (ang. ownership) rozumiemy przede wszystkim to, który fragment kodu jest odpowiedzialny za zwolnienie przydzielonego zasobu (pamięci, uchwytów do plików itp.). Jeśli operacja zwolnienia takiego zasobu nie zostanie poprawnie zaimplementowana, dojdzie do wycieku zasobów – zostaną one bezpowrotnie utracone (a przynajmniej na czas danego uruchomienia programu). Korzystając z „surowych” wskaźników właścicielem zasobu przydzielonej pamięci jest obiekt albo zasięg efektywnie wykonujący operację zwolnienia (w zależności od sposobu alokacji – za pomocą operatora `delete` albo z użyciem funkcji `free()`). Jeśli jednak nie zaimplementujemy poprawnie zwalniania takiej pamięci doprowadzimy do takich problemów jak: wyciek pamięci, powstanie wiszących wskaźników, lub **naruszenie pamięci** (ang. memory corruption). Jak zatem zapewnić poprawność zwalniania pamięci?

Język C++ zapewnia stosowny mechanizm – mechanizm RAII[4] – gwarantujący, że dla statycznie zaalokowanych obiektów *zawsze* (w odpowiednim momencie) zostanie wywołany ich destruktor. Inteligentny wskaźnik stanowi zatem obiektowe opakowanie dla „surowego” wskaźnika – w konstruktorze otrzymuje „surowy” wskaźnik na dynamicznie zaalokowany obiekt, a w destruktorze zwalnia tę pamięć – co dzięki RAII gwarantuje „szczelność” takiego rozwiązania. Inteligentny wskaźnik jest zatem właścicielem wska-

[3] zob. rozdz. 5.3 Szablony klas
[4] zob. rozdz. 9.6 RAII i cykl życia obiektów

zywanego obiektu i wyręcza nas w zwalnianiu pamięci przydzielonej na przechowanie tego obiektu, przy czym sam obiekt inteligentnego wskaźnika może być alokowany statycznie.

Standard C++11 udostępnia trzy rodzaje inteligentnych wskaźników:

- `std::unique_ptr`
  Implementuje posiadanie zasobu „na wyłączność” – w danym momencie tylko jeden wskaźnik tego rodzaju może być właścicielem danego obiektu. Gdy wskaźnik ten zostanie zniszczony, obiekt wskazywany zostaje automatycznie zwolniony. Stosuj domyślnie właśnie ten rodzaj inteligentnych wskaźników.
- `std::shared_ptr`
  Służy do dzielenia się zasobem – dowolna liczba inteligentnych wskaźników tego rodzaju może współdzielić obiekt, natomiast współdzielony obiekt zostanie zniszczony dopiero w chwili, gdy ostatni inteligentny wskaźnik będący (współ)właścicielem zostanie zniszczony.
- `std::weak_ptr`
  Sam nie „posiada” obiektu, natomiast służy do *obserwowania* obiektu zarządzanego przez wskaźniki współdzielone (m.in. posiada metody do określenia, czy wskazywany obiekt wciąż istnieje). Korzystaj z niego, aby uniknąć cyklicznych zależności (np. zniszczenie wskaźnika na listę cykliczną, w której elementy byłyby powiązane z użyciem wskaźników współdzielonych, nie spowoduje zwolnienia pamięci użytej do przechowywania tych elementów)[5].

Aby skorzystać z inteligentnych wskaźników, należy dołączyć standardowy plik nagłówkowy `<memory>`.

Poniżej omówiono zasadę działania i sposób korzystania z najczęściej wykorzystywanych klas inteligentnych wskaźników: `std::unique_ptr` oraz `std::shared_ptr`.

13.3.1 Szablon klasy `std::unique_ptr`

W najprostszym ujęciu szablon klasy `std::unique_ptr` konkretyzowany typem `T` (czyli `std::unique_ptr<T>`) realizuje funkcjonalność inteligentnego wskaźnika na typ `T` „na wyłączność” – otrzymuje „surowy” wskaźnik do dynamicznie zaalokowanego obiektu i nim zarządza (w szczególności automatycznie zwalnia pamięć zajmowaną przez wskazywany obiekt podczas wykonywania swojego destruktora).

W przypadku obiektu typu `std::unique_ptr<T>` nigdy nie ma wątpliwości, kto jest właścicielem wskazywanego obiektu (czyli kto ma później zwolnić zajmowaną przez niego pamięć), gdyż nie można stworzyć kopii obiektu takiego inteligentnego wskaźnika – szablon `std::unique_ptr` został zaprojektowany w taki sposób, że konstruktor kopiujący oraz operator przypisania zostały w nim oznaczone jako usunięte. (Pozostawienie tych operacji stanowiłoby naruszenie koncepcji „unikalnej własności”; utworzenie kopii spowodowałoby, że doszłoby później do próby dwukrotnego zwolnienia tej samej pamięci – raz przez oryginał obiektu inteligentnego wskaźnika, a drugi raz przez jego kopię – powodując błąd ochrony pamięci).

W przypadku klasycznych, „surowych” wskaźników, łatwo o błąd. Rozważ poniższy kod:

```
Foo *p = new Foo("useful object");
make_use(p);
```

Co się stanie ze wskaźnikiem po wywołaniu `make_use()`? Czy `make_use()` stworzy kopię wskaźnika, z której później będzie korzystać? Czy własność wskaźnika zostanie przeniesiona do funkcji `make_use()`, która dokona zwolnienia pamięci przechowującej wskazywany obiekt, czy też funkcja ta pozostawia kwestię zwolnienia `p` w gestii wywołującego? – Nie jesteśmy w stanie odpowiedzieć na te pytania bez dokonania inspekcji kodu tej funkcji lub zapoznania się z jej dokumentacją (co bywa rozwiązaniem zawodnym).

Zastosowanie szablonu `std::unique_ptr` w połączaniu z wprowadzoną w standardzie C++14 funkcją szablonową `std::make_unique` pozwala uniknąć tych problemów, gdyż funkcja ta wewnętrznie dokonuje dynamicznej alokacji i od razu zwraca obiekt inteligentnego wskaźnika, który z kolei nie umożliwia wykonania swojej kopii:

```
// Funkcja `make_unique` została wprowadzona w C++14.
std::unique_ptr<Foo> q = std::make_unique<Foo>(42);
v.push_back(q);  // BŁĄD: metoda `push_back()` jest zaimplementowana tak,
                 //   że wstawia do kontenera KOPIĘ obiektu
```

[5] (dla dociekliwych) zob. Using C++11's Smart Pointers (David Kieras, EECS Department, University of Michigan)

Pamiętaj, że `std::make_unique()` to tzw. **funkcja szablonowa**[6], w związku z czym jej wywołanie ma postać `std::make_unique<T>(args)`, gdzie:

- `T` to typ wskazywanego obiektu, a
- `args` to (opcjonalne) argumenty, które zostaną przekazane do konstruktora obiektu klasy `T`.

Przykładowo:

```
// Utwórz unique-pointer na obiekt typu `Foo` utworzony
//   z użyciem wywołania konstruktora jednoargumentowego: Foo(42)
std::unique_ptr<Foo> q = std::make_unique<Foo>(42);
```

Dla zainteresowanych...

Obiekty typu `std::unique_ptr` powinny być tworzone z użyciem dedykowanej funkcji `std::make_unique()`, głównie w celu zwiększenia bezpieczeństwa kodu. Rozważ poniższy przypadek:

```
fun(std::unique_ptr<A>(new A()), std::unique_ptr<B>(new B()));
```

W powyższej sytuacji, gdy konstruktor jednej z tych klas rzuci wyjątek, podczas gdy obiekt drugiej klasy został już utworzony, ale jeszcze nie objęty przez `std::unique_ptr`, to dojdzie do wycieku pamięci. Skorzystanie z `std::make_unique()` zapobiegnie powstaniu takiego wycieku.

Więcej informacji na temat tego zagadnienia znajdziesz na stronie sequence points oraz Sutter's Mill – GotW #89 Solution: Smart Pointers.

Więcej informacji o szablonie `std::unique_ptr`: C++11 FAQ – `std::unique_ptr`.

`std::unique_ptr` a zmienne lokalne

W poniższym przykładzie:

```
{
  int* buf = new int[256];

  int result = fill_buf(buf);  // BŁĄD: co, jeśli `fill_buf()` rzuci wyjątek?
                               //        Dojdzie do wycieku pamięci!
  if(result == EXIT_FAILURE) {
    return;  // BŁĄD: wycieki pamięci!
  }
  printf("Result: %d", result);

  delete[] buf;
}
```

mamy do czynienia z kilkoma poważnymi problemami związanymi z bezpieczeństwem programu:

- Zapominamy zwolnić `buf` przed wykonaniem instrukcji **`return`** – to klasyczny wyciek pamięci.
- Jeśli funkcja `fill_buf(buf)` rzuci jakikolwiek wyjątek, również nastąpi wyciek pamięci.

Oba powyższe problemy możemy rozwiązać stosując `std::unique_ptr`:

```
void function_b() {
  // Utwórz `unique_ptr` wskazujący na tablicę 256 elementów typu `int`.
  std::unique_ptr<int[]> buf = std::make_unique<int[]>(256);

  int result = fillBuf(buf);
  if (result == EXIT_FAILURE) {
    return;
  }
  printf("Result: %d", result);
}
```

[6] zob. rozdz. 5 Szablony

Ważne

W powyższym scenariuszu użycie `std::unique_ptr` jest zawsze zdecydowanie preferowane nad użyciem „surowego” wskaźnika!

`std::unique_ptr` a pola klasy

Poniższa klasa zawiera sporo „standardowego” kodu:

```
class A {
  public:
    int* i_ptr_;

    A() : i_ptr_(new int(0)) {}

    ~A() {
      delete i_ptr_;
    }

  private:
    // Ponieważ syntezowany konstruktor kopiujący oraz sytnezowany operator
    //   przypisania powieliłby jedynie WARTOŚĆ wskaźnika `i_ptr`, w efekcie
    //   otrzymalibyśmy dwie instancje `A` zawierające pole odnoszące się do
    //   tego samego obszaru w pamięci. Usunięcie pierwszej instancji zwolni
    //   ten obszar, przez co usunięcie drugiej spowoduje próbę ponownego
    //   zwolnienia tego samego obszaru pamięci -- co prowadzi do
    //   niezdefiniowanego zachowania programu.
    // Musimy zatem albo ręcznie zablokować możliwość kopiowania wartości,
    //   albo napisać własnoręcznie stosowne operacje, co nie jest ani
    //   wygodne, ani bezpieczne...
    A(const A&) = delete;
    A& operator=(const A&) = delete;
};
```

Po zastosowaniu użyciu `std::unique_ptr` klasa o analogicznej funkcjonalności będzie miała formę:

```
class B {
  public:
    std::unique_ptr<int> i_ptr_;

    B() : i_ptr_(new int(0)) {}

    // Nie musimy pisać poniższych prototypów, gdyż szablon
    //   `std::unique_ptr` nie umożliwia kopiowania, jednak
    //   możemy do zrobić, aby czytelniej zakomunikować ten
    //   fakt użytkownikom klasy `B`.
    B(const B&) = delete;
    B& operator=(const B&) = delete;

    // Ręczna implementacja destruktora nie jest konieczna, gdyż wywołany
    //   zostanie destruktor pola `i_ptr_` - i to on zwolni pamięć.
};
```

Ponieważ obiekt typu `std::unique_ptr` nie może być kopiowany (z definicji), dla klasy zawierającej pole typu `std::unique_ptr` nie zostaną wygenerowane domyślny konstruktor kopiujący oraz operator przypisania. Jeśli są one w danej klasie potrzebne, musimy je zaimplementować własnoręcznie odpowiednio obsługując pola typu `std::unique_ptr` (np. z użyciem funkcji `std::move()`[7]).

[7] zob. rozdz. 12.3 Zawłaszczanie zasobów

Korzystanie z obiektów wskazywanych przez `std::unique_ptr`

Używając `std::unique_ptr` wciąż możesz udostępniać wskazywany zasób za pomocą metody `get()` zwracającej „surowy wskaźnik” na ten zasób:

```
void use_unique_ptrs_resource(int* up) {
  std::cout << *up << std::endl;
}

// ...

auto up = std::make_unique<int>(1);
use_unique_ptrs_resource(up.get());
```

Ważne

Koncepcja „wyłączności” w przypadku inteligentnych wskaźników dotyczy jedynie *praw własności*, nie samego *dostępu do danych*!

W jakiej sytuacji przekazywać `std::unique_ptr` w jaki sposób?

Dobre praktyki

Zgodnie z konwencją *użycia kontenera* `std::unique_ptr`:
Chcąc przenieść prawa własności do obiektu typu `std::unique_ptr<T>` *przekazuj go do funkcji przez wartość. Chcąc tylko skorzystać ze wskazywanego przez niego obiektu, przekaż argument typu* `T*` *albo* `const T*` *(zgodnie z zasadami „const correctness”).*

Zgodnie ze wspomnianą *konwencją* użycia kontenera `std::unique_ptr` (wynikającą z jednej z głównych dobrych praktyk programowania mówiącej, że „bezpieczeństwo kodu jest ważniejsze niż jego wydajność”) należy go *zawsze* przekazywać przez wartość, gdyż:

- Jeśli chcesz „zawłaszczyć” obiekt typu `std::unique_ptr` przekazanie przez wartość sprawi, że najpierw z użyciem konstruktora przenoszącego `unique_ptr( unique_ptr&& u )` utworzony zostanie obiekt tymczasowy (przy czym konstruktor ten *gwarantuje*, że po jego wykonaniu argument będzie wskazywał na `nullptr` – zatem niezmiennik unikalnej własności zostaje zachowany), a następnie zasób tego tymczasowego obiektu (tj. przechowywany przez niego „surowy” wskaźnik) zostanie zawłaszczony. Kopiowanie obiektu typu `std::unique_ptr` jest wydajne, gdyż w praktyce obiekt ten składa się z pojedynczego pola typu całkowitego – adresu wskazywanego obiektu – zatem narzut wydajnościowy związany z dodatkową alokacją kilku bajtów jest minimalny w porównaniu do korzyści związanych z bezpieczeństwem kodu (to dlatego w kontekście przekazywania argumentów kontener `std::unique_ptr` jest traktowany jak typ prosty).
- Przekazywanie *niestałej* referencji do l-wartości typu `std::unique_ptr` (czyli `std::unique_ptr<T>&`), choć dopuszczalne przez język C++, jest niezgodne z konwencją użycia kontenera `std::unique_ptr` – jest ono równoważne temu, że funkcja *może, ale nie musi* „zawłaszczyć” wskazywany obiekt. Jednak zgodnie z zasadami bezpieczeństwa kodu w języku C++ „jeśli zasób nie ma być modyfikowany, należy przekazać stałą referencję do l-wartości”, z kolei zgodnie ze wspomnianą konwencją do „zawłaszczenia” zasobów obiektu typu `std::unique_ptr` służy przekazywanie przez wartość.
- Przekazywanie *stałej* referencji do l-wartości typu `std::unique_ptr` (czyli `const std::unique_ptr<T>&`), choć dopuszczalne przez język C++, jest niezgodne z tzw. **regułą minimalnej wiedzy** (ang. principle of least knowledge, Law of Demeter, LoD)[8] zgodnie z którą dana „jednostka” w programie (np. funkcja) powinna posiadać jedynie minimalną „wiedzę” o innych elementach programu (np. o ich stanie, udostępnianych operacjach itd.) w celu zwiększenia bezpieczeństwa kodu – zatem w tym przypadku, skoro nie dojdzie do zawłaszczenia zasobów obiektu

[8] zob. Law of Demeter (Wikipedia)

typu `std::unique_ptr`, wystarczy przekazanie „surowego” wskaźnika na obiekt typu `T` (uzyskany za pomocą metody `std::unique_ptr::get()`) zgodnie z zasadami *const correctness*[9].
- Przekazywanie referencji do r-wartości typu `std::unique_ptr` (czyli `std::unique_ptr<T>&&`) jest dopuszczalne, jednak w świetle powyższych zasad równoważny efekt można osiągnąć stosując przekazywanie przez wartość.

13.3.2 Szablon klasy `std::shared_ptr`

Szablon klasy `std::shared_ptr` działa podobnie jak szablon klasy `std::unique_ptr` – przechowuje wskaźnik, udostępnia podstawowy interfejs dla konstruowania i korzystania ze wskaźnika, oraz zapewnia zwolnienie pamięci przy niszczeniu obiektu. Natomiast w przeciwieństwie do `std::unique_ptr`, pozwala również na kopiowanie obiektu `std::shared_ptr` do innego `std::shared_ptr`, a jednocześnie zapewnia, że obiekt wskazywany zostanie zniszczony dokładnie w chwili, gdy zostaje zniszczony ostatni obiekt `std::shared_ptr` będący (współ)właścicielem tego obiektu (bądź gdy wszystkie obiekty zwolnią ten wskaźnik)[10].

Przykład działania:

```
class MyClass {
  public:
    MyClass(int id_);
};

auto ptr = std::make_shared<MyClass>(1);

std::shared_ptr<MyClass> another_ptr = ptr;
// Teraz zarówno `another_ptr` jak i `ptr` wskazują na obiekt o id_=1.

ptr.reset();
// Teraz `ptr` przestaje wskazywać na obiekt o id_=1, lecz obiekt ten
//   nie zostaje zniszczony, gdyż `another_ptr` wciąż się do niego odwołuje.

another_ptr.reset();
// Teraz żaden obiekt typu shared_ptr nie odwołuje się do obiektu
//   o id_=1, więc obiekt ten zostaje zniszczony.
```

Obiekty typu `std::shared_ptr` powinny być tworzone w podobny sposób, co obiekty typu `std::unique_ptr` – przy użyciu standardowej funkcji `std::make_shared()`[11].

13.3.3 Kiedy stosować `std::unique_ptr`, a kiedy `std::shared_ptr`?

Oto zasada, którą najlepiej stosować przy wyborze typu inteligentnego wskaźnika:

Zasada Autora

Gdy nie masz pewności, wybierz `std::unique_ptr` – zawsze będziesz później w stanie zamienić go na `std::shared_ptr`.

Wskaźnik typu `std::shared_ptr` powinien być stosowany wyłącznie *w sytuacjach, gdy rzeczywiście potrzebujemy współdzielić* prawa własności *do zasobów, a nie tylko sam* dostęp *do zasobów. Oto przykład sytuacji, w której zastosowanie `std::shared_ptr` jest uzasadnione: w wielowątkowej aplikacji kilka wątków jednocześnie zapisuje dane do tego samego pliku; plik powinien zostać zamknięty dopiero wtedy, gdy ostatni z wątków skończy swoje działanie.*

Oto argumenty, dlaczego `std::unique_ptr` powinien być *domyślnym* wyborem:

[9] Czyli odpowiednio: `T*` jeśli stan wskazywanego obiektu ma być modyfikowany, a w przeciwnym przypadku `const T*`

[10] W tym celu wykorzystywany jest mechanizm zliczania referencji

[11] To metoda preferowania nie tylko ze względu na zwiększenie bezpieczeństwa kodu, lecz także ze względu na mniejszą liczbę dokonywanych po drodze alokacji pamięci – zob. Using C++11's Smart Pointers – Using `std::shared_ptr`: Getting better memory allocation performance.

- Zawsze należy wybierać najprostsze rozwiązania wystarczające do osiągnięcia celu[12]: wybierz taki typ inteligentnego wskaźnika, który najlepiej wyraża Twoje zamiary, oraz który posiada funkcjonalność wystarczającą Ci *na chwilę obecną*. Jeśli tworzysz nowy obiekt i nie wiesz jeszcze, czy zajdzie potrzeba jego współdzielenia, to znaczy że na chwilę obecną nie ma takiej potrzeby – wybierz `std::unique_ptr`, który zapewnia jasno zdefiniowaną własność (własność na wyłączność).
- Używając `std::unique_ptr` wciąż możesz udostępniać wskazywany zasób – za pomocą metody `get()`; „wyłączność" dotyczy jedynie praw własności, nie zaś samego dostępu do danych[13].
- Wskaźnik typu `std::unique_ptr` jest wydajniejszy oraz zużywa mniej pamięci niż wskaźnik typu `std::shared_ptr` – `std::unique_ptr` nie potrzebuje mechanizmu zliczania referencji, zatem zapewnia niemal identyczną wydajność, co „surowy" wskaźnik.
- Użycie wskaźnika typu `std::unique_ptr` pozostawia Ci otwartą możliwość konwersji obiektu takiego typu na typ `std::shared_ptr`, z użyciem funkcji `std::move()`[14]. Wykonanie odwrotnej operacji nie jest możliwe.

13.3.4 Operacje przenoszące szablonu klasy `std::unique_ptr`

Zarówno konstruktor przenoszący, jak i przenoszący operator przypisania *gwarantują*, że po ich wykonaniu argument będzie wskazywał na `nullptr` – zatem niezmiennik unikalnej własności zostaje zachowany:

```
auto u1 = std::make_unique<int>(1);
auto u2 = std::move(u1);
// w tym miejscu zachodzi: u1 == nullptr
```

Operacja przenoszenia praw własności przydaje się zwłaszcza podczas przekazywania obiektów typu `std::unique_ptr` do konstruktora klasy zawierającej wskaźnik typu `std::unique_ptr`:

```
class Animal {
  public:
    virtual void make_sound() const = 0;
    virtual ~Animal() = default;
};

class Dog : public Animal {
  public:
    void make_sound() const override { std::cout << "Woof!\n"; }
};

// "Dom" może posiadać "zwierzę" (dowolnego gatunku) -- aby móc skorzystać
//   z polimorfizmu pole `pet_` musi być typu wskaźnikowego (w tym przypadku
//   referencja nie jest dopuszczalna, gdyż obiekt klasy `House` ma być
//   "właścicielem" zwierzęcia - dla referencji obiekt zwierzęcia istniałby
//   poza obiektem klasy `House`, a "dom" posiadałby jedynie uchwyt do danej
//   instancji "zwierzęcia").
class House {
private:
  std::unique_ptr<Animal> pet_;
public:
  // Przeniesienie praw własności z argumentu do pola pet_;
  House(std::unique_ptr<Animal> pet) : pet_(std::move(pet)) {}
};

// ...

House(std::make_unique<Dog>());
```

[12] zob. zasada KISS

[13] zob. rozdz. 13.3.1 Korzystanie z obiektów wskazywanych przez `std::unique_ptr`

[14] zob. rozdz. 12.3 Zawłaszczanie zasobów

13.3.5 Istotne ograniczenia w stosowaniu inteligentnych wskaźników

Choć mechanizm inteligentnych wskaźników został gruntownie przemyślany i zaprojektowany dla możliwie dużego bezpieczeństwa kodu, inteligentne wskaźniki nie są typami wbudowanymi w język C++ – są to zwykłe klasy, które podlegają standardowym regułom języka C++. Oznacza to, że jeśli nie zastosujemy się do pewnych zasad (których kompilator nie może wymusić), otrzymamy niezdefiniowane wyniki. W skrócie, zasady te to:

- Korzystaj z inteligentnych wskaźników tylko do wskazywania na obiekty utworzone dynamicznie.
- Musisz zapewnić, aby każdy zarządzany obiekt posiadał dokładnie jeden obiekt zarządzający. Możesz tego dokonać poprzez pisanie kodu w taki sposób, że nowo utworzony obiekt zostaje momentalnie przekazany do inteligentnego wskaźnika (a pozostałe inteligentne wskaźniki odwołują się poprzez ten pierwszy wskaźnik) – najlepiej z użyciem odpowiednio `std::make_unique()` albo `std::make_shared()`[15].
- Jeśli chcesz w pełni wykorzystać możliwości inteligentnych wskaźników, stosuj „surowe" wskaźniki w odniesieniu do tego samego obiektu wskazywanego przez inteligentny wskaźnik wyłącznie w kontekście, gdzie *nie* stają się one właścicielem obiektu. Przykładem takiej sytuacji jest przekazywanie argumentu do funkcji – wtedy należy zagwarantować, że wskazywany obiekt będzie „żył" przez cały czas wywołania funkcji (zob. rozdz. 13.3.1 Korzystanie z obiektów wskazywanych przez `std::unique_ptr`).

Ważne

Jeśli nie będziesz stosować powyższych zasad, stwarzasz ryzyko powstania „wiszących" wskaźników lub podwójnego zwalniania tej samej pamięci.

13.4 Zarządzanie pamięcią w C++98 a C++14

Przykłady 13.2 i 13.3 pokazują różnice w sposobie zarządzania pamięcią między standardami C++98 i C++14. W obu poniższych przykładach klasa `MyClass` jest zdefiniowana następująco:

```
class MyClass {
private:
    int id_;
public:
    MyClass(int _id) : id_(_id) {};
    int getId() const { return id_; }
    bool operator==(const MyClass& cls) { return id_ == cls.id_; }
};
```

Listing 13.2. Zarządzanie pamięcią w C++98.

```
#include <cstdlib>
#include <vector>
#include <iostream>

int main() {

  // Ręczna alokacja z użyciem `new`.
  std::vector<MyClass*> vw;
  vw.push_back(new MyClass(1));
  vw.push_back(new MyClass(2));

  MyClass* p = vw[1];

  // Iterowanie po kontenerze z użyciem iteratora.
  for (std::vector<MyClass*>::iterator i = vw.begin(); i != vw.end(); i++) {
    // Sprawdzanie poprawności wskaźnika z użyciem makra `NULL`.
    if(*i != NULL && **i == *p) {
```

[15]Rozwiązanie wykorzystujące wyrażenie z **new** jako argument konstruktora inteligentnego wskaźnika jest niezalecane ze względów bezpieczeństwa – zob. rozdz. 13.3.1 Szablon klasy `std::unique_ptr`

```
      std::cout << "Object #" << (*i)->getId() << " is a match\n";
    }
  };

  // Ręczne zwalnianie pamięci.
  for(std::vector<MyClass*>::iterator i = vw.begin(); i != vw.end(); i++) {
    delete *i;
  }

  return EXIT_SUCCESS;
}
```

Listing 13.3. Zarządzanie pamięcią w C++14.

```
#include <cstdlib>
#include <vector>
#include <iostream>

#include <memory>

int main() {

  // Automatyczna alokacja z użyciem `make_unique`.
  std::vector<std::unique_ptr<MyClass>> vw;
  vw.push_back(std::make_unique<MyClass>(1));
  vw.push_back(std::make_unique<MyClass>(2));

  // Automatyczna dedukcja typu z użyciem `auto`.
  const auto& p = vw[1];

  // Iterowanie po kontenerze z użyciem range-for.
  for (const auto& s : vw) {
    // Sprawdzanie poprawności wskaźnika z użyciem literału `nullptr`.
    if(s != nullptr && *s == *p) {
      std::cout << "Object #" << s->getId() << " is a match\n";
    }
  };

  // Automatyczne zwalnianie pamięci (zajmuje się tym destruktor szablonu
  //   klasy `std::unique_ptr`).

  return EXIT_SUCCESS;
}
```

13.5 `std::unique_ptr`: jeszcze jeden przykład...

Oto jeszcze jeden, bardziej rozbudowany przykład wykorzystania typu `std::unique_ptr` – do realizacji polimorfizmu. W przykładzie tym chcemy zaimplementować grę w labirynt pokoi (klasa `MazeGame`), przy czym jeden z wariantów gry (klasa `MagicMazeGame`) oprócz istnienia pokojów „zwykłych” (klasa `Room`) dopuszcza też istnienie pokojów „magicznych” (klasa `MagicRoom`), które zachowują się nieco inaczej niż „zwykłe” pokoje:

Listing 13.4. `std::unique_ptr` a polimorfizm

```
#include <cstdlib>
#include <iostream>
#include <vector>
#include <list>
#include <memory>

class Room {
private:
```

```
    // Dany 'pokój' nie posiada innego 'pokoju' na wyłączność, a jedynie
    //   przechowuje uchwyt do niego w celu korzystania z jego składowych.
    std::vector<Room*> connected_rooms_;

public:
    // Przekazywanie przez "surowy" wskaźnik sugeruje, że prawa własności nie
    //   są przenoszone.
    void connect(Room* room) {
        connected_rooms_.push_back(room);
    }
};

class MagicRoom : public Room {
private:
    int x_;
public:
    MagicRoom(int x) : Room(), x_(x) {
        std::cout << "Differs from the base class..." << std::endl;
    }
};


class MazeGame {
private:
    // 'Gra' jest wyłącznym właścicielem 'pokojów' -- jest odpowiedzialna za
    //   zwolnienie obiektów w odpowiednim momencie.
    std::list<std::unique_ptr<Room>> rooms_;

public:
    virtual std::unique_ptr<Room> make_default_room() {
        // Utworzenie inteligentnego wskaźnika typu `std::unique_ptr<Room>`
        // z użyciem dedykowanej funkcji `std::make_unique<Td>(ctor_args)`
        // -- funkcja dynamicznie alokuje obiekt typu `Td` utworzony za
        // pomocą następującego wywołania konstruktora: `Td(ctor_args)`.
        return std::make_unique<Room>();  // obiekt utworzony z użyciem
                                          //   wywołania `Room()`
    }

    void an_operation() {
        // Wywołania przenoszących operatorów przypisania.
        auto room1 = make_default_room();
        auto room2 = make_default_room();

        // Przekazanie "surowego" wskaźnika za pomocą metody
        //   `std::unique_ptr::get()`.
        room1->connect(room2.get());

        // "Zawłaszczenie" obiektów przez kontener z użyciem `std::move()`
        //   i `std::vector::push_back(T&&)` -- w celu poprawy wydajności.
        //   (wywołanie `rooms_.push_back(room)` mogłoby spowodować
        //   utworzenie niepotrzebnej kopii.
        rooms_.push_back(std::move(room1));
        rooms_.push_back(std::move(room2));
    }
};

class MagicMazeGame : public MazeGame {
public:
    MagicMazeGame() : MazeGame() {
        std::cout << "Differs from the base class..." << std::endl;
    }
```

```
    std::unique_ptr<Room> make_default_room() override {
        // Utworzenie inteligentnego wskaźnika typu `std::unique_ptr<Room>`
        // z użyciem dedykowanej funkcji `std::make_unique<Td>(ctor_args)`
        // -- funkcja dynamicznie alokuje obiekt typu `Td` utworzony za
        // pomocą następującego wywołania konstruktora: `Td(ctor_args)`.
        return std::make_unique<MagicRoom>(1); // obiekt utworzony z użyciem
        //   wywołania `MagicRoom(1)`
    }
};

int main() {
    MagicMazeGame mmg;
    mmg.an_operation();

    return EXIT_SUCCESS;
}
```

Materiały źródłowe

- Wskaźniki inteligentne (MSDN)
- Using C++11's Smart Pointers (David Kieras)
- Working with Dynamic Memory in C++ (C++ Primer, 5th Ed.)
- A brief introduction to C++'s model for type- and resource-safety (Bjarne Stroustrup et al.)
- Smart Pointers (Effective Modern C++ by Scott Meyers)

Rozdział 14

Przekazywanie wartości

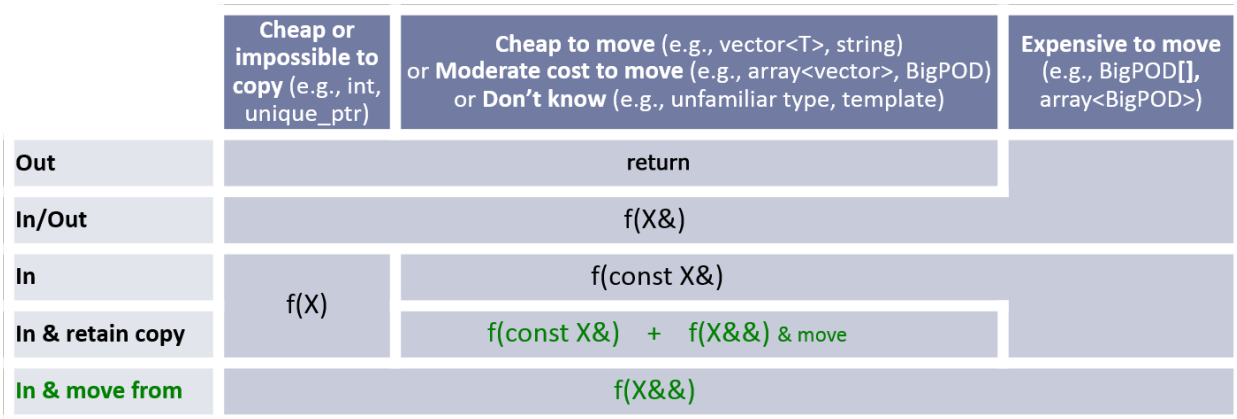
Znasz już kilka semantyk – wartości, wskaźników, referencji i przeniesienia – oraz zasady „const correctness". Nadeszła zatem pora, aby usystematyzować zasady przekazywania wartości do funkcji i z funkcji.

Ważne

Poniżej przedstawione zasady zostały opracowane na podstawie wytycznych ze strony C++ Core Guidelines, którą współtworzą Bjarne Stroustrup i Herb Sutter (odpowiednio twórca języka C++ oraz jeden z najbardziej wpływowych ekspertów od tego języka). Na potrzeby niniejszego kursu w skrypcie pominięto szereg „przypadków szczególnych", kiedy istnieją wyspecjalizowane techniki na osiągnięcie pożądanego rezultatu[1] – w związku z tym podane w nim zasady należy traktować jako „zasady kciuka".

Podstawowe zasady przedstawia Rys. 14.1, przy czym została na nim zastosowana poniższa terminologia w odniesieniu do przekazywania wartości do i z funkcji:

- wartość „*out*" – oznacza, że ważna jest tylko wartość przekazywana z funkcji
- wartość „*in/out*" – oznacza, że ważna jest zarówno wartość przekazywana do i z funkcji
- wartość „*in*" – oznacza, że ważna jest tylko wartość przekazywana do funkcji
- wartość „*in & retain copy*" – oznacza taką wartość „in", która zostanie skopiowana
- wartość „*in & move from*" – oznacza taką wartość „in", która zostanie zawłaszczona



Rysunek 14.1. Sposoby przekazywania wartości do funkcji i z funkcji. (Źródło: C++ Core Guidelines (isocpp.org))

Oto kilka uwag do zasad z rysunku 14.1:

- **Parametry „do przeniesienia" przekazuj przez `X&&` (a następnie je przenoś).** [F.18]
  - **Wyjątek:** Wybrane typy o unikalnej własności (np. `std::unique_ptr`) przekazuj przez wartość.
    W przypadku tych typów z unikalnym właścicielem – czyli uniemożliwiających kopiowanie, a jedynie umożliwiających przenoszenie – które są „tanie" w przenoszeniu (np. `std::unique_ptr`) mogą być przekazywane przez wartość – jest to najprostsze rozwiązanie, a osiąga ten sam efekt

[1] Przykładowe pominięte zagadnienia to *perfect forwarding* oraz optymalizacja dla dużych obiektów kosztownych w przenoszeniu.

minimalnym narzutem wydajnościowym[2]. Co do zasady należy preferować prostotę i czytelność (i bezpieczeństwo kodu) ponad wydajność.

- **Chcąc zwrócić wartość „*out*" preferuj użycie `return` ponad zwracanie jej przez referencję za pomocą parametrów** [F.20]
  - Zwróć uwagę, że „wartość zwracana" to nie to samo, co „zwracanie przez wartość" – np. w przypadku metod klas będących „getterami" możesz spokojnie zwracać typ `const T&`.
  - Wartość zwracana z użyciem `return` dokumentuje się sama jako „wyłącznie wyjściowa", podczas gdy użycie parametru referencyjnego `X&` jest niejednoznaczne (może oznaczać albo wartość „*in/out*", albo wartość „*out*") oraz jest podatne na niewłaściwe użycie danych. Duże obiekty (takie jak kontenery standardowe) są zoptymalizowane pod kątem niejawnych operacji przeniesienia m.in. w celu unikania zbędnego zarządzania pamięcią – zatem je również można zwracać z użyciem `return` bez straty dla wydajności.
  - Dla tzw. typów *non-value*[3], takich jak typy w hierarchii dziedziczenia, zwracaj obiekt z użyciem inteligentnych wskaźników (domyślnie: `std::unique_ptr`).
- **Chcąc zwrócić kilka wartości „out" preferuj zwracanie wartości struktury lub krotki.** [F.21]
  Wartość zwracana z użyciem `return` dokumentuje się sama jako „wyłącznie wyjściowa". Jeśli zwracane wartości są ze sobą powiązane znaczeniowo (np. reprezentują pewną abstrakcję danych), zastosuj nazwany typ strukturalny. W przeciwnym razie, zastosuj anonimową krotkę (czyli np. kontener `std::tuple`) w połączeniu z mechanizmem rozpakowania takiej krotki (tj. funkcją `std::tie()`) w miejscu wywołania rozpatrywanej funkcji[4].
- **Preferuj `X*` ponad `X&`, jeśli „brak argumentu" jest poprawną możliwością.** [F.60]
  Wskaźnik (`X*`) posiada specjalny „niepoprawny" stan – wskaźnik pusty (o literale `nullptr`), podczas gdy referencja (`X&`) takiego stanu nie posiada (nie istnieje coś takiego, jak „pusta referencja"): jeśli w Twoim problemie występuje taki „niepoprawny" stan – zastosuj wskaźnik, w przeciwnym razie – zastosuj referencję (ze względu na prostszy zapis i potencjalną możliwość zastosowania lepszych optymalizacji kodu).
- **Zgodnie z konwencjami funkcje „zawłaszczające" argument powinny robić to zawsze, a nie w zależności od wybranej ścieżki sterowania.**
  Wynika to z faktu, że w przypadku niektórych typów danych użycie operacji przeniesienia zmienia stan obiektu z którego zasoby zostały przeniesione, np. kontener `std::vector` staje się pusty – programista zwykle chce oczekiwać takiego jego stanu po powrocie z funkcji.

Dla zainteresowanych...

Wprowadzenie semantyki przeniesienia i związanych z nią mechanizmów (np. copy elision) sprawiają, że w niektórych przypadkach wybór „najbardziej wydajnego" sposobu przekazywania wartości może być bardzo niejednoznaczny – przykładem może być choćby przekazywanie obiektów typu `std::string`, o czym możesz przeczytać w tej dyskusji. Jednocześnie zwróć uwagę, że sytuację może zmienić też wprowadzenie nowych funkcjonalności do biblioteki standardowej – co miało miejsce w przypadku wspomnianego typu `std::string`, o czym przeczytasz tu .

[2] Przekazywanie przez wartość generuje jedną dodatkową (tanią) operację przenoszenia.
[3] Czyli takich, których zachowanie nie jest określone tylko i wyłącznie przez stan obiektu.
[4] zob. rozdz. 7.3.3 Kontener `std::tuple`

Rozdział 15

Wyjątki i ich obsługa

Wyjątek (ang. exception) to anomalia występująca podczas działania programu – np. utrata połączenia z bazą danych lub napotkanie na niepoprawnie sformatowane dane wejściowe – która uniemożliwia dalsze poprawne jego wykonywanie, a zarazem nie może być obsłużona w chwili jej wystąpienia. Dobrze napisany program powinien zapewniać rozsądną obsługę takich sytuacji. Przykładowo, gdy problemem są nieprawidłowe dane podane przez użytkownika, program może poprosić go o ich ponowne podanie.

Mechanizm obsługi wyjątków pozwala oddzielić kod odpowiedzialny za wykrywanie i zgłaszanie sytuacji wyjątkowych od kodu odpowiedzialnego za obsługę takich sytuacji – oba wspomniane fragmenty nie potrzebują o sobie nawzajem żadnej informacji, a jedynym sposobem komunikacji jest tzw. *obiekt wyjątku*. W szczególności fragment programu, który po napotkaniu na sytuację wyjątkową zgłasza ją, nie musi wiedzieć który inny fragment programu będzie odpowiedzialny za jej obsłużenie (ani czy w ogóle dany wyjątek zostanie obsłużony w programie) – po prostu fragment programu, który wykrył sytuację wyjątkową, po zgłoszeniu wyjątku kończy swoje działanie.

Informacja

Programiści tworzący programy, które dokonują obsługi wyjątków w celu kontynuowania działania, muszą zachowywać dużą czujność – muszą pisać kod w taki sposób, aby zapewnić poprawność tworzonych obiektów, zapobiec wyciekowi pamięci, oraz zapewnić zachowanie poprawnego stanu programu po obsłużeniu wyjątku.

Należy mieć świadomość, że pisanie kodu w pełni *bezpiecznego pod względem obsługi wyjątków to zadanie niezmiernie wymagające (i wykraczające daleko poza ramy tego podręcznika). Mimo to znajomość podstawowych zagadnień związanych z wyjątkami jest niezmiernie przydatna nawet w pracy z programami o niewielkiej złożoności.*

15.1 Elementy języka C++ służące do obsługi wyjątków

Na obsługę wyjątków w języku C++ składają się następujące elementy:

- wyrażenie `throw` – służy do zgłaszania wyjątku
- konstrukcja `try-catch` – służy do wykrywania i obsługi wyjątków; wyjątki zgłoszone w trakcie wykonywania fragmentu kodu objętego blokiem `try` są zwykle obsługiwane przez jedną z klauzul `catch` (ang. `catch` clause)
- klasy wyjątków – klasy te pozwalają przekazywać informację o okolicznościach wystąpienia wyjątku pomiędzy miejscem jego zgłoszenia, a miejscem jego obsługi

Elementy te zostaną szerzej omówione w kolejnych podrozdziałach.

15.1.1 Wyrażenie `throw`

Wyrażenie `throw` służy do **zgłoszenia wyjątku** (ang. exception raising), przy czym typ wyrażenia umieszczonego po słowie kluczowym `throw` określa typ rzuconego wyjątku. Zwykle po wyrażeniu `throw` występuje średnik, przez co staje się ono instrukcją.

Przykład zgłaszania wyjątku:

```
int foo() {
  // Rzucony wyjątek ma typ `std::runtime_error`
```

```
    throw std::runtime_error("some error...");
}
```

15.1.2 Konstrukcja `try-catch`

Ogólna postać konstrukcji **`try-catch`** wygląda następująco:

```
try {
  program-statements
} catch (exception-declaration-1) {
  handler-statements-1
} catch (exception-declaration-2) {
  handler-statements-2
} // ...
```

Blok **`try`** zawiera instrukcje stanowiące fragment normalnej logiki programu. Po bloku **`try`** występuje co najmniej jedna klauzula **`catch`** służąca do obsługi wyjątków, które mogły zostać zgłoszone podczas wykonywania ciągu instrukcji z bloku **`try`**. Po wykonaniu danej klauzuli **`catch`** program jest wykonywany dalej, począwszy od instrukcji znajdującej się po końcu całej konstrukcji **`try-catch`**.

Klauzula `catch`

Klauzula **`catch`** składa się z następujących elementów: słowa kluczowego **`catch`**, nawiasów okrągłych zawierających deklarację obiektu-wyjątku (dopuszczalne jest pominięcie nazwy), oraz samego bloku zawierającego instrukcje procedury obsługi wyjątku (zob. przykład 15.1).

Deklaracja wyjątku w klauzuli **`catch`** przypomina listę parametrów funkcji z dokładnie jednym parametrem – zostanie on zainicjalizowany obiektem rzuconym przez wyrażenie **`throw`**. Podobnie jak w przypadku „funkcyjnej" listy parametrów, możemy pominąć nazwę parametru klauzuli **`catch`**, jeśli nie potrzebujemy dostępu do obiektu rzuconego przez **`throw`**. Typ deklaracji określa jakiego rodzaju wyjątki dana klauzula może obsłużyć. Klauzula obsługująca wyjątek typu powiązanego relacją dziedziczenia powinna definiować swój parametr jako referencję (aby nie doszło do „przycięcia"[1]).

Podobnie jak w przypadku każdego innego bloku, zmienne zadeklarowane wewnątrz bloku **`try`** nie są widoczne wewnątrz klauzul **`catch`**.

Listing 15.1. Przykład klauzuli **`catch`** (z anonimowym parametrem).

```
try {
  foo();
} catch (std::runtime_error&) {
  // Klauzula akceptująca wyjątki klasy `std::runtime_error`
  // (oraz typy potomne dla `std::runtime_error`).
}
```

15.2 Rzucanie i wychwytywanie wyjątku

W chwili rzucenia wyjątku za pomocą wyrażenia **`throw`** normalne wykonywanie programu zostaje przerwane – rozpoczyna się proces poszukiwania klauzuli **`catch`** pozwalającej na obsługę wyjątku rzuconego typu.

Ponieważ instrukcje następujące po instrukcji **`throw`** nie są wykonywane, z punktu widzenia kontroli sterowania programu **`throw`** działa podobnie jak instrukcja **`return`**. Należy jednak zaznaczyć, że instrukcja **`throw`** powoduje również przedwczesne zakończenie wszystkich funkcji leżących na **stosie wywołań**[2] (ang. function call stack) głębiej w stosunku do funkcji, w której rzucony wyjątek zostanie obsłużony – proces ten nosi nazwę *odwijania stosu*.

[1] zob. rozdz. 8.2.1 Zasada „nie ma niejawnej konwersji między obiektami"

[2] zob. stos wywołań funkcji

15.2.1 Odwijanie stosu

W mechanizmie wyjątków C++ kontrola sterowania zostaje przekazana z instrukcji `throw` do pierwszej instrukcji `catch`, jaka może obsłużyć typ rzuconego wyjątku. Gdy osiągnięta zostanie instrukcja `catch`, wszystkie zmienne automatyczne, które znajdują się w zakresie między instrukcjami `throw` i `catch` są niszczone w procesie, który jest określany jako **odwijanie stosu** (ang. stack unwinding).
Przebieg programu posiadającego konstrukcję `try-catch` wygląda następująco:

1. Kontrola sterowania programu osiąga blok `try`, czyli tzw. **sekcję strzeżoną kodu** (ang. guarded code section), poprzez normalne wykonywanie programu. Rozpoczyna się sekwencyjne wykonywane instrukcji w sekcji strzeżonej.
2. Jeśli podczas wykonywania sekcji strzeżonej nie zostanie zgłoszony żaden wyjątek, następujące po bloku `try` klauzule `catch` nie zostają wykonane – wykonywanie jest kontynuowane od instrukcji znajdującej się po konstrukcji `try-catch`.
3. Jeśli podczas wykonywania którejkolwiek z instrukcji w sekcji strzeżonej zostanie zgłoszony nieobsłużony wyjątek, program szuka takiej klauzuli `catch`, która może obsłużyć wyjątek zgłoszonego typu. Klauzule obsługi `catch` są rozpatrywane w kolejności ich występowania po bloku `try`. Jeśli nie znaleziono odpowiedniej klauzuli obsługi, rozpatrywany jest następny dynamicznie otaczający blok `try` – zewnętrzny blok `try` w przypadku zagnieżdżonych bloków `try`, albo blok `try` obejmujący instrukcję, która spowodowała wywołanie aktualnie wykonywanej funkcji. Proces ten jest kontynuowany aż do zweryfikowania najbardziej oddalonego otoczenia bloku `try`.
   (a) Jeśli zostanie znaleziona odpowiadająca klauzula obsługi `catch`, jej parametr jest inicjowany obiektem wyjątku i rozpoczyna się proces odwijania stosu. Wiąże się to ze zniszczeniem wszystkich obiektów automatycznych, które zostały w pełni skonstruowane (ale nie zostały jeszcze zniszczone) między początkiem bloku `try` skojarzonego z klauzulą obsługi `catch`, a instrukcją `throw` zgłaszającą wyjątek[3]. Niszczenie obiektów następuje zgodnie z zasadami RAII[4], w odwrotnej kolejności do konstrukcji – w przypadku obiektów typu klasowego wywołany zostanie stosowny destruktor. Klauzula obsługi `catch` jest wykonywana, a program wznawia wykonanie po ostatniej klauzuli obsługi – to znaczy, w pierwszej instrukcji lub konstrukcji, która nie jest klauzulą obsługi `catch`.
   (b) Jeśli nadal nie znaleziono pasującej klauzuli obsługi lub jeśli podczas procesu odwijania wystąpi wyjątek (ale zanim klauzula obsługi przejmie kontrolę), wywoływana jest funkcja `std::terminate()`.

Przykład przebiegu odwijania stosu: MSDN

Ważne

Ponieważ główna funkcja programu nie różni się w zachowaniu wiele od każdej innej funkcji zdefiniowanej w programie – wyjątek nieobsłużony nigdzie w programie spowoduje zakończenie całego programu!

15.2.2 Znajdowanie pasującej klauzuli obsługi

W trakcie poszukiwań pasującej klauzuli obsługi wybierana jest nie tyle klauzula najlepiej pasująca do typu wyjątku, co *pierwsza* pasująca klauzula. W związku z tym najbardziej wyspecjalizowana klauzula powinna pojawiać się jako pierwsza w kolejności na liście wszystkich klauzul `catch` – w przypadku hierarchii dziedziczenia dla wyjątków klauzule dla typów potomnych powinny pojawiać się przed klauzulami dla typów macierzystych.

Czasem możemy chcieć obsłużyć w identyczny sposób wszystkie potencjalnie występujące wyjątki, bez względu na ich typ. W tym celu używamy klauzuli `catch (...)`, w której zamiast typu podajemy wielokropek. Taka klauzula powinna znajdować się na samym końcu listy klauzul.

[3] Gdy wyjątek wystąpi w konstruktorze, wszystkie dotychczas zainicjalizowane składowe również zostaną poprawnie usunięte. Podobna zasada odnosi się błędów podczas inicjalizacji kolekcji elementów – wszystkie elementy zainicjalizowane przed wystąpieniem wyjątku zostaną poprawnie usunięte.

[4] zob. rozdz. 9.6 RAII i cykl życia obiektów

15.2.3 Wyjątki a destruktory

Dzięki mechanizmowi odwijania stosu mamy gwarancję, że dla każdego zainicjalizowanego obiektu zostanie wywołany jego destruktor – nawet w sytuacji wystąpienia wyjątku. W związku z tym warto stosować klasy w celu kontroli alokacji zasobów, gdyż mamy wtedy gwarancję ich zwolnienia – o ile tylko odpowiednio zaimplementujemy destruktory takich klas.

Należy jednak pamiętać, że podczas odwijania stosu mamy do czynienia z sytuacją, gdy wyjątek został już zgłoszony, lecz jeszcze nie został obsłużony. Zgłoszenie kolejnego wyjątku, który nie zostanie obsłużony w miejscu jego zgłoszenia (tj. wewnątrz tej samej funkcji), spowoduje zakończenie działania programu – program w danym momencie może bowiem przetwarzać tylko jeden wyjątek (nie istnieje „bufor wyjątków"). W związku z tym destruktory powinny umieszczać kod, który potencjalnie może spowodować zgłoszenie wyjątku, wewnątrz sekcji strzeżonej.

15.3 Klasy wyjątków

Ponieważ klauzule obsługi wybierane są na podstawie zgodności zadeklarowanego w nich typu wyjątku z typem rzuconego wyjątku, zagadnienie umiejętnego konstruowania hierarchii klas wyjątków jest bardzo istotne.

15.3.1 Standardowe klasy wyjątków

Biblioteka C++ definiuje kilka klas wyjątków, które służą do zgłaszania problemów napotykanych podczas wykonywania funkcji z biblioteki standardowej – ale z których i my możemy swobodnie korzystać. W przypadku większości z nich tworząc obiekt wyjątku możemy przekazać do konstruktora komunikat o okolicznościach wystąpienia wyjątku (jako łańcuch znaków w stylu C lub obiekt klasy `std::string`), co pozwala na lepszą diagnostykę problemu w przypadku rzucenia wyjątku – informację tę możemy później uzyskać za pomocą metody `what()`. Przykładowo, wykonanie poniższego programu:

```
#include <cstdlib>
#include <iostream>
#include <exception>

int main(void) {
    try {
      // Rzuć wyjątek typu `std::logic_error` posiadający powiązaną wiadomość.
      throw std::logic_error("Custom error message!");
    } catch (std::logic_error& ex) {
      // Wypisz łańcuch znaków powiązany z obiektem wyjątku.
        std::cout << ex.what() << std::endl;
    }

    return EXIT_SUCCESS;
}
```

spowoduje wyświetlenie na konsoli komunikatu:

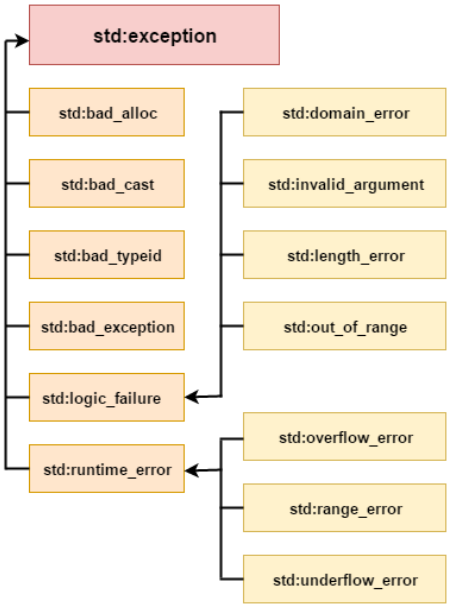
```
Custom error message!
```

Hierarchia standardowych klas wyjątków została zilustrowana na Rys. 15.1. Dwie najistotniejsze z punktu widzenia programistów standardowe (ogólne) klasy wyjątków to:

- `std::logic_error` – obiekty tej klas są rzucane jako wyjątki, aby zasygnalizować błędy prawdopodobnie do wychwycenia przed uruchomieniem programu (np. związane z naruszeniem warunków wstępnych – przykładowo błędnie sformatowane dane wejściowe)
- `std::runtime_error` – obiekty tej klas są rzucane jako wyjątki, aby zasygnalizować błędy możliwe do wychwycenia dopiero w trakcie działania programu (np. brak możliwości nawiązania połączenia z serwerem)

15.3.2 Korzystanie z własnych typów wyjątków

Aplikacje często rozszerzają standardową hierarchię wyjątków o własne klasy wyjątków, które reprezentują sytuacje wyjątkowe właściwe dla problemu modelowanego w ramach aplikacji. Przykładem takiej



Rysunek 15.1. Hierarchia standardowych klas wyjątków. (Źródło: javatpoint)

sytuacji może być aplikacja realizująca obliczenia numeryczne, w przypadku których sytuacją wyjątkową będzie choćby dzielenie przez 0.

Definiując własne klasy wyjątków należy *wywodzić* je ze standardowych klasach wyjątków[5] – w praktyce z klasy `std::logic_error` albo `std::runtime_error`. Dzięki temu nie tylko możemy skorzystać z funkcjonalności wspomnianych standardowych klas, lecz również ułatwiamy użytkownikom kodu obsługę naszych wyjątków – nawet jeśli użytkownik nie zamieści kodu obsługi wyjątków zdefiniowanego przez nas typu, wyjątki takie wciąż mogą być wychwycone podczas obsługi klasy `std::exception` (każdorazowe zapewnienie obsługi klasy `std::exception` w przypadku umieszczania w kodzie bloku **`try-catch`** należy do dobrej praktyki).

Przykład 15.2 pokazuje poprawny sposób definiowania i korzystania z własnych typów wyjątków – na przykładzie błędu dzielenia przez 0. Błąd ten występuje w sytuacji, gdy wartość danych wejściowych (tu: dzielnika) znajduje się poza dopuszczalnym zakresem – w związku z tym właściwą klasą macierzystą dla naszej klasy wyjątku jest `std::domain_error`. W przykładzie tym wywołujemy konstruktor klasy macierzystej `std::domain_error` z łańcuchem znaków, który umożliwia przekazanie opisu sytuacji powodującej zgłoszenie wyjątku (to standardowa funkcjonalność wszystkich klas dziedziczących po klasie `std::exception`).

Listing 15.2. Definiowanie własnej klasy wyjątków.

```
#include <cstdlib>
#include <stdexcept>
#include <iostream>

class DivideByZeroException : public std::domain_error {
public:
  DivideByZeroException(): std::domain_error("Attempted"
      " to divide by zero") {}
};

double quotient(double numerator, double denominator) {
  if (denominator == 0.0) {
     throw DivideByZeroException();
  } else {
    return numerator / denominator;
  }
}
```

[5] Czyli niestandardowa klasa wyjątku powinna dziedziczyć po jednej ze standardowych klas wyjątków.

```
int main() {
  try {
     double result = quotient(1, 2);
     std::cout << "The quotient is: " << result << std::endl;
  } catch (DivideByZeroException& divide_by_zero_exception) {
    std::cout << "Exception occurred: "
      << divide_by_zero_exception.what() << std::endl;
  }

  return EXIT_SUCCESS;
}
```

Dla zainteresowanych...

Jeśli żadna z wyspecjalizowanych standardowych klas wyjątków nie nadaje się do zastosowania w Twojej sytuacji, jako klasę macierzystą wybierz klasę `std::exception`. Musisz wówczas zaimplementować metodę `what()` (zdefiniowaną w klasie `std::exception` jako metoda czysto wirtualna), która zwraca łańcuch znaków będący opisem wyjątku – zwróć uwagę, że nagłówek tej metody zawiera specyfikator `noexcept`[6]:

```
const char* what() const noexcept;
```

Implementując tę metodę w swojej klasie nie zapominaj o umieszczeniu tego specyfikatora na końcu nagłówka metody!

15.4 Kiedy stosować wyjątki, a kiedy nie?

W czasach programowania proceduralnego (np. w języku C), w którym nie istniały mechanizmy obsługi wyjątków, powszechną praktyką było informowanie o ewentualnych problemach podczas wykonywania programu za pomocą **kodów błędu** (ang. error codes) – zwykle wartości całkowitych, z których każda miała ustalone przez danego programistę znaczenie, przykładowo:

Listing 15.3. Anty-przykład zgłaszania problemów za pomocą kodów błędów.

```
class BankAccount {
public:
  int withdraw(int amount) {
    if (amount > _balance) {
      return EXIT_FAILURE;
    }
    else {
      balance -= amount;
      return EXIT_SUCCESS;
    }
    }
// ...
private:
    int _balance;
};
```

Główne problemy w przypadku stosowania kodów błędów to:

- konieczność umieszczania w kodzie wielu instrukcji warunkowych sprawdzających poprawność wykonania danej funkcji
- brak informacji o okolicznościach wystąpienia wyjątku
- brak możliwości zastosowania np. w przypadku konstruktorów (gdyż konstruktor nie umożliwia zwracania wartości)

W sytuacji jak w przykładzie 15.3 rozsądniej jest użyć mechanizmu wyjątków – jak w przykładzie 15.4.

Listing 15.4. Przykład zgłaszania problemów za pomocą wyjątków.

```
void withdraw(int amount) {
  if (amount > _balance) {
    throw BalanceException();
```

[6] zob. noexcept specifier

```
  }
  balance -= amount;
}
```

Należy jednak pamiętać, aby zgłaszać wyjątki tylko w sytuacjach, gdy faktycznie fragment kodu nie jest w stanie poprawnie wykonać swojego zadania. Jeśli „wyjątkowa" sytuacja stanowi fragment normalnego toku wykonywania programu, wyjątek nie powinien być zgłaszany. Rozważ następujące przykłady hipotetycznej funkcji służącej do nawiązywania połączenia z serwerem:

- Funkcja `connect_once()` ma podjąć jedną próbę połączenia. W takim przypadku, jeśli funkcji nie uda się nawiązać połączenia, powinna ona zgłosić wyjątek.
- Funkcja `connect_three_times()` ma podjąć trzy próby połączenia. W takim przypadku, funkcja powinna zgłosić wyjątek dopiero w chwili, gdy trzecia próba zakończyła się niepowodzeniem.

W szczególności należy unikać stosowania mechanizmu obsługi wyjątków do kontrolowania normalnego procesu wykonywania programu, przykładowo:

Listing 15.5. Anty-przykład stosowania wyjątków do kontroli normalnego przebiegu programu.

```
void increase_count() {
  /* ... */
  if (count >= 5000) {
    throw MaximumCountReachedException();
  }
}

void use_exceptions_for_flow_control() {
  try {
    while (true) {
      increase_count();
    }
  } catch (MaximumCountReachedException& ex) {
    /* ciało klauzuli celowo puste */
  }
  /* kontynuuj wykonanie funkcji */
}
```

W powyższym przykładzie funkcja `use_exceptions_for_flow_control()` korzysta z pętli nieskończonej w celu zwiększania licznika, aż do momentu zgłoszenia wyjątku. Takie rozwiązanie sprawia, że kod jest nie tylko trudny w analizie, ale też ma niską wydajność (gdyż zgłoszenie wyjątku wiąże się m.in. z koniecznością przygotowania informacji o stanie programu w chwili jego wystąpienia w celu dokonania odwijania stosu).

Ważne

Wyjątki powinny być zgłaszane tylko w przypadku wystąpienia błędów lub w sytuacjach krytycznych – czyli gdy dalsze wykonanie programu zwyczajnie jest niemożliwe!

Rozdział 16

Przestrzenie nazw

Duże programy zwykle korzystają z niezależnie rozwijanych bibliotek, z których każda definiuje pewien zbiór nazw (np. klas, funkcji i szablonów). Biblioteki, które umieszczają swoje nazwy w **globalnej przestrzeni nazw** (ang. global namespace), powodują tzw. **zaśmiecenie przestrzeni nazw** (ang. namespace pollution). W efekcie, gdy aplikacja korzysta z wielu takich bibliotek od różnych dostawców, niemal na pewno dojdzie do **kolizji nazw** (ang. name collision) – czyli sytuacji, gdy dwie biblioteki deklarują tak samo nazwany identyfikator w ten samej przestrzeni nazw, przy czym deklaracje te różnią się między sobą, przykładowo:

```
/* biblioteka A  (A.hpp) */
int foo();  // `foo` to identyfikator funkcji
```

```
/* biblioteka B  (B.hpp) */
extern double foo;  // `foo` to identyfikator zmiennej globalnej
```

Powyższy przykład poskutkuje błędem kompilacji „redefinicja symbolu `foo` jako symbolu innego typu".

Ponieważ w języku C można definiować nazwy wyłącznie w globalnej przestrzeni nazw, programiści języka C zwykli unikać zaśmiecania tej przestrzeni poprzez stosowanie **dekorowania nazw** (ang. name decoration, name mangling)[1] – przykładowo poprzez dodanie nazwy „modułu", do którego należy dana funkcja czy zmienna:

```
// Zamiast dwóch funkcji `total()`...
double account_total(Account);
double bank_total(Account[]);
```

Takie rozwiązanie wciąż jest jednak dalekie jest od ideału – korzystanie z długich nazw jest kłopotliwe.

Przestrzenie nazw udostępniają znacznie lepiej kontrolowany mechanizm unikania kolizji oznaczeń – w danej przestrzeni nazw (np. przestrzeni globalnej) mogą istnieć podprzestrzenie, co umożliwia stworzenie hierarchii przestrzeni. Przykładowo, biblioteka standardowa języka C++ umieszcza nazwy w „standardowej" przestrzeni nazw (będącej podprzestrzenią globalnej przestrzeni nazw), czyli w przestrzeni nazw „`::std`".

16.1 Definiowanie przestrzeni nazw

Definicja przestrzeni nazw ma następującą formę:

```
namespace namespace_name {
  // Tu umieść deklaracje i definicje składowych przestrzeni nazw:
  //   klas, zmiennych, funkcji, szablonów oraz innych przestrzeni nazw
}
```

Oto przykładowa przestrzeń nazw:

```
namespace cpp_script_agh {
  void foo();
  class Bar { /* ... */ };
}
```

[1] Technikę tę stosują również kompilatory współczesnych języków programowania w celu wygenerowania unikatowych nazw funkcji, struktur, klas oraz innych typów danych. W języku C++ dekorowanie nazw służy m.in. umożliwieniu przeciążania funkcji.

Definicje przestrzeni nazw mogą być rozbite na wiele fragmentów – w sytuacji, gdy kompilator natrafia na kolejną definicję przestrzeni nazw o tej samej nazwie, po prostu dodaje jej elementy do już istniejącej przestrzeni. Pozwala to umieszczać elementy tej samej przestrzeni nazw w różnych plikach.

16.2 Korzystanie z przestrzeni nazw

Aby odwołać się do składowej przestrzeni nazw, należy użyć jej **kwalifikowanego identyfikatora** (ang. qualified identifier) – nazwę składowej należy poprzedzić **operatorem zakresu** `::` (ang. scope operator) oraz nazwą odpowiedniej przestrzeni:

```
cpp_script_agh::foo();  // wywołanie funkcji `foo()` z przestrzeni
                        // nazw `cpp_script_agh`
```

W obrębie tej samej przestrzeni nazw nie trzeba stosować identyfikatorów kwalifikowanych:

```
namespace dummy_namespace {
  int x = 2;
  int y = x;  // kwalifikacja nie jest wymagana
}
```

Co do zasady, aby odwołać się do identyfikatora `x` z globalnej przestrzeni nazw należy użyć zapisu `::x` – zatem przykładowo w pełni purystyczny zapis odwołania do obiektu `cout` ze standardowej przestrzeni nazw (zdefiniowanej w globalnej przestrzeni nazw) to „`::std::cout`”. Jednak w praktyce początkowy operator zakresu, odwołujący się do globalnej przestrzeni nazw, można pominąć – kompilator doda go automatycznie.

16.2.1 Deklaracje `using`

Korzystanie z kwalifikowanych nazw bywa kłopotliwe, dlatego możemy określić, dla których składowych przestrzeni nazw chcemy móc pominąć ich kwalifikację. Służy do tego deklaracja **`using`**, która ma następującą postać:

```
using namespace_name::name;
```

Począwszy od miejsca wystąpienia takiej deklaracji możemy pomijać prefiks `namespace_name::` przy odwoływaniu się do składowej `name`, przykładowo:

```
#include <cstdlib>
#include <iostream>

using std::cout;
// Od teraz możesz używać `cout` jako aliasu dla `std::cout`.

int main() {
  cout << "X" << std::endl;

  return EXIT_SUCCESS;
}
```

Korzystając z deklaracji **`using namespace`** `namespace_name;` możemy za pomocą jednej instrukcji zaznaczyć, że do wszystkich składowych przestrzeni nazw `namespace_name` chcemy odwoływać się bez kwalifikacji:

```
#include <cstdlib>
#include <iostream>

using namespace std;  // <- To rozwiązanie NIEZALECANE!

int main() {
  cout << "X" << endl;

  return EXIT_SUCCESS;
}
```

Ważne

Pliki nagłówkowe nie powinny zawierać deklaracji `using` (w jakiejkolwiek postaci), gdyż zawartość takich plików jest kopiowana do tekstu modułów korzystających z tych plików – w efekcie program, który nie planował korzystać z danych nazw, może natrafić na nieoczekiwane konflikty nazw.

Rozdział 17

Inne zagadnienia

Poniższy rozdział omawia kwestie, które były sygnalizowane w różnych wcześniejszych rozdziałach, lecz które nie są ściśle powiązane tematycznie z żadnym z wcześniej omawianych zagadnień.

17.1 Przeciążanie funkcji

O funkcjach, które nazywają się tak samo, znajdują się w tym samym zakresie, lecz różnią listą parametrów (liczbą parametrów i/lub ich typem), mówimy że są **przeciążone** (ang. overloaded). Przykładowo:

```
void print(const char *cp);
void print(const int *beg, const int *end);
void print(const int ia[], size_t size);
```

Wszystkie te funkcje służą do wykonywania tej samej operacji – wypisywania łańcucha znaków w stylu C, lecz obsługują parametry różnych typów.

Gdy wywołujemy taką przeciążoną funkcję, kompilator wybiera właściwą wersję funkcji na podstawie przekazanych argumentów – proces ten nosi nazwę **dopasowania funkcji** (ang. function matching, overload resolution):

```
int j[2] = {0, 1};
print("Hello World");         // wywołanie print(const char*)
print(j, end(j) - begin(j)); // wywołanie print(const int*, size_t)
print(begin(j), end(j));      // wywołanie print(const int*, const int*)
```

Mechanizm przeciążania funkcji eliminuje konieczność wymyślania (i zapamiętywania) takich nazw funkcji, które istnieją tylko po to, aby kompilator mógł określić, którą funkcję wywołać (jak w języku C).

Błędem jest deklarowanie dwóch funkcji, których nagłówki różnią się wyłącznie typem zwracanej wartości; podobnie tzw. „wysokopoziomowy” kwalifikator `const` nie sprawia, że typy parametrów są różne:

```
Record lookup(Phone);
Record lookup(const Phone);  // redeklaracja Record lookup(Phone)
Record lookup(Phone*);
Record lookup(Phone* const); // redeklaracja Record lookup(Phone*)
```

Możemy natomiast przeciążyć funkcję, gdy parametr posiada tzw. „niskopoziomowy” kwalifikator `const` (stała referencja, stały wskaźnik):

```
// cztery deklaracje tej samej przeciążonej funkcji
Record lookup(Account&);
Record lookup(const Account&);
Record lookup(Account*);
Record lookup(const Account*);
```

Powyższy mechanizm ma duże znaczenie przy zachowywaniu tzw. *const correctness*[1].

[1] zob. rozdz. 6.3 *const correctness*

17.1.1 Kiedy nie przeciążać funkcji?

Przeciążanie powinniśmy stosować wyłącznie w przypadku, gdy przeciążane funkcje realizują podobną (od strony semantycznej) operację. Rozważ poniższy przykład metod służących przemieszczaniu wskaźnika myszy na ekranie (każda z metod zwraca uchwyt do hipotetycznego okna programu znajdującego się pod wskaźnikiem myszy w jego nowym w nowym położeniu):

```
// powrót kursora do pozycji początkowej
Screen& Pointer::move_home();
// ustawienie kursora na zadaną pozycję
Screen& Pointer::move_abs(int, int);
// przemieszczenie kursora w zadanym kierunku
Screen& Pointer::move_rel(int, int, std::string direction);
```

W powyższym przykładzie *nie chcemy* definiować przeciążonej metody `Pointer::move`, gdyż każda z tych trzech dotychczasowych metod realizuje zupełnie inną (od strony semantycznej) operację.

Gdybyśmy jednak mimo wszystko zdecydowali się na dokonanie przeciążenia:

```
Screen& Pointer::move();
Screen& Pointer::move(int, int);
Screen& Pointer::move(int, int, string direction);
```

otrzymalibyśmy później w programie takie mało czytelne (na podstawie samej nazwy metody) wywołania:

```
pointer.move();  // Co ta metoda w zasadzie ma robić?!
```

Rozważając dylemat „przeciążać czy nie” warto odpowiedzieć sobie na pytanie – „która wersja wywołania jest czytelniejsza?”:

```
pointer.move_home();  // raczej ta...
pointer.move();
```

i podjąć odpowiednią decyzję projektową.

17.2 Argumenty domyślne

Argumenty domyślne (ang. default arguments) umożliwiają wywołanie funkcji bez podawania jednego lub kilku końcowych argumentów. Argumenty domyślne są sygnalizowane z użyciem poniższego zapisu w obrębie listy parametrów funkcji w miejscu jej deklaracji

```
TSpec param_name = init
```

gdzie `TSpec` to specyfikacja typu parametru, `param_name` to nazwa parametru, a `init` to domyślna wartość.

Oto przykład prototypu funkcji z domyślnymi argumentami:

```
void foo(int a, int b = 9, int c = 8);
```

Jeśli nie podasz danego końcowego argumentu, przyjęta zostanie wartość domyślna – przykładowo:

```
foo(1, 2, 3);  // równoznaczne wywołaniu: foo(1, 2, 3);
foo(1, 2);     // równoznaczne wywołaniu: foo(1, 2, 8);
foo(1);        // równoznaczne wywołaniu: foo(1, 9, 8);
```

Na chwilę obecną przyjmij, że w deklaracji funkcji wszystkie parametry począwszy od pierwszego posiadającego argument domyślny muszą również posiadać argument domyślny[2] – zatem poniższy prototyp funkcji jest błędny, gdyż parametr `c` nie posiada argumentu domyślnego, choć wcześniejszy parametr `b` go posiada:

```
void foo(int a, int b = 0, int c);  // BŁĄD: parametr `c` musi posiadać
                                    //   argument domyślny
```

Argumenty domyślne są dopuszczalne wyłącznie w obrębie listy parametrów *deklaracji* funkcji i wyrażeń lambda[3].

[2] Szczegółowe zasady dotyczące tego, kiedy wymagane jest podawanie kolejnych wartości domyślnych, zostały omówione tu.

[3] Jeśli dana funkcja posiada osobno deklarację i osobno definicję, argumenty domyślne należy podać wyłącznie w *deklaracji*.

17.3 Inicjalizacja danych

Inicjalizacja (ang. initialization) danych w języku C++ to zaskakująco złożone zagadnienie, w związku z czym w niniejszym rozdziale omówiono wyłącznie najistotniejsze kwestie.

17.3.1 Inicjalizacja to nie przypisanie!

Choć w zapisie inicjalizacja i przypisanie wyglądają podobnie (w obu przypadkach używamy symbolu „=”):

```
int i = 0;  // inicjalizacja
i = 1;      // przypisanie
```

są to w istocie dwie zupełnie różne operacje! O inicjalizacji mówimy, gdy nadajemy zmiennej (początkową) wartość w momencie jej tworzenia. Przypisanie zastępuje aktualną wartość zmiennej inną wartością.

17.3.2 Inicjalizacja domyślna

Gdy definiujemy zmienną bez jej jawnej inicjalizacji, taka zmienna jest **inicjalizowana domyślnie** (ang. default initialized) – nadawana jej jest „domyślna” wartość. To, jaka to będzie wartość, zależy m.in. od typu zmiennej oraz od miejsca jej definicji.

Wartość obiektu typu wbudowanego (np. dowolnego typu arytmetycznego), który nie został jawnie zainicjalizowany, zależy od miejsca jego definicji. Zmienne globalne są domyślnie inicjalizowane wartością 0. Zmienne lokalne (zdefiniowane wewnątrz funkcji) generalnie nie są domyślnie inicjalizowane[4] – wartość takiej niezainicjalizowanej zmiennej jest niezdefiniowana, a próba korzystania z wartości takiej zmiennej (np. odczyt wartości) prowadzi do niezdefiniowanego zachowania programu.

W przypadku obiektów typu klasowego, to od klasy zależy czy mechanizm domyślnej inicjalizacji będzie dostępny[5], a jeśli tak – jaką wartość otrzyma tak zainicjalizowany obiekt[6]:

```
std::string s;  // domyślna inicjalizacja - wartość ""

class Foo {
public:
  Foo(int) {}
};

Foo f;  // BŁĄD: klasa Foo nie pozwala na domyślną inicjalizację,
        //   gdyż jedyny dostępny konstruktor to konstruktor
        //   jednoargumentowy
```

Ważne

Niezainicjalizowane obiekty typów wbudowanych zdefiniowane w ciele funkcji mają niezdefiniowaną wartość. Wartość obiektów typu klasowego, których nie zainicjalizujemy jawnie, zależy od wartości zdefiniowanych przez daną klasę.

17.3.3 Sposoby inicjalizacji zmiennych

We współczesnym języku C++ istnieje wiele sposobów na inicjalizację zmiennych – stosuj poniższe *ogólne* zasady:

1. Stosuj zapis analogiczny do operatora przypisania w przypadku „prostej” inicjalizacji (z użyciem zestawu literałów zwyczajnie kopiowanych do inicjalizowanego obiektu danych):

```
int i = 1;
std::string s = "Hello";
std::pair<bool, double> p = {true, 2.0};
```

[4] Wyjątek stanowią lokalne zmienne statyczne, inicjalizowane domyślnie wartością 0.

[5] zob. rozdz. 9.5.5 = delete

[6] zob. rozdz. 9.2.1 Konstruktor domyślny

```
std::vector<std::string> v = {"one", "two", "three"};
```

Choć w kodzie pojawia się symbol „=”, to *nie jest* operacja przypisania – gdyż stojący po lewej stronie „=” obiekt dopiero jest tworzony! Nie są zatem tworzone żadne dodatkowe obiekty tymczasowe, nie ma narzutu wydajnościowego. Co więcej, w tym sposobie inicjalizacji kompilator dopuszcza stosowanie wyłącznie jawnie zdefiniowanych konstruktorów.

2. Stosuj składnię konstruktora, gdy inicjalizacja nie jest wystarczająco intuicyjna dla czytelnika (np. wymaga zastosowania szczególnej logiki) albo gdy wymaga jawnego wywołania konkretnego konstruktora:

```
MyClass c(1.7, false, "test");
std::vector<double> v(10, 0.97);  // Wypełnij kontener 10 kopiami obiektu
                                  //   podanego jako inicjalizator
                                  //   (tu: wartością 0.97).
```

3. Stosuj składnię „jednolitej inicjalizacji” („`{ }`” bez „=”), wprowadzoną w standardzie C++11, tylko wówczas, gdy żaden z wcześniejszych sposobów nie nadaje się do zastosowania[7].

[7] Przykład takiej sytuacji znajdziesz tu.

Podziękowania

Osoby, które pomogły w tworzeniu niniejszego skryptu, są wymienione w kolejności alfabetycznej.

Korekta

Kamil Baradziej
Dawid Bogon
Dawid Bugajny
Gustaw Cyburt
Filip Gacek
Wojciech Grzeliński
Michał Hesek
Andrzej Janik
Michał Krzyszczuk
Kamil Maćków
Jakub Mazur
Franciszek Morytko
Kacper Motyka
Michał Motyl
Michał Nedza
Tomasz Niedziela
Mateusz Pilecki
Piotr Piwoński
Michał Pogorzelec
Artur Połeć
Kamil Pytel
Łukasz Rams
Igor Ratajczyk
Michał Rola
Małgorzata Rucka
Mateusz Smolarczyk
Patryk Siwek
Maciej Stroiński
Agata Suliga
Paweł Szwarnowski
Karol Talaga
Wojciech Tokarz
Rafał Węgrzyn
Jacek Wójtowicz
Olaf Zdziebko