SAMS PUBLISHING

Poznaj język C++ w 21 dni!

- Jak nauczyć się języka C++ w 3 tygodnie?
- Jak działają wskaźniki?
- Jak wykorzystać polimorfizm w programowaniu obiektowym?

Jesse Liberty, Siddhartha Rao, Bradley L. Jones

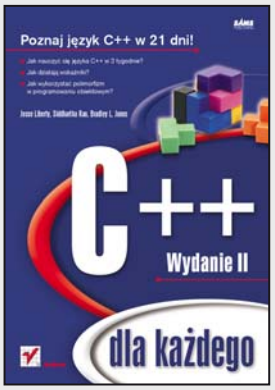
C++

Wydanie II

dla każdego

Helion

C++ dla każdego. Wydanie II



Autorzy: Jesse Liberty, Siddhartha Rao, Bradley L. Jones
Tłumaczenie: Robert Górczyński
ISBN: 978-83-246-2782-0
Tytuł oryginału: Sams Teach Yourself C++ in One Hour a Day (6th Edition)
Format: B5, stron: 850

Poznaj język C++ w 21 dni!

Pomimo swojego wieku, język C++ wciąż utrzymuje wysoką formę. Przez lata zdobył i ugruntował sobie świetną pozycję na rynku języków programowania. Dzięki temu, nawet dziś w czasach gdzie króluje język Java oraz platforma .NET, wciąż swoich zwolenników. Ba! Istnieją takie gałęzie oprogramowania, w których jest on niezastąpiony. Dlatego jeżeli oczekujesz najwyższej wydajności, operowania blisko sprzętu oraz pełnej kontroli nad sposobem wykonywania programu powinieneś postawić właśnie na C++.

Dzięki książce, którą trzymasz w ręku będzie to stosunkowo proste zadanie. Poświęcając tylko godzinę dziennie zdobędziesz wiedzę, która pozwoli Ci spokojnie rozpocząć przygodę z językiem C++, poznać jego tajniki, zalety oraz wady. Z każdą kolejną godziną będziesz zdobywać coraz to bardziej zaawansowaną i ciekawą wiedzę. Jaki więc zakres obejmuje niniejszy podręcznik? Kompletny, którego opanowanie pozwoli Ci na pisanie programów o różnym stopniu złożoności oraz swobodne poruszanie się w świecie języka C++. Zdobędziesz informacje na temat stałych, zmiennych, tablic, instrukcji warunkowych oraz pętli. Ponadto dowiesz się, jak wykorzystać wskaźniki oraz dyrektywy kompilatora. Natomiast lektura ostatnich rozdziałów książki pozwoli Ci na swobodne poruszanie się w świecie programowania obiektowego, strumieni oraz klas STL. Obiekty, klasy, dziedziczenie czy polimorfizm – te pojęcia nie skryją przed Tobą już żadnej tajemnicy! Kolejne wydanie cenionej książki sprawdzi się w rękach każdego adepta języka C++. Jeżeli chcesz poznać ten język, to nie ma na co czekać. Lepszej okazji nie będzie!

- Historia języka C++
- Konstrukcja programu w języku C++
- Komentowanie kodu
- Zmienne i stałe
- Zastosowanie tablic i ciągów tekstowych
- Formułowanie wyrażeń
- Instrukcje warunkowe i operatory
- Wykorzystanie i tworzenie funkcji
- Zakresy zmiennych
- Sterowanie przebiegiem programu
- Zasada działania wskaźników
- Programowanie obiektowe – klasy, obiekty, dziedziczenie, polimorfizm
- Rzutowanie
- Wykorzystanie strumieni
- Kompilowanie warunkowe, instrukcje kompilatora

Sprawdź czy język C++ spełni Twoje oczekiwania!

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

Spis treści

Część IV Jeszcze więcej STL

Część V Zaawansowane koncepcje C++

Dodatki

Lekcja 10

Klasy i obiekty

Klasy rozszerzają wbudowane w C++ możliwości, ułatwiające rozwiązywanie złożonych, „rzeczywistych" problemów.

Z tej lekcji dowiesz się:

- **czym są klasy i obiekty,**
- **jak definiować nową klasę oraz tworzyć obiekty tej klasy,**
- **czym są funkcje i dane składowe,**
- **czym są konstruktory i jak z nich korzystać.**

Czy C++ jest zorientowane obiektowo?

W pewnym momencie język C, czyli poprzednik C++, był najpopularniejszym językiem programowania na świecie używanym do tworzenia oprogramowania komercyjnego. Był wykorzystywany do tworzenia systemów operacyjnych (na przykład systemu Unix), do tworzenia oprogramowania działającego w czasie rzeczywistym (kontrola maszyn, urządzeń i elektroniki). Dopiero później zaczął być stosowany jako język programowania służący do budowy innych języków konwencjonalnych. Opracowanie języka C pozwoliło na znalezienie łatwiejszego i bezpieczniejszego sposobu zbliżenia oprogramowania i osprzętu.

Język C jest etapem pośrednim pomiędzy wysokopoziomowymi językami aplikacji „firmowych", takimi jak COBOL, a niskopoziomowym, wysokowydajnym, lecz trudnym do użycia asemblerem. C wymusza programowanie „strukturalne", w którym poszczególne zagadnienia są dzielone na mniejsze jednostki powtarzalnych działań, zwanych *funkcjami*, natomiast dane są umieszczane w pakietach nazywanych *strukturami*.

Jednak języki takie jak Smalltalk i CLU zaczęły torować drogę w nowym kierunku — programowania zorientowanego obiektowo. Ten rodzaj programowania pozwala na umieszczanie w pojedynczej jednostce — obiekcie — podzespołów danych takich jak struktura oraz funkcja.

Świat jest wypełniony przedmiotami: samochodami, psami, drzewami, chmurami, kwiatami. Rzeczy. Każda *rzecz* ma charakterystykę (szybki, przyjazny, brązowy, puszysty, ładny). Większość rzeczy cechuje jakieś zachowanie (ruch, szczekanie, wzrost, deszcz, uwiąd). Nie myślimy o danych samochodu i o tym, jak moglibyśmy nimi manipulować — myślimy o samochodzie jako o rzeczy: do czego jest podobny i co robi. Z tym samym możemy się zetknąć po przeniesieniu do świata komputerów dowolnego „rzeczywistego" obiektu.

Programy, które piszemy na początku dwudziestego pierwszego wieku, są dużo bardziej złożone niż te, które były pisane pod koniec wieku dwudziestego. Programy stworzone w językach proceduralnych są trudne w zarządzaniu i konserwacji, a ich rozbudowa jest niemożliwa. Graficzne interfejsy użytkownika, Internet, telefonia cyfrowa i bezprzewodowa oraz wiele innych technologii, znacznie zwiększyły poziom skomplikowania nowych projektów, a wymagania konsumentów dotyczące jakości interfejsu użytkownika wzrosły.

Programowanie zorientowane obiektowo oferuje narzędzie pomagające w sprostaniu wyzwaniom stojącym przed twórcami oprogramowania. Chociaż nie istnieje złoty środek dla tworzenia skomplikowanego oprogramowania, języki zorientowane obiektowo stworzyły silne więzy pomiędzy strukturami danych a metodami manipulowania tymi danymi. Ponadto jeszcze bardziej zbliżają się do sposobu myślenia ludzi (programistów i klientów), usprawniając tym samym komunikację oraz poprawiając jakość tworzonego oprogramowania. A co najważniejsze, w programowaniu zorientowanym obiektowo nie musisz już myśleć o strukturach danych i manipulujących nimi funkcjach; myślisz o obiektach w taki sposób, jakby były odpowiednikami obiektów w rzeczywistości: czyli jak one wyglądają i jak działają.

Język C++ stanowi pomost pomiędzy programowaniem zorientowanym obiektowo a językiem C. Celem jego autorów było stworzenie obiektowo zorientowanego języka dla tej szybkiej i efektywnej platformy służącej do tworzenia oprogramowania komercyjnego, ze szczególnym naciskiem na wysoką wydajność. W dalszej części lekcji przekonasz się, jak C++ osiąga cele postawione przed językiem.

Tworzenie nowych typów

Programy są zwykle pisane w celu rozwiązania jakiegoś realnego problemu, takiego jak prowadzenie rejestru pracowników czy symulacja działania systemu grzewczego. Choć istnieje możliwość rozwiązywania tych problemów za pomocą programów napisanych wyłącznie przy użyciu liczb całkowitych i znaków, jednak w przypadku większych, bardziej rozbudowanych problemów, dużo łatwiej jest stworzyć reprezentacje obiektów, o których się mówi. Innymi słowy, symulowanie działania systemu grzewczego będzie łatwiejsze, gdy stworzymy zmienne reprezentujące pomieszczenia, czujniki ciepła, termostaty i bojlery. Im bardziej te zmienne odpowiadają rzeczywistości, tym łatwiejsze jest napisanie programu.

Poznałeś już kilka typów zmiennych, m.in. liczby całkowite i znaki. Typ zmiennej dostarcza nam kilka informacji o niej. Na przykład, jeśli zadeklarujesz zmienne `Height` (wysokość) i `Width` (szerokość) jako liczby całkowite typu `unsigned short int`, wiesz, że w każdej z nich możesz przechować wartość z przedziału od 0 do 65 535 (przy założeniu że typ `unsigned short int` zajmuje dwa bajty pamięci). Są to liczby całkowite bez znaku; próba przechowania w nich czegokolwiek innego powoduje błąd. W zmiennej typu `unsigned short` nie możesz umieścić swojego imienia, nie powinieneś nawet próbować.

Deklarując te zmienne jako `unsigned short int`, wiesz, że możesz dodać do siebie wysokość i szerokość oraz przypisać tę wartość innej zmiennej.

Typ zmiennych informuje:

- o ich rozmiarze w pamięci,
- jaki rodzaj informacji mogą zawierać,
- jakie działania można na nich wykonywać.

W tradycyjnych językach, takich jak C, typy były wbudowane w język. W C++ programista może rozszerzyć język, tworząc potrzebne mu typy, zaś każdy z tych nowych typów może być w pełni funkcjonalny i dysponować tą samą siłą, co typy wbudowane.

Wady tworzenia typów za pomocą słowa kluczowego struct

Pewne możliwości rozszerzenia języka C nowymi typami wiążą się z łączeniem w struktury (`struct`) powiązanych ze sobą zmiennych, które mogą być dostępne w postaci nowego typu danych za pomocą instrukcji `typedef`. Jednak istnieją pewne wady takiego podejścia:

- Struktury i operujące na nich funkcje nie są spójnymi całościami. Funkcje można znaleźć jedynie poprzez odczyt plików nagłówkowych z dostępnych bibliotek i wyszukanie tych z nowym typem jako parametrem.
- Koordynacja aktywności grupy powiązanych funkcji w strukturze jest trudniejszym zadaniem, ponieważ dowolny fragment logiki programu może w dowolnej chwili zmienić cokolwiek w strukturze. Z tego powodu nie ma możliwości ochrony struktury przed ingerencją.
- Wbudowane operatory nie działają ze strukturami — nie działa nawet dodanie dwóch struktur za pomocą znaku plus (+), choć taka operacja mogłaby się wydawać najbardziej naturalnym sposobem przedstawienia rozwiązania problemu (na przykład, kiedy każda struktura przedstawia skomplikowane fragmenty tekstu, które mają być ze sobą połączone).

Wprowadzenie klas i elementów składowych

Nowy typ zmiennych tworzy się, deklarując klasę. *Klasa* jest właściwie grupą zmiennych — często o różnych typach — skojarzonych z zestawem odnoszących się do nich funkcji.

Jedną z możliwości myślenia o samochodzie jest potraktowanie go jako zbioru kół, drzwi, foteli, okien itd. Inna możliwość to wyobrażenie sobie, co samochód może zrobić: jeździć, przyspieszać, zwalniać, zatrzymywać się, parkować itd. Klasa umożliwia hermetyzację, czyli upakowanie, tych różnych części oraz różnych działań w jeden zbiór, który jest nazywany obiektem.

Upakowanie wszystkiego, co wiesz o samochodzie, w jedną klasę przynosi programiście liczne korzyści. Wszystko jest na miejscu, ułatwia to odwoływanie się, kopiowanie i manipulowanie danymi. Klienci Twojej klasy — tj. te części programu, które z niej korzystają — mogą używać Twojego obiektu bez zastanawiania się, co znajduje się w środku i jak on działa.

Klasa może składać się z dowolnej kombinacji zmiennych prostych oraz zmiennych innych klas. Zmienna wewnątrz klasy jest nazywana *zmienną składową* lub *daną składową*. Klasa `Car` (samochód) może posiadać składowe reprezentujące siedzenia, typ radia, opony itd. Zmienne składowe są zmiennymi w danej klasie. Stanowią one część klasy, tak jak koła i silnik stanowią część samochodu.

Klasa może zawierać również funkcje, które są wówczas nazywane *funkcjami składowymi* lub *metodami*. Podobnie jak zmienne składowe, stanowią część klasy i określają, co dana klasa może zrobić.

Funkcje w danej klasie zwykle manipulują zmiennymi składowymi. Przykładowo, metodami klasy `Car` mogą być `Start()` (uruchom) oraz `Brake()` (hamuj). Klasa `Cat` (kot) może posiadać zmienne składowe, reprezentujące wiek i wagę; jej metodami mogą być `Sleep()` (śpij), `Meow()` (miaucz) czy `ChaseMice()` (łap myszy).

Deklarowanie klasy

Deklaracja klasy informuje kompilator o jej istnieniu. Aby zadeklarować klasę, użyj słowa kluczowego `class`, po którym następuje otwierający nawias klamrowy, a następnie lista danych składowych i metod tej klasy. Deklaracja kończy się zamykającym nawiasem klamrowym i średnikiem. Oto deklaracja klasy o nazwie `Cat` (kot):

```
class Cat
{
    unsigned int  itsAge;
    unsigned int  itsWeight;
    void Meow();
};
```

Zadeklarowanie takiej klasy nie powoduje zaalokowania pamięci dla obiektu `Cat`. Informuje jedynie kompilator, czym jest typ `Cat`, jakie dane zawiera (`itsAge` — jego wiek oraz `itsWeight` — jego waga) oraz co może robić (`Meow()` — miaucz). Informuje także kompilator, jak duża jest zmienna typu `Cat` — to jest, jak dużo miejsca w pamięci ma przygotować w przypadku tworzenia zmiennej typu `Cat`. W tym przykładzie, o ile typ `int` ma cztery bajty, zmienna typu `Cat` zajmuje osiem bajtów: cztery bajty dla zmiennej `itsAge` i cztery dla zmiennej `itsWeight`. Funkcja `Meow()` nie zajmuje miejsca, gdyż dla funkcji składowych (metod) miejsce nie jest rezerwowane. Jest to wskaźnik do funkcji, który na platformie 32-bitowej może zająć cztery bajty.

Kilka słów o konwencji nazw

Jako programista, musisz nazwać wszystkie swoje zmienne składowe, funkcje składowe oraz klasy. Jak przeczytałeś w lekcji 3., „Stałe i zmienne”, nazwy te powinny być zrozumiałe i znaczące. Dobrymi nazwami klas mogą być wspomniana

`Cat`, `Rectangle` (prostokąt) czy `Employee` (pracownik). `Meow()`, `ChaseMice()` czy `StopEngine()` (zatrzymaj silnik) również są dobrymi nazwami funkcji, gdyż informują, co robią te funkcje. Wielu programistów nadaje nazwom zmiennych składowych przedrostek „its" (jego), tak jak w zmiennych `itsAge`, `itsWeight` czy `itsSpeed` (jego szybkość). Pomaga to w odróżnieniu zmiennych składowych od innych zmiennych.

Niektórzy programiści wolą przedrostek „my" (mój), tak jak w nazwach `myAge`, `myWeight` czy `mySpeed`. Jeszcze inni używają po prostu litery `m` (od słowa *member* — składowa), czasem wraz ze znakiem podkreślenia (_): `mAge` i `m_age`, `mWeight` i `m_weight` czy `mSpeed` i `m_speed`.

Niektórzy programiści lubią poprzedzić każdą nazwę klasy określoną literą — na przykład `cCat` czy `cPerson` — podczas, gdy inni używają wyłącznie dużych lub małych liter. Ja sam korzystam z konwencji, w której wszystkie nazwy klas rozpoczynają się od dużej litery, tak jak `Cat` czy `Person` (osoba).

Wielu programistów rozpoczyna wszystkie nazwy funkcji od dużej litery, zaś wszystkie nazwy zmiennych — od małej. Słowa zwykle rozdzielane są znakiem podkreślenia — tak jak w `Chase_Mice` — lub poprzez zastosowanie dużej litery dla każdego słowa — na przykład `ChaseMice` czy `DrawCircle` (rysuj okrąg).

Ważne jest, by wybrać określony styl i trzymać się go w każdym programie. Z czasem rozwiniesz swój styl nie tylko na konwencje nazw, ale także na wcięcia, wyrównanie nawiasów klamrowych oraz styl komentarzy.

Uwaga

W firmach programistycznych powszechne jest określenie standardu wielu elementów stylu zapisu kodu źródłowego. Sprawia on, że wszyscy programiści mogą łatwo odczytywać wzajemnie swój kod. Niestety, przenosi się to również do firm opracowujących systemy operacyjne i biblioteki klas przeznaczonych do ponownego użytku. Zazwyczaj oznacza to, że w programach C++ jest jednocześnie stosowanych wiele różnych konwencji nazw.

Ostrzeżenie

Jak już wcześniej wspomniano, język C++ uwzględnia wielkość liter, dlatego wszystkie nazwy klas powinny przestrzegać tej samej konwencji. Dzięki temu nigdy nie będziesz musiał sprawdzać pisowni nazwy klasy (czy to było `Rectangle`, `rectangle` czy `RECTANGLE`?).

Definiowanie obiektu

Po zadeklarowaniu klasy można jej następnie używać jako nowego typu dla definiowanych zmiennych tego typu. Definiowanie obiektu nowego typu przypomina definiowanie zmiennej całkowitej:

```
unsigned int GrossWeight; // definicja zmiennej typu unsigned int
Cat Frisky;               // definicja zmiennej typu Cat
```

Ten kod definiuje zmienną o nazwie `GrossWeight` (łączna waga), której typem jest `unsigned int`. Oprócz tego definiuje zmienną o nazwie `Frisky`, która jest obiektem klasy (typu) `Cat`.

Klasy a obiekty

Nigdy nie karmi się definicji kota, lecz konkretnego kota. Należy dokonać rozróżnienia pomiędzy ideą kota a konkretnym kotem, który właśnie ociera się o Twoje nogi. C++ również dokonuje rozróżnienia pomiędzy klasą `Cat`, będącą ideą kota, a poszczególnymi obiektami typu `Cat`. Tak więc `Frisky` jest obiektem typu `Cat`, tak jak `GrossWeight` jest zmienną typu `unsigned int`. Obiekt jest indywidualnym egzemplarzem klasy.

Dostęp do elementów składowych klasy

Gdy zdefiniujesz już faktyczny obiekt `Cat` — na przykład:

```
Cat Frisky;
```

w celu uzyskania dostępu do jego składowych możesz użyć operatora kropki (`.`). Aby zmiennej składowej `itsWeight` obiektu `Frisky` przypisać wartość `50`, powinieneś napisać:

```
Frisky.itsWeight = 50;
```

Aby wywołać funkcję `Meow()`, możesz napisać:

```
Frisky.Meow();
```

Gdy używasz metody klasy, oznacza to, że wywołujesz tę metodę. W tym przykładzie wywołałeś metodę `Meow()` obiektu `Frisky`.

Przypisywać należy obiektom, nie klasom

W C++ nie przypisuje się wartości typom; przypisuje się je zmiennym. Na przykład, nie można napisać:

```
int = 5;              // źle
```

Kompilator uzna to za błąd, gdyż nie można przypisać wartości pięć typowi całkowitemu. Zamiast tego musisz zdefiniować zmienną typu całkowitego i przypisać jej wartość 5. Na przykład:

```
int x ;               // definicja zmiennej typu int
x = 5;                // ustawienie wartości zmiennej x na 5
```

Jest to skrócony zapis stwierdzenia: „Przypisz wartość 5 zmiennej x, która jest zmienną typu int". Nie można również napisać:

```
Cat.itsAge = 5;       // źle
```

Kompilator uzna to za błąd, gdyż nie możesz przypisać wartości 5 do elementu itsAge klasy Cat. Zamiast tego musisz zdefiniować egzemplarz obiektu klasy Cat i dopiero wtedy przypisać wartość jego składowej. Na przykład:

```
Cat Frisky;           // podobnie jak  int x;
Frisky.itsAge = 5;    // podobnie jak  x = 5;
```

Czego nie zadeklarujesz, tego klasa nie będzie miała

Przeprowadź taki eksperyment: podejdź do trzylatka i pokaż mu kota. Następnie powiedz: To jest Frisky. Frisky zna sztuczkę. Frisky, zaszczekaj! Dziecko roześmieje się i powie: „Nie, głuptasie, koty nie szczekają!".

Jeśli napisałeś:

```
Cat Frisky;           // tworzy obiekt Cat o nazwie Frisky
Frisky.Bark();        // nakazuje Friskiemu szczekać
```

Kompilator wyświetli komunikat: „Nie, głuptasie, koty (cats) nie szczekają!". (Być może w Twoim kompilatorze ten komunikat będzie brzmiał nieco inaczej). Kompilator wie, że Frisky nie może szczekać, gdyż klasa Cat nie posiada metody Bark() (szczekaj). Kompilator nie pozwoli Friskiemu nawet zamiauczeć, jeśli nie zdefiniujesz dla niego funkcji Meow() (miaucz).

Tak	Nie
Do deklarowania klasy używaj słowa kluczowego `class`. W celu uzyskania dostępu do zmiennych i funkcji składowych klasy używaj operatora kropki (`.`).	Nie myl deklaracji z definicją. Deklaracja mówi czym jest klasa, a definicja przygotowuje pamięć dla obiektu. Nie myl klasy z obiektem. Nie przypisuj klasie wartości. Wartości przypisuj danym składowym obiektu.

Dostęp prywatny i publiczny

W deklaracji klasy używanych jest także kilka innych słów kluczowych. Dwa najważniejsze z nich to: `public` (publiczny) i `private` (prywatny).

Słowa kluczowe `private` i `public` są używane wraz z egzemplarzami klasy — zarówno ze zmiennymi składowymi, jak i metodami klasy. Prywatne składowe mogą być używane tylko przez metody należące do danej klasy. Składowe publiczne są dostępne dla innych funkcji i klas. To rozróżnienie jest ważne, choć na początku może sprawiać kłopot. Wszystkie składowe klasy — dane i metody — są domyślnie prywatne. Aby to lepiej wyjaśnić, spójrzmy na poprzedni przykład:

```
class Cat
{
    unsigned int  itsAge;
    unsigned int  itsWeight;
    void Meow();
};
```

W tej deklaracji, składowe `itsAge`, `itsWeight` oraz `Meow()` są prywatne, gdyż wszystkie składowe klasy są prywatne domyślnie. Oznacza to, że dopóki nie postanowisz inaczej, pozostaną one prywatne. Jeśli jednak w funkcji `main()` napiszesz na przykład:

```
Cat Bobas;
Bobas.itsAge = 5;      // błąd! nie można używać prywatnych danych!
```

kompilator uzna to za błąd. We wcześniejszej deklaracji powiedziałeś kompilatorowi, że składowych `itsAge`, `itsWeight` oraz `Meow()` będziesz używał tylko w funkcjach składowych klasy `Cat`. W powyższym fragmencie kodu próbujesz odwołać się do zmiennej składowej obiektu `Bobas` spoza metody klasy `Cat`. To, że `Bobas` jest obiektem klasy `Cat`, nie oznacza, że możesz korzystać z tych elementów obiektu `Bobas`, które są prywatne.

Właśnie to jest źródłem niekończących się kłopotów początkujących programistów C++. Już słyszę, jak narzekasz: „Hej! Właśnie napisałem, że Bobas jest kotem, tj. obiektem klasy `Cat`. Dlaczego `Bobas` nie ma dostępu do swojego własnego wieku?". Odpowiedź brzmi: `Bobas` ma dostęp, ale Ty nie masz. `Bobas`, w swoich własnych metodach, ma dostęp do wszystkich swoich składowych, zarówno publicznych, jak i prywatnych. Nawet jeśli to Ty tworzysz obiekt klasy `Cat`, nie możesz przeglądać ani zmieniać tych jego składowych, które są prywatne.

Aby mieć dostęp do składowych obiektu `Cat`, pewne elementy składowe powinny być publiczne:

```
class Cat
{
    public:
        unsigned int  itsAge;
        unsigned int  itsWeight;
        void Meow();
};
```

Teraz składowe `itsAge`, `itsWeight` oraz `Meow()` są publiczne. `Bobas.itsAge = 5;` kompiluje się bez problemów.

Uwaga

> Słowo kluczowe `public` ma zastosowanie względem wszystkich elementów składowych w danej deklaracji aż do napotkania słowa kluczowego `private` i na odwrót. W ten sposób sekcje klasy można bardzo łatwo zadeklarować jako publiczne bądź prywatne.

Listing 10.1 przedstawia deklarację klasy `Cat` z publicznymi zmiennymi składowymi.

Listing 10.1. Dostęp do publicznych składowych w prostej klasie

```
 1:  // Demonstruje deklarację klasy oraz
 2:  // definicje obiektu tej klasy.
 3:
 4:  #include <iostream>
 5:
 6:  class Cat           // deklaruję klasę Cat (kot)
 7:  {
 8:      public:         // następujące po tym składowe są publiczne
 9:          int itsAge;     // zmienna składowa
10:          int itsWeight;  // zmienna składowa
11:  };                      // zwróć uwagę na średnik
12:
13:  int main()
14:  {
15:      Cat Frisky;
```

```
16:      Frisky.itsAge = 5;    // przypisanie do zmiennej składowej
17:      std::cout << "Frisky jest kotem i ma " ;
18:      std::cout << Frisky.itsAge << " lat.\n";
19:      return 0;
20:  }
```

Wynik ▼

```
Frisky jest kotem i ma 5 lat.
```

Analiza ▼

Linia 6. zawiera słowo kluczowe `class`. Informuje ono kompilator, że następuje po nim deklaracja klasy. Nazwa nowej klasy następuje bezpośrednio po słowie kluczowym `class`. W tym przypadku nazwą klasy jest `Cat` (kot).

Ciało deklaracji rozpoczyna się w linii 7. od otwierającego nawiasu klamrowego i kończy się zamykającym nawiasem klamrowym i średnikiem w linii 11. Linia 8. zawiera słowo kluczowe `public`, które wskazuje, że wszystko, co po nim nastąpi, będzie publiczne, aż do natrafienia na słowo kluczowe `private` lub koniec deklaracji klasy. Linie 9. i 10. zawierają deklaracje składowych klasy, `itsAge` (jego wiek) oraz `itsWeight` (jego waga).

W linii 13. rozpoczyna się funkcja `main()`. W linii 15. `Frisky` jest definiowany jako egzemplarz klasy `Cat` — tj. jako obiekt klasy `Cat`. W linii 16. wiek `Friska` jest ustawiany na 5. W liniach 17. i 18. zmienna składowa `itsAge` zostaje użyta do wyświetlenia informacji o kocie `Frisky`. Powinieneś zauważyć, jak w liniach 16. i 18. następuje uzyskanie dostępu do obiektu `Frisky`. Zmienna `itsAge` jest dostępna poprzez nazwę obiektu (w tym przykładzie `Frisky`), następnie kropkę i nazwę zmiennej składowej (w tym przypadku `itsAge`).

Uwaga

Spróbuj wykomentować linię 8., po czym skompiluj program ponownie. W linii 16. wystąpi błąd, gdyż zmienna składowa `itsAge` nie będzie już składową publiczną. Domyślnie, wszystkie składowe klasy są prywatne.

Oznaczanie danych składowych jako prywatnych

Powinieneś przyjąć jako ogólną regułę, że dane składowe klasy należy utrzymywać jako prywatne. Oczywiście, możesz się zastanawiać, w jaki sposób uzyskać informacje dotyczące klasy, jeśli wszystkie elementy składowe będą prywatne.

Przykładowo, jeśli zmienna `itsAge` będzie prywatna, jak można odczytać bądź ustawić wiek obiektu `Cat`?

W celu uzyskania dostępu do danych prywatnych w klasie musisz stworzyć publiczne funkcje składowe, zwane *akcesorami*. Funkcje te umożliwią odczyt zmiennych składowych i przypisywanie im wartości. Te funkcje dostępowe (akcesory) są funkcjami składowymi, używanymi przez inne części programu w celu odczytywania i ustawiania prywatnych zmiennych składowych. Publiczny akcesor jest funkcją składową klasy, używaną albo do odczytu wartości prywatnej zmiennej składowej klasy, albo do ustawiania wartości tej zmiennej.

Dlaczego miałbyś utrudniać sobie życie dodatkowym poziomem pośredniego dostępu? Po co dodawać kolejne funkcje, skoro prościej i łatwiej jest bezpośrednio posługiwać się danymi? Jaki cel ma stosowanie akcesorów?

Odpowiedź na te pytania jest całkiem prosta: akcesory umożliwiają oddzielenie szczegółów *przechowywania* danych klasy od szczegółów jej *używania*. Dzięki temu możesz zmieniać sposób przechowywania danych klasy bez konieczności przepisywania funkcji, które z tych danych korzystają.

Jeśli funkcja, która chce poznać wiek kota, odwoła się bezpośrednio do zmiennej `itsAge` klasy `Cat`, będzie musiała zostać przepisana, jeżeli Ty, jako autor klasy `Cat`, zdecydujesz się na zmianę sposobu przechowywania tej zmiennej. Jednak posiadając funkcję składową `GetAge()` (pobierz wiek), klasa `Cat` może łatwo zwrócić właściwą wartość bez względu na to, w jaki sposób przechowywany będzie wiek. Funkcja wywołująca nie musi wiedzieć, czy jest on przechowywany jako zmienna typu `unsigned int` czy `long` lub czy wiek jest obliczany w miarę potrzeb.

Ta technika ułatwia zapanowanie nad programem. Przedłuża istnienie kodu, gdyż zmiany projektowe nie powodują, że program staje się przestarzały.

Oprócz tego funkcje akcesorów mogą zawierać dodatkową logikę — na przykład mało prawdopodobne jest, aby wiek kota (obiektu `Cat`) przekroczył wartość 100, a jego waga przekroczyła wartość 1000. Wymienione wartości powinny być niedozwolone. Funkcja akcesora może wymusić stosowanie tego rodzaju ograniczeń, jak również wykonywać wiele innych zadań.

Listing 10.2 przedstawia klasę `Cat` zmodyfikowaną tak, by zawierała prywatne dane składowe i publiczne akcesory. Zwróć uwagę, że ten listing przedstawia wyłącznie deklarację klasy, nie ma w nim kodu wykonywalnego.

Listing 10.2. Klasa z akcesorami

```
 1:  // Deklaracja klasy Cat
 2:  // Dane składowe są prywatne, publiczne akcesory pośredniczą
 3:  // w ustawianiu i odczytywaniu wartości składowych prywatnych
 4:  class Cat
 5:  {
 6:      public:
 7:          // publiczne akcesory
 8:          unsigned int GetAge();
 9:          void SetAge(unsigned int Age);
10:
11:          unsigned int GetWeight();
12:          void SetWeight(unsigned int Weight);
13:
14:          // publiczna funkcja składowa
15:          void Meow();
16:
17:          // prywatne dane składowe
18:      private:
19:          unsigned int  itsAge;
20:          unsigned int  itsWeight;
21:  };
```

Analiza

Ta klasa posiada pięć metod publicznych. Linie 8. i 9. zawierają akcesory dla składowej `itsAge`. Możesz zobaczyć, że w linii 8. znajduje się metoda pobierająca wiek kota, natomiast w linii 9. metoda ustawiająca wiek kota. Linie 11. i 12. zawierają podobne akcesory dla składowej `itsWeight`. Te akcesory ustawiają zmienne składowe i zwracają ich wartości.

W linii 15. jest zadeklarowana publiczna funkcja składowa `Meow()`. Ta funkcja nie jest akcesorem. Nie zwraca wartości ani ich nie ustawia; wykonuje inną usługę dla klasy — wyświetla słowo `Miau`. Zmienne składowe są zadeklarowane w liniach 19. i 20.

Aby ustawić wiek `Friska`, powinieneś przekazać wartość metodzie `SetAge()` (ustaw wiek), na przykład:

```
Cat Frisky;
Frisky.SetAge(5); // ustawia wiek Friska
                  // używając publicznego akcesora
```

W dalszej części lekcji poznasz kod, dzięki któremu metoda `SetAge()` oraz inne mogą działać.

Zadeklarowanie metod lub danych jako prywatnych umożliwia kompilatorowi wyszukanie w programach pomyłek, zanim staną się one błędami. Każdy szanujący się programista potrafi znaleźć sposób na obejście prywatności składowych. Stroustrup, autor języka C++, stwierdza, że „...mechanizmy ochrony z poziomu języka chronią przed pomyłką, a nie przed świadomym oszustwem" (WNT, 1995).

Słowo kluczowe class

Składnia słowa kluczowego `class` jest następująca:

```
class nazwa_klasy
{
    // słowa kluczowe kontroli dostępu
    // zadeklarowane zmienne i składowe klasy
};
```

Słowo kluczowe `class` służy do deklarowania nowych typów. Klasa stanowi zbiór danych składowych klasy, które są zmiennymi różnych typów, także innych klas. Klasa zawiera także funkcje klasy — tzw. metody — które są funkcjami używanymi do manipulowania danymi w danej klasie i wykonywania innych usług dla klasy.

Obiekty nowego typu definiuje się w taki sam sposób, w jaki definiuje się inne zmienne. Należy określić typ (klasę), a po nim nazwę zmiennej (obiektu). Do uzyskania dostępu do funkcji i danych klasy służy operator kropki (`.`).

Słowa kluczowe kontroli dostępu określają, które sekcje klasy są prywatne, a które publiczne. Domyślnie wszystkie składowe klasy są prywatne. Każde słowo kluczowe zmienia kontrolę dostępu od danego miejsca aż do końca klasy, lub kontrolę wystąpienia następnego słowa kluczowego kontroli dostępu. Deklaracja klasy kończy się zamykającym nawiasem klamrowym i średnikiem.

Przykład 1

```
class Cat
{
    public:
        unsigned int Age;
        unsigned int Weight;
        void Meow();
};

Cat Frisky;
Frisky.Age = 8;
Frisky.Weight = 18;
Frisky.Meow();
Przykład 2
class Car
{
```

```
public:           // pięć następnych składowych jest publicznych
        void Start();
        void Accelerate();
        void Brake();
        void SetYear(int year);
        int GetYear();

    private:  // pozostała część jest prywatna
        int Year;
        char Model [255];
};      // koniec deklaracji klasy

Car OldFaithful;   // tworzy egzemplarz klasy
int bought;  // lokalna zmienna typu int
OldFaithful.SetYear(84);  // ustawia składową Year na 84
bought = OldFaithful.GetYear(); // ustawia zmienną bought na 84
OldFaithful.Start(); //wywołuje metodę Start
```

Tak	Nie
Używaj publicznych akcesorów, czyli publicznych funkcji dostępowych.	Nie deklaruj zmiennych składowych jako publiczne, o ile nie ma takiej potrzeby.
Odwołuj się do prywatnych zmiennych składowych z funkcji składowych klasy.	Nie używaj prywatnych zmiennych składowych klasy poza tą klasą.

Implementowanie metod klasy

Jak już się przekonałeś, akcesory stanowią publiczny interfejs do prywatnych danych klasy. Każdy akcesor musi posiadać, wraz z innymi zadeklarowanymi metodami klasy, implementację. Implementacja jest nazywana *definicją funkcji*.

Definicja funkcji składowej rozpoczyna się podobnie jak definicja zwykłej funkcji. W pierwszej kolejności trzeba podać typ zwracany z danej funkcji lub użyć słowa kluczowego `void`, jeśli funkcja niczego nie zwraca. Następnie podajemy nazwę klasy, po której występują dwa dwukropki, nazwa funkcji i jej parametry. Listing 10.3 przedstawia pełną deklarację prostej klasy `Cat`, wraz z implementacją jej akcesorów i jednej ogólnej funkcji tej klasy.

Listing 10.3. Implementacja metod prostej klasy

```
 1: // Demonstruje deklarowanie klasy oraz
 2: // definiowanie jej metod
 3: #include <iostream>        // dla cout
 4:
 5: class Cat                   // początek deklaracji klasy
 6: {
 7:     public:                      // początek sekcji publicznej
 8:         int GetAge();            // akcesor
 9:         void SetAge(int age);    // akcesor
10:         void Meow();             // ogólna funkcja
11:     private:                     // początek sekcji prywatnej
12:         int itsAge;              // zmienna składowa
13: };
14:
15: // GetAge, publiczny akcesor
16: // zwracający wartość składowej itsAge
17: int Cat::GetAge()
18: {
19:     return itsAge;
20: }
21:
22: // definicja SetAge, akcesora
23: // publicznego
24: // ustawiającego składową itsAge
25: void Cat::SetAge(int age)
26: {
27:     // ustawia zmienną składową itsAge
28:     // zgodnie z wartością przekazaną w parametrze age
29:     itsAge = age;
30: }
31:
32: // definicja metody Meow
33: // zwraca: void
34: // parametery: brak
35: // działanie: wyświetla na ekranie słowo "miauczy"
36: void Cat::Meow()
37: {
38:     std::cout << "Miauczy.\n";
39: }
40:
41: // tworzy kota, ustawia jego wiek, sprawia,
42: // że miauczy, wyświetla jego wiek i ponownie miauczy.
43: int main()
44: {
45:     Cat Frisky;
46:     Frisky.SetAge(5);
47:     Frisky.Meow();
```

```
48:      std::cout << "Frisky jest kotem i ma " ;
49:      std::cout << Frisky.GetAge() << " lat.\n";
50:      Frisky.Meow();
51:      return 0;
52:  }
```

Wynik ▼

```
Miauczy.
Frisky jest kotem i ma 5 lat.
Miauczy.
```

Analiza ▼

Linie od 5. do 13. zawierają definicję klasy `Cat` (kot). Linia 7. zawiera słowo kluczowe `public`, które informuje kompilator, że to, co po nim następuje, jest zestawem publicznych składowych. Linia 8. zawiera deklarację publicznego akcesora `GetAge()` (pobierz wiek). `GetAge()` zapewnia dostęp do prywatnej zmiennej składowej `itsAge` (jego wiek), zadeklarowanej w linii 12. Linia 9. zawiera publiczny akcesor `SetAge()` (ustaw wiek). Funkcja `SetAge()` otrzymuje parametr typu `int`, który następnie przypisuje składowej `itsAge`.

Linia 10. zawiera deklarację metody `Meow()` (miaucz). Funkcja `Meow()` nie jest akcesorem. Jest to ogólna metoda klasy, wyświetlająca na ekranie słowo „Miauczy.”

Linia 11. rozpoczyna sekcję prywatną, która obejmuje jedynie zadeklarowaną w linii 12. prywatną składową `itsAge`. Deklaracja klasy kończy się w linii 13. zamykającym nawiasem klamrowym i średnikiem.

Linie od 17. do 20. zawierają definicję składowej funkcji `GetAge()`. Ta metoda nie ma parametrów i zwraca wartość całkowitą. Zauważ, że ta metoda klasy zawiera nazwę klasy, dwa dwukropki oraz nazwę funkcji (linia 17.). Ta składnia informuje kompilator, że definiowana funkcja `GetAge()` jest właśnie tą funkcją, która została zadeklarowana w klasie `Cat`. Poza formatem tego nagłówka, definiowanie funkcji własnej `GetAge()` niczym nie różni się od definiowania innych (zwykłych) funkcji.

Funkcja `GetAge()` posiada tylko jedną linię i zwraca po prostu wartość zmiennej składowej `itsAge`. Zauważ, że funkcja `main()` nie ma dostępu do tej zmiennej składowej, gdyż jest ona prywatna dla klasy `Cat`. Funkcja `main()` ma za to dostęp do publicznej metody `GetAge()`.

Ponieważ ta metoda jest składową klasy `Cat`, ma pełny dostęp do zmiennej `itsAge`. Dzięki temu może zwrócić funkcji `main()` wartość zmiennej `itsAge`.

Linia 25. zawiera definicję funkcji składowej `SetAge()`. Możesz zauważyć, że ta funkcja pobiera jedną wartość w postaci liczby całkowitej (`age`) i nie zwraca żadnej wartości, co jest wskazane przez słowo kluczowe `void`. Ta funkcja posiada parametr w postaci wartości całkowitej i przypisuje składowej `itsAge` jego wartość (linia 29.). Ponieważ jest składową klasy `Cat`, ma bezpośredni dostęp do jej zmiennych prywatnych i publicznych.

Linia 36. rozpoczyna definicję (czyli implementację) metody `Meow()` klasy `Cat`. Jest to jednoliniowa funkcja wyświetlająca na ekranie słowo „Miauczy", zakończone znakiem nowej linii. Pamiętaj, że znak `\n` powoduje przejście do nowej linii. Możesz zauważyć, że metoda `Meow()` jest zaimplementowana podobnie jak funkcja akcesora, czyli rozpoczyna się od typu zwracanej wartości, a następnie mamy nazwę klasy, nazwę funkcji oraz parametry (w tym przypadku nie ma żadnych parametrów).

Linia 43. rozpoczyna ciało funkcji `main()`, czyli właściwy program. W linii 45., funkcja `main()` deklaruje obiekt `Cat` o nazwie `Frisky`. Ujmując to inaczej, można powiedzieć, że funkcja `main()` deklaruje kota (`Cat`) o imieniu `Frisky`.

W linii 46. zmiennej `itsAge` tego obiektu jest przypisywana wartość 5 (poprzez użycie akcesora `SetAge()`). Zauważ, że wywołanie tej metody następuje dzięki użyciu nazwy obiektu (`Frisky`), po której zastosowano operator kropki (`.`) i nazwę metody (`SetAge()`). W podobny sposób wywoływane są wszystkie inne metody wszystkich klas.

Uwaga
Uwaga

Pojęcia *funkcja składowa* i *metoda składowa* mogą być używane zamiennie.

Linia 47. wywołuje funkcję składową `Meow()`, zaś w linii 49. za pomocą akcesora `GetAge()`, wyświetlany jest komunikat. Linia 50. ponownie wywołuje funkcję `Meow()`. Choć wymienione metody są częścią klasy (`Cat`) i używane poprzez obiekt (`Frisky`), to działają dokładnie tak samo jak funkcje, które widziałeś do tej pory.

Konstruktory i destruktory

Istnieją dwa sposoby definiowania zmiennej całkowitej. Można zdefiniować zmienną, a następnie, w dalszej części programu, przypisać jej wartość. Na przykład:

```
int Weight;  // definiujemy zmienną
...          // tu inny kod
Weight = 7;  // przypisujemy jej wartość
```

Możemy też zdefiniować zmienną i jednocześnie zainicjalizować ją. Na przykład:

```
int Weight = 7;  // definiujemy i inicjalizujemy wartością 7
```

Inicjalizacja łączy w sobie definiowanie zmiennej oraz początkowe przypisanie wartości. Nic nie stoi na przeszkodzie temu, by zmienić później wartość zmiennej. Inicjalizacja powoduje tylko, że zmienna nigdy nie będzie pozbawiona sensownej wartości.

W jaki sposób zainicjalizować składowe klasy? Klasy posiadają specjalne funkcje składowe, zwane konstruktorami. *Konstruktor* (ang. *constructor*) może w razie potrzeby posiadać parametry, ale nie może zwracać wartości — nawet typu `void`. Konstruktor jest metodą klasy o takiej samej nazwie, jak nazwa klasy.

Gdy zadeklarujesz konstruktor, powinieneś także zadeklarować *destruktor* (ang. *destructor*). Konstruktor tworzy i inicjalizuje obiekt danej klasy, zaś destruktor porządkuje obiekt i zwalnia pamięć, którą mogłeś w niej zaalokować (albo w konstruktorze, albo w trakcie cyklu życiowego obiektu). Destruktor zawsze nosi nazwę klasy, poprzedzoną znakiem tyldy (~). Destruktory nie mają argumentów i nie zwracają wartości. Dlatego deklaracja destruktora klasy `Cat` ma następującą postać:

```
~Cat();
```

Domyślne konstruktory i destruktory

Istnieje wiele rodzajów konstruktorów; niektóre z nich posiadają argumenty, inne nie. Konstruktor, którego można wywołać bez żadnych argumentów, jest nazywany konstruktorem *domyślnym*. Istnieje tylko jeden rodzaj destruktora. On także nie posiada argumentów.

Jeśli nie stworzysz konstruktora lub destruktora, kompilator stworzy je za Ciebie. Konstruktor dostarczany przez kompilator jest konstruktorem domyślnym — czyli konstruktorem bez argumentów. Stworzone przez kompilator domyślny konstruktor i destruktor nie mają żadnych argumentów, a na dodatek w ogóle nic nie robią! Jeżeli chcesz, aby wykonywały jakiekolwiek operacje, musisz samodzielnie utworzyć domyślny konstruktor i destruktor.

Użycie domyślnego konstruktora

Do czego może przydać się konstruktor, który nic nie robi? Jest to problem techniczny: wszystkie obiekty muszą być konstruowane i niszczone, dlatego w odpowiednich momentach wywoływane są te nic nie robiące funkcje.

Aby móc zadeklarować obiekt bez przekazywania parametrów, na przykład

```
Cat Boots;  // Boots nie ma parametrów
```

musisz posiadać konstruktor w postaci

```
Cat();
```

Gdy definiujesz obiekt klasy, wywoływany jest konstruktor. Gdyby konstruktor klasy `Cat` miał dwa parametry, mógłbyś zdefiniować obiekt `Cat`, pisząc

```
Cat Frisky (5, 7);
```

W tym przypadku pierwszy parametr mógłby być wiekiem, natomiast drugi mógłby oznaczać wagę kota. Gdyby konstruktor miał jeden parametr, napisałbyś

```
Cat Frisky (3);
```

W przypadku, gdy konstruktor nie ma żadnych parametrów (gdy jest konstruktorem *domyślnym*), możesz opuścić nawiasy i napisać

```
Cat Frisky;
```

Jest to wyjątek od reguły, zgodnie z którą wszystkie funkcje wymagają zastosowania nawiasów, nawet jeśli nie mają parametrów. Właśnie dlatego możesz napisać:

```
Cat Frisky;
```

Jest to interpretowane jako wywołanie konstruktora domyślnego. Nie dostarczamy mu parametrów i pomijamy nawiasy.

Zwróć uwagę, że nie musisz używać domyślnego konstruktora dostarczanego przez kompilator. Zawsze możesz napisać własny konstruktor domyślny — tj. konstruktor bez parametrów. Możesz zastosować w nim ciało funkcji, w którym możesz zainicjalizować obiekt. Zgodnie z konwencją, zawsze zaleca się zdefiniowanie konstruktora i ustawienie zmiennym składowym odpowiednich wartości domyślnych w celu zagwarantowania prawidłowego zachowania obiektu.

Zgodnie z konwencją, gdy deklarujesz konstruktor, powinieneś także zadeklarować destruktor, nawet jeśli nie robi on niczego. Nawet jeśli destruktor domyślny będzie działał poprawnie, nie zaszkodzi zadeklarować własnego. Dzięki niemu kod staje się bardziej przejrzysty.

Listing 10.4 zawiera nową wersję klasy Cat, w której do zainicjalizowania obiektu kota nie użyto domyślnego konstruktora. Wiek kota został ustawiony zgodnie z wartością otrzymaną jako parametr konstruktora. Listing pokazuje również miejsce, w którym wywoływany jest destruktor.

Listing 10.4. Użycie konstruktora i destruktora

```
// Demonstruje deklarowanie konstruktora
// i destruktora dla klasy Cat
// Domyślny konstruktor został stworzony przez programistę
#include <iostream>            // dla cout

class Cat                      // początek deklaracji klasy
{
    public:                        // początek sekcji publicznej
        Cat(int initialAge);       // konstruktor
        ~Cat();                    // destruktor
        int GetAge();              // akcesor
        void SetAge(int age);      // akcesor
        void Meow();
    private:                       // początek sekcji prywatnej
        int itsAge;                // zmienna składowa
};

// konstruktor klasy Cat
Cat::Cat(int initialAge)
{
   itsAge = initialAge;
}

Cat::~Cat()                    // destruktor, nic nie robi
{
}

// GetAge, publiczny akcesor
// zwraca wartość składowej itsAge
int Cat::GetAge()
{
    return itsAge;
}

// definicja SetAge, akcesora
// publicznego
void Cat::SetAge(int age)
{
    // ustawia zmienną składową itsAge
    // zgodnie z wartością przekazaną w parametrze age
    itsAge = age;
```

```
42:  }
43:
44:  // definicja metody Meow
45:  // zwraca: void
46:  // parametery: brak
47:  // działanie: wyświetla na ekranie słowo "miauczy"
48:  void Cat::Meow()
49:  {
50:      std::cout << "Miauczy.\n";
51:  }
52:
53:  // tworzy kota, ustawia jego wiek, sprawia,
54:  // że miauczy, wyświetla jego wiek i ponownie miauczy.
55:  int main()
56:  {
57:      Cat Frisky(5);
58:      Frisky.Meow();
59:      std::cout << "Frisky jest kotem i ma " ;
60:      std::cout << Frisky.GetAge() << " lat.\n";
61:      Frisky.Meow();
62:      Frisky.SetAge(7);
63:      std::cout << "Teraz Frisky ma " ;
64:      std::cout << Frisky.GetAge() << " lat.\n";
65:      return 0;
66:  }
```

Wynik ▼

```
Miauczy.
Frisky jest kotem i ma 5 lat.
Miauczy.
Teraz Frisky ma 7 lat.
```

Analiza ▼

Listing 10.4 przypomina listing 10.3, jednak w linii 9. dodano konstruktor, posiadający argument w postaci wartości całkowitej. Linia 10. deklaruje destruktor, który nie posiada parametrów. Destruktory nigdy nie mają parametrów, zaś destruktory i konstruktory nie zwracają żadnych wartości — nawet typu `void`.

Linie od 19. do 22. zawierają implementację konstruktora. Jest ona podobna do implementacji akcesora `SetAge()`. Jak możesz zobaczyć, nazwa klasy poprzedza nazwę konstruktora. Jak już wcześniej wspomniano, to identyfikuje metodę (w tym przypadku `Cat()`) jako część klasy `Cat`. Ponieważ wymieniona metoda jest konstruktorem, to nie zwraca żadnej wartości, nawet `void`. Jednak w linii 21. konstruktor pobiera wartość początkową przypisaną danej składowej `itsAge`.

Linie od 24. do 26. przedstawiają implementację destruktora `~Cat()`. W tej chwili ta funkcja nie robi nic, ale jeśli deklaracja klasy zawiera deklarację destruktora, zdefiniowany musi zostać wtedy także ten destruktor. Podobnie jak w przypadku konstruktora oraz innych metod, destruktor jest poprzedzony nazwą klasy. Podobnie jak w przypadku konstruktora, ale odmiennie niż w przypadku innych metod, nie podaje się zwracanego typu oraz parametrów. To jest standardowy wymóg dla destruktorów.

Linia 57. zawiera definicję obiektu `Frisky`, stanowiącego egzemplarz klasy `Cat`. Do konstruktora obiektu `Frisky` przekazywana jest wartość 5. Nie ma potrzeby wywoływania funkcji `SetAge()`, gdyż `Frisky` został stworzony z wartością 5 znajdującą się w zmiennej składowej `itsAge`, tak jak pokazano w linii 60. W linii 62. zmiennej `itsAge` obiektu `Frisky` jest przypisywana wartość 7. Tę nową wartość wyświetla linia 64.

Tak	Nie
W celu zainicjalizowania obiektów używaj konstruktorów.	Pamiętaj, że konstruktory i destruktory nie mogą zwracać wartości.
Jeżeli dodałeś konstruktor, dodaj również destruktor.	Pamiętaj, że destruktory nie mogą mieć parametrów.

Funkcje składowe const

Słowo kluczowe `const` służy do deklarowania zmiennych, które nie ulegają zmianie. `Const` można stosować również wraz z funkcjami składowymi w klasie. Jeśli zadeklarujesz metodę klasy jako `const`, obiecujesz w ten sposób, że metoda ta nie zmieni wartości żadnej ze składowych klasy.

Aby zadeklarować metodę w ten sposób, umieść słowo kluczowe `const` za nawiasami, lecz przed średnikiem. Przykładowo:

```
void SomeFunction() const;
```

Pokazana powyżej deklaracja funkcji składowej `const` o nazwie `SomeFunction()` nie posiada argumentów i zwraca typ `void`. Wiesz, że powyższa funkcja nie zmieni żadnych elementów składowych w klasie, ponieważ została zadeklarowana wraz ze słowem kluczowym `const`.

Wraz z modyfikatorem `const` często deklarowane są akcesory, które jedynie pobierają wartość. Przedstawiona wcześniej klasa `Cat` posiada dwa akcesory:

```
void SetAge(int anAge);
int GetAge();
```

Funkcja `SetAge()` nie może być funkcją `const`, gdyż modyfikuje wartość zmiennej składowej `itsAge`. Natomiast funkcja `GetAge()` może i powinna być `const`, gdyż nie modyfikuje wartości żadnej ze składowych klasy. Funkcja `GetAge()` po prostu zwraca bieżącą wartość składowej `itsAge`. Zatem deklaracje tych funkcji można przepisać następująco:

```
void SetAge(int anAge);
int GetAge() const;
```

Gdy zadeklarujesz funkcję jako `const`, zaś implementacja tej funkcji modyfikuje obiekt poprzez modyfikację wartości którejkolwiek ze składowych, kompilator zgłosi błąd. Na przykład, gdy napiszesz funkcję `GetAge()` w taki sposób, że będziesz zapamiętywał ilość zapytań o wiek kota, spowodujesz błąd kompilacji. Jest to spowodowane tym, że wywołując tę metodę, modyfikujesz zawartość obiektu `Cat`.

Deklarowanie funkcji jako `const` wszędzie tam, gdzie jest to możliwe, należy do tradycji programistycznej. Za każdym razem, gdy to zrobisz, umożliwisz kompilatorowi wykrycie pomyłki, zanim stanie się ona błędem, który ujawni się już podczas działania programu.

Po co używać kompilatora do wykrywania błędów?

Gdyby można było tworzyć programy w stu procentach pozbawione błędów, byłoby cudowanie, jednak tylko bardzo niewielu programistów jest w stanie tego dokonać. Wielu programistów opracowało jednak system pozwalający zminimalizować ilość błędów przez wczesne ich wykrycie i poprawienie.

Choć błędy kompilatora są irytujące i stanowią dla programisty przekleństwo, jednak są czymś dużo lepszym niż opisana dalej alternatywa. Język o słabej kontroli typów umożliwia naruszanie zasad kontraktu bez słowa sprzeciwu ze strony kompilatora, jednak program może załamać się w trakcie działania — na przykład wtedy, gdy pracuje z nim Twój szef. Co gorsze, w takim przypadku testowanie w celu wychwycenia tych błędów będzie mało pomocne, ponieważ istnieje zbyt dużo ścieżek wykonywania kodu rzeczywistych programów, aby móc przetestować je wszystkie.

Błędy *czasu kompilacji* — tj. błędy wykryte podczas kompilowania programu — są zdecydowanie lepsze niż błędy *czasu działania* — tj. błędy wykryte podczas działania programu. Są lepsze, gdyż dużo łatwiej i precyzyjniej można określić

ich przyczynę. Może się zdarzyć, że program zostanie wykonany wielokrotnie bez wykonania wszystkich istniejących ścieżek wykonania kodu. Dlatego błąd czasu działania może przez dłuższy czas pozostać niezauważony. Błędy kompilacji są wykrywane podczas każdej kompilacji, są więc dużo łatwiejsze do zidentyfikowania i poprawienia. Celem dobrego programowania jest ochrona przed pojawianiem się błędów czasu działania. Jedną ze znanych i sprawdzonych technik jest wykorzystanie kompilatora do wykrycia pomyłek już na wczesnym etapie tworzenia programu.

Gdzie umieszczać deklaracje klasy i definicje metod

Każda funkcja, którą zadeklarujesz dla klasy, musi posiadać definicję. Definicja jest nazywana także *implementacją funkcji*. Podobnie jak w przypadku innych funkcji, definicja metody klasy posiada nagłówek i ciało.

Definicja musi znajdować się w pliku, który może zostać znaleziony przez kompilator. Większość kompilatorów C++ wymaga, by taki plik miał rozszerzenie *.c* lub *.cpp*. W tej książce korzystamy z rozszerzenia *.cpp*, ale aby mieć pewność, sprawdź, czego oczekuje Twój kompilator.

Uwaga

Wiele kompilatorów zakłada, że pliki z rozszerzeniem *.c* są programami C, zaś pliki z rozszerzeniem *.cpp* są programami C++. Możesz używać dowolnego rozszerzenia, ale rozszerzenie *.cpp* wyeliminuje ewentualne nieporozumienia.

W pliku, w którym umieszczasz implementację funkcji, możesz umieścić również jej deklarację, ale nie należy to do dobrych obyczajów. Zgodnie z konwencją zaadoptowaną przez większość programistów, deklaracje umieszcza się w tak zwanych plikach nagłówkowych, zwykle posiadających tę samą nazwę, lecz z rozszerzeniem *.h*, *.hp* lub *.hpp*. W tej książce dla plików nagłówkowych stosujemy rozszerzenie *.h*, ale sprawdź w swoim kompilatorze, jakie rozszerzenie powinieneś stosować.

Na przykład, deklarację klasy `Cat` powinieneś umieścić w pliku o nazwie *Cat.h*, zaś definicję metod tej klasy w pliku o nazwie *Cat.cpp*. Następnie powinieneś dołączyć do pliku *.cpp* plik nagłówkowy, poprzez umieszczenie na początku pliku *Cat.cpp* następującej dyrektywy:

```
#include "Cat.h"
```

Informuje ona kompilator, by wstawił w tym miejscu zawartość pliku *Cat.h* tak, jakbyś ją wpisał ręcznie. Uwaga: niektóre kompilatory nalegają, by wielkość liter w nazwie pliku w dyrektywie `#include` zgadzała się z wielkością liter w nazwie pliku na dysku.

Dlaczego masz się trudzić, rozdzielając program na pliki *.h* i *.cpp*, skoro i tak plik *.h* jest wstawiany do pliku *.cpp*? W większości przypadków klienci klasy nie dbają o szczegóły jej implementacji. Odczytanie pliku nagłówkowego daje im wystarczającą ilość informacji, by zignorować plik implementacji. Poza tym, ten sam plik *.h* możesz dołączać do wielu różnych plików *.cpp*.

Uwaga
Uwaga

> Deklaracja klasy mówi kompilatorowi, czym jest ta klasa, jakie dane zawiera oraz jakie funkcje posiada. Deklaracja klasy jest nazywana jej *interfejsem*, gdyż informuje kompilator w jaki sposób ma z nią współdziałać. Ten interfejs jest zwykle przechowywany w pliku *.h*, często nazywanym plikiem nagłówkowym.
>
> Definicja funkcji mówi kompilatorowi, jak działa dana funkcja. Definicja funkcji jest nazywana implementacją metody klasy i jest przechowywana w pliku *.cpp*. Szczegóły dotyczące implementacji klasy należą wyłącznie do jej autora. Klienci klasy — tj. części programu używające tej klasy — nie muszą, ani nie powinny wiedzieć, jak zaimplementowane zostały funkcje.

Implementacja inline

Możesz poprosić kompilator, by uczynił zwykłą funkcję funkcją `inline`, funkcjami `inline` mogą stać się również metody klasy. W tym celu należy umieścić słowo kluczowe `inline` przed typem zwracanej wartości. Na przykład, implementacja `inline` funkcji `GetWeight()` wygląda następująco:

```
inline int Cat::GetWeight()
{
    return itsWeight;  // zwraca daną składową itsWeight
}
```

Definicję funkcji można także umieścić w deklaracji klasy, co automatycznie sprawia, że ta funkcja staje się funkcją `inline`. Na przykład:

```
class Cat
{
    public:
        int GetWeight() { return itsWeight; }  // inline
        void SetWeight(int aWeight);
};
```

Zwróć uwagę na składnię definicji funkcji `GetWeight()`. Ciało funkcji `inline` zaczyna się natychmiast po deklaracji metody klasy; po nawiasach nie występuje średnik. Podobnie jak w innych funkcjach, definicja zaczyna się od otwierającego nawiasu klamrowego i kończy zamykającym nawiasem klamrowym. Jak zwykle, białe znaki nie mają znaczenia; możesz zapisać tę deklarację jako:

```
class Cat
{
    public:
        int GetWeight() const
        {
            return itsWeight;
        }  // inline
        void SetWeight(int aWeight);
};
```

Listingi 10.5 i 10.6 odtwarzają klasę `Cat`, tym razem jednak deklaracja klasy została umieszczona w pliku *Cat.h*, zaś jej definicja w pliku *Cat.cpp*. Oprócz tego, na listingu 10.5 akcesor `Meow()` został zadeklarowany jako funkcja `inline`.

Listing 10.5. Deklaracja klasy Cat w pliku CAT.h

```
 1:  #include <iostream>
 2:  class Cat
 3:  {
 4:      public:
 5:          Cat (int initialAge);
 6:          ~Cat();
 7:          int GetAge() const { return itsAge;}        // inline!
 8:          void SetAge (int age) { itsAge = age;}        // inline!
 9:          void Meow() const { std::cout << "Miauczy.\n";}  // inline!
10:      private:
11:      int itsAge;
12:  };
```

Listing 10.6. Implementacja klasy Cat w pliku CAT.cpp

```
 1:  // Demonstruje funkcje inline
 2:  // oraz dołączanie pliku nagłówkowego
 3:  // pamiętaj o włączeniu plików nagłówkowych!
 4:  #include "Cat.h"
 5:
 6:
 7:  Cat::Cat(int initialAge)   //konstruktor
 8:  {
 9:      itsAge = initialAge;
10:  }
```

```
11:
12:  Cat::~Cat()             //destruktor, nic nie robi
13:  {
14:  }
15:
16:  // tworzy kota, ustawia jego wiek, sprawia
17:  // że miauczy, wyświetla jego wiek i ponownie miauczy.
18:  int main()
19:  {
20:      Cat Frisky(5);
21:      Frisky.Meow();
22:      std::cout << "Frisky jest kotem i ma " ;
23:      std::cout << Frisky.GetAge() << " lat.\n";
24:      Frisky.Meow();
25:      Frisky.SetAge(7);
26:      std::cout << "Teraz Frisky ma " ;
27:      std::cout << Frisky.GetAge() << " lat.\n";
28:      return 0;
29:  }
```

Wynik ▼

```
Miauczy.
Frisky jest kotem i ma 5 lat.
Miauczy.
Teraz Frisky ma 7 lat.
```

Analiza ▼

Kod zaprezentowany na listingach 10.5 i 10.6 jest podobny do kodu z listingu 10.4, trzy metody zostały zadeklarowane w pliku deklaracji jako `inline`, a deklaracja została przeniesiona do pliku *Cat.hpp* (listing 10.5).

Funkcja `GetAge()` jest deklarowana w linii 6. pliku *Cat.h*, gdzie znajduje się także jej implementacja `inline`. Linie 7. i 8. zawierają kolejne funkcje `inline`, jednak w stosunku do poprzednich, „zwykłych” implementacji, działanie tych funkcji nie zmienia się.

Linia 4. listingu 10.6 zawiera dyrektywę `#include "Cat.h"`, która powoduje wstawienie do pliku zawartości pliku *Cat.h*. Dołączając plik *Cat.h*, informujesz prekompilator, by odczytał zawartość tego pliku i wstawił ją w miejscu wystąpienia dyrektywy `#include` (tak jakbyś, począwszy od linii 5, sam wpisał tę zawartość).

Ta technika umożliwia umieszczenie deklaracji w pliku innym niż implementacja, a jednocześnie zapewnienie kompilatorowi dostępu do niej. W programach C++

technika ta jest powszechnie wykorzystywana. Zwykle deklaracje klas znajdują się w plikach *.h*, które są dołączane do powiązanych z nimi plików *.cpp* za pomocą dyrektyw `#include`.

Linie od 18. do 29. stanowią powtórzenie funkcji `main()` z listingu 10.4. Oznacza to, że funkcje `inline` działają tak samo jak zwykłe funkcje.

Klasy, których danymi składowymi są inne klasy

Budowanie złożonych klas przez deklarowanie prostszych klas i dołączanie ich do deklaracji bardziej skomplikowanej klasy nie jest niczym niezwykłym. Na przykład, możesz zadeklarować klasę koła, klasę silnika, klasę skrzyni biegów itd., a następnie połączyć je w klasę „samochód". Deklaruje to relację posiadania. Samochód posiada silnik, koła i skrzynię biegów.

Weźmy inny przykład. Prostokąt składa się z odcinków. Odcinek jest zdefiniowany przez dwa punkty. Punkt jest zdefiniowany przez współrzędną x i współrzędną y. Listing 10.7 przedstawia pełną deklarację klasy `Rectangle` (prostokąt), która może wystąpić w pliku *Rectangle.h*. Ponieważ prostokąt jest zdefiniowany jako cztery odcinki łączące cztery punkty, zaś każdy punkt odnosi się do współrzędnej w układzie, najpierw zadeklarujemy klasę `Point` (punkt) jako przechowującą współrzędne x oraz y punktu. Listing 10.8 zawiera implementacje obu klas.

Listing 10.7. Deklarowanie kompletnej klasy

```
// początek Rect.h
#include <iostream>
class Point     // przechowuje współrzędne x,y
{
    // bez konstruktora, używa domyślnego
    public:
        void SetX(int x) { itsX = x; }
        void SetY(int y) { itsY = y; }
        int GetX()const { return itsX;}
        int GetY()const { return itsY;}
    private:
        int itsX;
        int itsY;
};    // koniec deklaracji klasy Point

```

```
16:
17:  class  Rectangle
18:  {
19:      public:
20:          Rectangle (int top, int left, int bottom, int right);
21:          ~Rectangle () {}
22:
23:          int GetTop() const { return itsTop; }
24:          int GetLeft() const { return itsLeft; }
25:          int GetBottom() const { return itsBottom; }
26:          int GetRight() const { return itsRight; }
27:
28:          Point  GetUpperLeft() const { return itsUpperLeft; }
29:          Point  GetLowerLeft() const { return itsLowerLeft; }
30:          Point  GetUpperRight() const { return itsUpperRight; }
31:          Point  GetLowerRight() const { return itsLowerRight; }
32:
33:          void SetUpperLeft(Point Location)  {itsUpperLeft = Location;}
34:          void SetLowerLeft(Point Location)  {itsLowerLeft = Location;}
35:          void SetUpperRight(Point Location)  {itsUpperRight = Location;}
36:          void SetLowerRight(Point Location)  {itsLowerRight = Location;}
37:
38:          void SetTop(int top) { itsTop = top; }
39:          void SetLeft (int left) { itsLeft = left; }
40:          void SetBottom (int bottom) { itsBottom = bottom; }
41:          void SetRight (int right) { itsRight = right; }
42:
43:          int GetArea() const;
44:
45:      private:
46:          Point  itsUpperLeft;
47:          Point  itsUpperRight;
48:          Point  itsLowerLeft;
49:          Point  itsLowerRight;
50:          int    itsTop;
51:          int    itsLeft;
52:          int    itsBottom;
53:          int    itsRight;
54:  };
55:  // koniec Rect.h
```

Listing 10.8. Rect.cpp

```
 1:  // początek rect.cpp
 2:  #include "rect.h"
 3:  Rectangle::Rectangle(int top, int left, int bottom, int right)
```

```
 4:  {
 5:      itsTop = top;
 6:      itsLeft = left;
 7:      itsBottom = bottom;
 8:      itsRight = right;
 9:
10:      itsUpperLeft.SetX(left);
11:      itsUpperLeft.SetY(top);
12:
13:      itsUpperRight.SetX(right);
14:      itsUpperRight.SetY(top);
15:
16:      itsLowerLeft.SetX(left);
17:      itsLowerLeft.SetY(bottom);
18:
19:      itsLowerRight.SetX(right);
20:      itsLowerRight.SetY(bottom);
21:  }
22:
23:
24:  // oblicza obszar prostokąta przez obliczenie
25:  // i pomnożenie szerokości i wysokości
26:  int Rectangle::GetArea() const
27:  {
28:      int Width = itsRight-itsLeft;
29:      int Height = itsTop - itsBottom;
30:      return (Width * Height);
31:  }
32:
33:  int main()
34:  {
35:      //inicjalizuje lokalną zmienną typu Rectangle
36:      Rectangle MyRectangle (100, 20, 50, 80 );
37:
38:      int Area = MyRectangle.GetArea();
39:
40:      std::cout << "Obszar: " << Area << "\n";
41:      std::cout << "Wsp. X lewego górnego rogu: ";
42:      std::cout << MyRectangle.GetUpperLeft().GetX();
43:      return 0;
44:  }
```

Wynik ▼

```
Obszar: 3000
Wsp. X lewego górnego rogu: 20
```

Analiza ▼

Linie od 3. do 14. listingu 10.7 deklarują klasę `Point` (punkt), która służy do przechowywania współrzędnych `x` i `y` określonego punktu rysunku. W tym programie nie wykorzystujemy należycie klasy `Point`. Jej zastosowania wymagają jednak inne metody rysunkowe.

Uwaga
Uwaga

> Gdy nadasz klasie nazwę `Rectangle`, niektóre kompilatory zgłoszą błąd. Wynika to z istnienia wewnętrznej klasy o nazwie `Rectangle`. W takim przypadku po prostu zmień nazwę klasy na `myRectangle`.

W deklaracji klasy `Point`, w liniach 12. i 13., zadeklarowaliśmy dwie zmienne składowe (`itsX` oraz `itsY`). Te zmienne przechowują współrzędne punktu. Zakładamy, że współrzędna `x` rośnie w prawo, a współrzędna `y` w górę. Istnieją także inne systemy. W niektórych programach okienkowych współrzędna `y` rośnie „w dół" okna.

Klasa `Point` używa akcesorów `inline`, zwracających i ustawiających współrzędne `X` i `Y` punktu. Te akcesory zostały zadeklarowane w liniach od 7. do 10. Punkty używają konstruktora i destruktora domyślnego. W związku z tym ich współrzędne trzeba ustawiać jawnie.

Linia 17. rozpoczyna deklarację klasy `Rectangle` (prostokąt). Klasa ta składa się z czterech punktów reprezentujących cztery narożniki prostokąta. Konstruktor klasy `Rectangle` (linia 20.) otrzymuje cztery wartości całkowite, `top` (górna), `left` (lewa), `bottom` (dolna) oraz `right` (prawa). Do czterech zmiennych składowych (listing 10.8) kopiowane są cztery parametry konstruktora i tworzone są cztery punkty.

Oprócz standardowych akcesorów, klasa `Rectangle` posiada funkcję `GetArea()` (pobierz obszar), zadeklarowaną w linii 43. Zamiast przechowywać obszar w zmiennej, funkcja `GetArea()` oblicza go w liniach od 28. do 30. listingu 10.8. W tym celu oblicza szerokość i wysokość prostokąta, następnie mnoży je przez siebie.

Uzyskanie współrzędnej `x` lewego górnego wierzchołka prostokąta wymaga dostępu do punktu `UpperLeft` (lewy górny) i zapytania o jego współrzędną `X`. Ponieważ funkcja `GetUpperLeft()` jest funkcją klasy `Rectangle`, może ona bezpośrednio odwoływać się do prywatnych danych tej klasy, włącznie ze zmienną (`itsUpperLeft`). Ponieważ `itsUpperLeft` jest obiektem klasy `Point`, a zmienna `itsX` tej klasy jest prywatna, funkcja `GetUpperLeft()` nie może odwoływać się do niej bezpośrednio. Zamiast tego, w celu uzyskania tej wartości musi użyć publicznego akcesora `GetX()`.

Linia 33. listingu 10.8 stanowi początek ciała programu. Pamięć nie jest alokowana aż do linii 36.; w obszarze tym nic się nie dzieje. Jedyna rzecz, jaką zrobiliśmy, to poinformowanie kompilatora, jak ma stworzyć punkt i prostokąt (gdyby były potrzebne w przyszłości). W linii 36. definiujemy obiekt typu `Rectangle`, przekazując mu wartości `top`, `left`, `bottom` oraz `right`.

W linii 38. tworzymy lokalną zmienną `Area` (obszar) typu `int`. Ta zmienna przechowuje obszar stworzonego przez nas prostokąta. Zmienną `Area` inicjalizujemy za pomocą wartości zwróconej przez funkcję `GetArea()` klasy `Rectangle`. Klient klasy `Rectangle` może stworzyć obiekt tej klasy i uzyskać jego obszar, nie znając nawet implementacji funkcji `GetArea()`.

Plik *Rect.h* został przedstawiony na listingu 10.7. Obserwując plik nagłówkowy, który zawiera deklarację klasy `Rectangle`, programista może wysnuć wniosek, że funkcja `GetArea()` zwraca wartość typu `int`. Sposób, w jaki funkcja `GetArea()` uzyskuje tę wartość, nie interesuje klientów klasy `Rectangle`. Autor klasy `Rectangle` mógłby zmienić funkcję `GetArea()`; nie wpłynęłoby to na programy, które z niej korzystają, o ile wartością zwrotną nadal byłaby liczba całkowita.

Linia 42. listingu 10.8 może wydawać się nieco dziwna, ale jeśli zastanowisz się nad tym, co tam się dzieje, powinna stać się jasna. W tej linii kodu następuje pobranie współrzędnej x lewego górnego punktu prostokąta. Względem prostokąta wywoływana jest metoda `GetUpperLeft()`, która zwraca obiekt typu `Point`. Z tego obiektu potrzebujemy współrzędnej x. Wcześniej dowiedziałeś się, że akcesorem dla punktu x klasy `Point` jest `GetX()`. Dlatego też w celu pobrania współrzędnej x lewego górnego narożnika prostokąta należy użyć akcesora `GetX()` względem funkcji akcesora `GetUpperLeft()` obiektu `MyRectangle`. To właśnie przedstawiono w linii 42.:

```
MyRectangle.GetUpperLeft().GetX();
```

Często zadawane pytanie

Jaka jest różnica pomiędzy deklaracją a definicją?

Odpowiedź

Deklaracja wprowadza nową nazwę, lecz nie alokuje pamięci; dokonuje tego definicja.

Wszystkie deklaracje (z kilkoma wyjątkami) są także definicjami. Najważniejszym wyjątkiem jest deklaracja funkcji globalnej (prototyp) oraz deklaracja klasy (zwykle w pliku nagłówkowym).

Struktury

Bardzo bliskim kuzynem słowa kluczowego `class` jest słowo kluczowe `struct`, używane do deklarowania struktur. W C++ struktura jest odpowiednikiem klasy, ale wszystkie jej składowe są domyślnie publiczne. Możesz zadeklarować strukturę dokładnie tak, jak klasę; możesz zastosować w niej te same zmienne i funkcje składowe. Gdy przestrzegasz jawnego deklarowania publicznych i prywatnych sekcji klasy, nie ma żadnej różnicy pomiędzy klasą a strukturą. Spróbuj wprowadzić do listingu 10.7 następujące zmiany:

- w linii 3., zmień `class Point` na `struct Point`,
- w linii 17., zmień `class Rectangle` na `struct Rectangle`.

Następnie skompiluj i uruchom program. Otrzymane wyniki nie powinny się od siebie różnić.

Prawdopodobnie zastanawiasz się, dlaczego dwa słowa kluczowe spełniają tę samą funkcję. Przyczyn należy szukać w historii języka. Język C++ powstawał jako rozszerzenie języka C. Język C posiada struktury, ale nie posiadają one metod. Bjarne Stroustrup, twórca języka C++, rozbudował struktury, ale zmienił ich nazwę na klasy, odzwierciedlając w ten sposób ich nowe, rozszerzone możliwości oraz zmianę w domyślnej dostępności elementów składowych. To pozwala również na używanie w programach C++ ogromnej biblioteki funkcji języka C.

Tak	Nie
Umieszczaj deklarację klasy w pliku nagłówkowym *.h*, zaś funkcje składowe definiuj w pliku *.cpp*. Używaj `const` wszędzie tam, gdzie jest to możliwe.	Nie przechodź dalej, zanim dokładnie nie zrozumiesz zasady działania klasy.

Podsumowanie

W tej lekcji nauczyłeś się tworzyć nowe typy danych za pomocą klas. Wiesz także, jak definiować zmienne tych nowych typów, które nazywamy *obiektami*.

Klasa może mieć dane składowe, którymi są zmienne różnych typów, łącznie z innymi klasami. Poza tym, klasa może zawierać funkcje składowe — nazywane również *metodami*. Wymienione funkcje składowe służą do manipulowania danymi składowymi oraz przeprowadzania innych operacji.

Elementy składowe klasy, zarówno dane, jak i funkcje, mogą być publiczne bądź prywatne. Elementy publiczne są dostępne w dowolnym miejscu programu. Z kolei elementy prywatne są dostępne jedynie dla funkcji składowych danej klasy. Domyślnie, elementy składowe są definiowane jako prywatne.

Dobrym nawykiem w programowaniu jest izolowanie interfejsu — inaczej deklaracji — klasy w pliku nagłówkowym. Zazwyczaj odbywa się to poprzez użycie pliku z rozszerzeniem *.h*, a następnie wykorzystanie go w plikach kodu (z rozszerzeniem *.cpp*) za pomocą instrukcji `include`. Implementacja metod klasy znajduje się w pliku z rozszerzeniem *.cpp*.

Konstruktory klasy mogą być używane do inicjalizacji danych składowych obiektu. Natomiast destruktory są wykonywane w trakcie niszczenia obiektu. W destruktorze bardzo często następuje zwolnienie pamięci i innych zasobów, które mogły zostać zaalokowane przez metody klasy.

Pytania i odpowiedzi

Pytanie: Jak duży jest obiekt klasy?

Odpowiedź: Wielkość obiektu w pamięci zależy od wielkości jego zmiennych składowych. Metody klasy zużywają bardzo niewielkie ilości pamięci, która jest wykorzystywana do przechowywania informacji o położeniu metody (wskaźnika).

Niektóre kompilatory umieszczają zmienne w pamięci w taki sposób, że dwubajtowe zmienne zużywają więcej niż tylko dwa bajty pamięci. Zajrzyj do dokumentacji używanego kompilatora, aby to sprawdzić. Jednak na tym etapie nie należy przejmować się takimi szczegółami.

Pytanie: Jeżeli zadeklaruję klasę `Cat` wraz z prywatnym elementem składowym `itsAge`, a następnie zdefiniuję dwa obiekty `Frisky` i `Boots`, czy obiekt `Boots` będzie miał dostęp do zmiennej składowej `itsAge` obiektu `Frisky`?

Odpowiedź: Nie. Różne egzemplarze klasy mogą uzyskać dostęp jedynie do danych publicznych innych egzemplarzy tej klasy. Innymi słowy, jeżeli `Frisky` i `Boots` to egzemplarze klasy `Cat`, funkcje składowe obiektu `Frisky` mogą uzyskać dostęp do danych obiektu `Frisky`, ale już nie do danych obiektu `Boots`.

Pytanie: Dlaczego nie powinienem definiować jako publicznych wszystkich danych składowych?

Odpowiedź: Zdefiniowanie danych składowych jako prywatne umożliwia klientowi klasy używanie tych danych bez żadnej zależności dotyczącej sposobu ich przechowywania bądź obliczania. Przykładowo, jeżeli klasa `Cat` (kot) ma metodę `GetAge()`, klient klasy `Cat` może zapytać obiekt o wiek kota, nie znając i nie przejmując się sposobem, w jaki klasa `Cat` przechowuje ten wiek bądź jak oblicza go w locie. Oznacza to, że programista klasy `Cat` może w przyszłości zmienić projekt tej klasy bez wymagania od użytkowników klasy `Cat` zmiany ich programów.

Pytanie: Jeżeli użycie funkcji `const` do zmiany klasy powoduje błąd w trakcie kompilacji, to dlaczego nie powinienem po prostu pozbyć się słów kluczowych `const`, by mieć pewność, że te błędy nie wystąpią?

Odpowiedź: Jeżeli funkcja składowa nie powinna zmieniać klasy, to użycie słowa kluczowego `const` jest dobrym sposobem zagwarantowania, że kompilator pomoże Ci w wykryciu popełnionych błędów. Przykładowo, funkcja `GetAge()` może nie mieć żadnego powodu do zmiany klasy `Cat`, ale w implementacji może znajdować się poniższa linia kodu:

```
if (itsAge = 100) cout << "Hej! Masz 100 lat\n";
```

Zdefiniowanie funkcji `GetAge()` jako `const` powoduje, że kompilator zgłosi błąd w powyższej linii kodu. Celem programisty było sprawdzenie, czy wartość zmiennej `itsAge` wynosi 100, ale ten kod przez pomyłkę przypisuje wartość 100 zmiennej `itsAge`. Ponieważ przypisanie powoduje zmianę klasy — a zdefiniowano, że ta metoda nie powinna modyfikować klasy — kompilator wychwytuje błąd.

Ten rodzaj błędu może być trudny do wychwycenia poprzez samo przeglądanie kodu. Oko bardzo często widzi to, co chce zobaczyć. Co ważniejsze, może wydawać się, że program działa prawidłowo, choć zmiennej `itsAge` przypisano niewłaściwą wartość. Wcześniej czy później to musi spowodować problemy.

Pytanie: Czy istnieje jakikolwiek powód używania struktury w programie C++?

Odpowiedź: Wielu programistów C++ rezerwuje słowo kluczowe `struct` dla klas pozbawionych funkcji. To relikt starych struktur C, które nie miały funkcji. Szczerze mówiąc, to myląca i kiepska praktyka programistyczna. Dzisiejsza struktura pozbawiona metod jutro może ich potrzebować. Wówczas będziesz zmuszony albo do zmiany na typ `class`, albo do złamania reguły i umieszczenia metod w strukturze. Jeżeli musisz wywołać starą funkcję języka C wymagającą określonej struktury, jest to jedyny dobry powód jej używania.

Pytanie: Niektórzy programiści pracujący z programowaniem zorientowanym obiektowo używają terminu *tworzenie egzemplarza*. Co to oznacza?

Odpowiedź: *Tworzenie egzemplarza* to po prostu określenie procesu tworzenia obiektu na podstawie klasy. Określony obiekt zdefiniowany jako konkretny typ klasy to pojedynczy *egzemplarz* klasy.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Co to jest operator kropki i do czego służy?
2. Co powoduje zarezerwowanie pamięci — deklaracja czy definicja?
3. Czy deklaracja klasy jest jej interfejsem, czy implementacją?
4. Jaka jest różnica między publicznymi i prywatnymi danymi składowymi?
5. Czy funkcja składowa może być prywatna?
6. Czy dana składowa może być publiczna?
7. Jeżeli zadeklarujesz dwa obiekty `Cat`, to czy mogą mieć różne wartości w swoich elementach składowych `itsAge`?
8. Czy deklaracje klas są kończone średnikiem? A definicje metod klasy?
9. Jaki będzie nagłówek funkcji `Meow()` klasy `Cat`, jeśli nie pobiera ona parametrów i zwraca `void`?
10. Która funkcja jest wywoływana w celu inicjalizacji klasy?

Ćwiczenia

1. Napisz kod deklarujący klasę o nazwie `Employee` wraz z następującymi danymi składowymi: `itsAge`, `itsYearsOfService` oraz `itsSalary`.
2. Przepisz deklarację klasy `Employee` w taki sposób, aby dane składowe były prywatne. Zapewnij publiczne metody akcesorów pozwalające na pobieranie i ustawianie wartości każdej danej składowej.

3. Napisz program używający klasy `Employee`, który tworzy dwóch pracowników. Ustaw wartości ich danych składowych `itsAge`, `itsYearsOfService` i `itsSalary`, a następnie wyświetl je. Będziesz musiał dodać kod także w metodach akcesorów.
4. Kontynuując ćwiczenie 3., napisz kod dla metody klasy `Employee`, której celem będzie podanie zarobków każdego pracownika zaokrąglonych do najbliższego tysiąca.
5. Zmień klasę `Employee` w taki sposób, abyś mógł zainicjalizować dane składowe `itsAge`, `itsYearsOfService` oraz `itsSalary` w trakcie tworzenia pracownika.
6. **Łowcy błędów:** Co jest nie tak z poniższą deklaracją?

```
class Square
{
    public:
        int Side;
}
```

7. **Łowcy błędów:** Dlaczego poniższa deklaracja nie jest zbyt użyteczna?

```
class Cat
{
        int GetAge() const;
    private:
        int itsAge;
};
```

8. **Łowcy błędów:** Wymień trzy błędy w poniższym kodzie, które powinny być znalezione przez kompilator.

```
class  TV
{
    public:
        void SetStation(int Station);
        int GetStation() const;
    private:
        int itsStation;
};
main()
{
    TV myTV;
    myTV.itsStation = 9;
    TV.SetStation(10);
    TV myOtherTv(2);
}
```