

Pierwsza klasa

Ten rozdział wprowadza pojęcia klasy i obiektu, a także tłumaczy, w jaki sposób funkcje składowe działają na obiektach. Klasy i obiekty można projektować na różne sposoby, tworząc różne struktury — w tym rozdziale przedstawimy niektóre z nich. Napiszemy parę słów na temat języka UML. Wyjaśnimy, czym są składowe `static` i `const`. Pojawią się także konstruktory, destruktory, operacje kopiowania oraz przyjaciele.

2.1. Na początku było struct

W języku C słowo `struct` pozwala programistom na zdefiniowanie ustrukturyzowanego fragmentu pamięci, który może przechowywać heterogeniczne dane. Przykład 2.1 definiuje prostą strukturę: fragment pamięci złożony z innych, mniejszych fragmentów.

PRZYKŁAD 2.1. `src/structdemo/demostruct.h`

```
[...]
struct Fraction { // ułamek
    int numer, denomin; // licznik, mianownik
    string description; // opis
};
```

Każdy z elementów osadzonych wewnętrz `struct` jest dostępny poprzez nazwę. Elementy te nazywamy **danymi składowymi**, rzadziej — **polami**. Kod, który znajduje się poza definicją struktury, nazywamy **kodem klienckim**. Przykład 2.2 pokazuje kod kliencki korzystający ze struktury, traktujący ją jako zamkniętą całość.

Funkcja `printFraction` wyświetla poszczególne składowe we własny, oryginalny sposób. Zwróć uwagę również na to, że kod kliencki może stworzyć ułamek z niepoprawnym opisem.

PRZYKŁAD 2.2. src/structdemo/demostruct.cpp

```
[...]
void printFraction(Fraction f) {
    cout << f.numer << "/" << f.denom << endl;
    cout << " =? " << f.description << endl;
}
int main() {
    Fraction f1;
    f1.numer = 4;
    f1.denom = 5;
    f1.description = "cztery piąte";
    Fraction f2 = {2, 3, "dwie trzecie";           // /1
    f1.numer = f1.numer + 2;                      // /2
    printFraction(f1);
    printFraction(f2);
    return 0;
}
```

Wynik:

```
6/5
=? cztery piąte
2/3
=? dwie trzecie
```

- // /1 Przekazywanie struktury przez wartość może być kosztowne, jeśli jej składowe są duże.
- // /2 Inicjalizacja składowych.
- // /3 Kod kliencki może zmieniać wartość składowych.

2.2. Definicje klas

W C++ istnieje typ danych `class`, który w dużej mierze przypomina `struct`. Definicja klasy wygląda następująco:

```
class NazwaKlasy {
public:
    składowePubliczne
private:
    składowePrywatne
};
```

Pierwsza linia definicji klasy to **nagłówek klasy**.

Klasa składa się z **danych składowych** (inaczej **pól**), **funkcji składowych** (inaczej **metod**) oraz **modyfikatorów dostępu** (`public`, `private`, `protected`). Funkcje składowe służą do inicjalizacji danych składowych oraz do wykonywania na nich wszelkich innych operacji. Więcej informacji na temat funkcji znajdziesz w rozdziale 5., gdzie nacisk został położony przede wszystkim na cechy funkcji obecne w C++, ale nieobecne w innych językach. Na razie będziemy korzystać z funkcji na tyle ostrożnie, żeby ich znaczenie wynikało z kontekstu i odwoływało się do Twojego doświadczenia z innymi językami.

Po zdefiniowaniu klasy możesz używać jej nazwy jako **typu** zmiennych, parametrów i wartości zwracanych przez funkcje. Zmienne, których typ jest klasą, nazywamy **obiektami** lub **instancjami** tej klasy.

Funkcje składowe klasy opisują *zachowanie* wszystkich obiektów tej klasy. Każda funkcja składowa ma dostęp do wszystkich składowych klasy. Funkcje spoza klasy mogą uzyskać dostęp do danych składowych jedynie za pośrednictwem funkcji składowych.

Zbiór wartości danych składowych wewnątrz obiektu nazywamy *stanem* obiektu.

2.2.1. Pliki nagłówkowe

Definiując klasę (lub dowolny inny typ), powinieneś umieścić jej definicję w **pliku nagłówkowym**, najlepiej o nazwie takiej samej jak klasa i rozszerzeniu *.h*. Przykład 2.3 prezentuje plik nagłówkowy zawierający definicję klasy.

PRZYKŁAD 2.3. src/classes/fraction/fraction.h

```
#ifndef _FRACTION_H_
#define _FRACTION_H_

#include <QString>

class Fraction {
public:
    void set(int numerator, int denominator);
    double toDouble() const;
    QString toString() const;
private:
    int m_Numerator;
    int m_Denominator;
};

#endif
```

Pliki nagłówkowe są załączane do innych plików przez preprocesor. Aby uniknąć przypadkowego załączenia tego samego pliku nagłówkowego więcej niż raz w tym samym skompilowanym pliku, *ubierz go w makra preprocesora #ifndef...#define...#endif* („zdefiniuj, jeśli jeszcze nie jest zdefiniowane”). Więcej na temat makr znajdziesz w podrozdziale C.2.

Definicje funkcji składowych z reguły powinny znaleźć się poza definicją klasy, w osobnym **pliku implementacyjnym** o rozszerzeniu *.cpp*.

Przykład 2.4 to plik implementacyjny zawierający definicje funkcji zadeklarowanych w przykładzie 2.3.

PRZYKŁAD 2.4. src/classes/fraction/fraction.cpp

```
#include <QString>
#include "fraction.h"
```

```
void Fraction::set(int nn, int nd) {
    m_Numerator = nn;
    m_Denominator = nd;
}

double Fraction::toDouble() const {
    return 1.0 * m_Numerator / m_Denominator;
}

QString Fraction::toString() const {
    return QString("%1 / %2").arg(m_Numerator).arg(m_Denominator);
}
```

Każdy identyfikator posiada swój **zakres** (podrozdział 20.2), czyli obszar kodu, w którym jego nazwa jest znana (*widoczna*) i dostępna. W poprzednich przykładach mieliśmy do czynienia z identyfikatorami o zakresie blokowym. Zakres blokowy rozpoczyna się w linii, w której zadeklarowano identyfikator aż do końca bloku zawierającego tę deklarację. Zatem identyfikator nie był widoczny powyżej deklaracji ani poniżej końca bloku.

Nazwy składowych klasy mają **zakres klasy**. Obejmuje on całą definicję klasy, niezależnie od tego, w którym miejscu zadeklarowana została dana składowa. Obejmuje również plik implementacyjny (*.cpp*). Zdefiniowanie elementu klasy poza definicją klasy wymaga użycia **operatora zakresu** (w postaci `NazwaKlasy::`) przed nazwą elementu. Operator zakresu informuje kompilator o tym, że zakres klasy należy rozszerzyć poza definicję klasy. Rozszerzenie obejmuje kod pomiędzy symbolem `::` a nawiasem zamykającym definicję funkcji.

Przykładowo: składowe `Fraction::m_Numerator` i `Fraction::m_Denominator` są widoczne wewnętrz definicji `Fraction::toString()` i `Fraction::toDouble()`, mimo że zadeklarowano je w osobnym pliku.

Czasami niezbędne jest wyświetlenie obiektu, zapisanie go w pliku lub przesłanie siecią do innego programu. Operacje te można wykonać na wiele sposobów. Funkcja składowa `toString()` tradycyjnie zwracała串 znaków zawierający „migawkę” aktualnego stanu obiektu. Przydaje się ona podczas debugowania, wyświetlania stanu obiektu na ekranie, zapisywania obiektu, jego przesyłania i konwersji. W celu zwiększenia elastyczności możesz dodać do funkcji `toString()` jeden lub więcej parametrów pozwalających na formatowanie tekstu. Wiele klas Qt korzysta z tego rozwiązania. Na przykład `Qdate::toString()` zwraca datę w wielu różnych formatach w zależności od przekazanego jako argument obiektu `Qt::DateFormat`.

Powinieneś wyrobić sobie nawyk dodawania funkcji składowej `toString()` do pisanych przez Ciebie klas, które przechowują dane. Definicje klas *nie powinny* za to mieć funkcji składowych wyświetlających lub przekazujących wartości danych składowych (jak funkcja do wyświetlania `display()`, do drukowania `print()` czy `saveToFile()` i `readFromFile()` do zapisywania w pliku i czytania z niego)¹.

¹ Serializacja danych w oparciu o strumienie wejściowe i wyjściowe zostanie umówiona w sekcji 7.4.1.

2.3. Modyfikatory dostępu do składowych

Do tej pory zajmowaliśmy się **kodem definiującym klasę**, przechowywanym w dwóch miejscach w plikach nagłówkowych z definicją klasy i innymi deklaracjami, oraz **kodem implementującym klasę**, przechowywanym w odpowiednich plikach *.cpp* zawierających definicje, dla których nie stało miejsca w pliku nagłówkowym. Jest jeszcze trzecia kategoria kodu. Kod kliencki to kod poza zakresem klasy, jednak odwołujący się do obiektów lub składowych danej klasy. **Kod kliencki** z reguły załącza plik nagłówkowy zawierający definicję klasy. Przykład 2.5 pokazuje plik *fraction.h*, który zawiera kod definiujący klasę *Fraction*, ale jednocześnie będący klientem klasy *QString*.

PRZYKŁAD 2.5. src/classes/fraction/fraction.h

```
#ifndef _FRACTION_H_
#define _FRACTION_H_

#include <QString>

class Fraction {
public:
    void set(int numerator, int denominator);
    double toDouble() const;
    QString toString() const;
private:
    int m_Numerator;
    int m_Denominator;
};

#endif
```

Modyfikatory dostępu, czyli słowa *public* (publiczny), *protected* (chroniony) i *private* (prywatny) są wykorzystywane w definicji klasy do określenia, z których miejsc programu można odwołać się do danej składowej. Poniższa lista zawiera przybliżone definicje tych trzech terminów. Uszczegółowienia znajdziesz w przypisach.

- Do składowej *public* można odwołać się (korzystając z obiektu tej klasy)² wszędzie tam, gdzie załączony został plik z definicją klasy.
- Do składowej *protected* można odwołać się wewnątrz definicji funkcji składowej tej samej klasy lub wewnątrz funkcji składowej klasy *pochodnej*³.
- Składowa prywatna jest dostępna jedynie dla funkcji składowych tej samej klasy⁴.

² Wyjątkiem są tutaj składowe *public static*, do których można odwołać się bez obiektu. Zostaną one omówione w podrozdziale 2.9.

³ Pojęcie klasy pochodnej zostanie zdefiniowane w rozdziale 6., poświęconym dziedziczeniu i polymorfizmowi.

⁴ Do składowych prywatnych odwołać się mogą również przyjaciele klasy, o których więcej dowiesz się z podrozdziału 2.6.

Domyślną opcją jest dostęp prywatny. Jeśli nie określisz zasad dostępu do składowej, będzie ona prywatna.

Dostępność a widoczność

Istnieje pewna subtelna różnicę pomiędzy **dostępnością** a **widocznością**. Element nazwany jest widoczny w całym swoim zakresie. Dostępność wiąże się ze składowymi klas. Jeśli nazwana składowa klasy ma być dostępna, najpierw musi być widoczna. Jednak nie wszystkie elementy widoczne są dostępne. Dostępność zależy od modyfikatorów dostępu `public`, `private` i `protected`.

Celem kodu klienckiego w przykładzie 2.6 jest zademonstrowanie różnego rodzaju błędów dostępności i widoczności. Przykład koncentruje się na *zakresie blokowym*. Zmienna (ale nie składowa klasy) zadeklarowana wewnątrz bloku jest widoczna i dostępna jedynie pomiędzy swoją deklaracją a klamrą zamkającą blok. W przypadku funkcji blok zawierający definicję funkcji „zna” także listę parametrów.

PRZYKŁAD 2.6. src/classes/fraction/fraction-client.cpp

```
#include <QTextStream>
#include "fraction.h"

int main() {
    const int DASHES = 30;
    QTextStream cout(stdout);

    {
        int i;
        for (i = 0; i < DASHES; ++i)
            cout << "=";
        cout << endl;
    }

    cout << "i = " << i << endl; //1
    Fraction f1, f2;
    f1.set(3, 4); //2
    f2.set(11,12);
    f2.m_Numerator = 12; //3
    cout << "Pierwszy ułamek to: " << f1.toString() << endl;
    cout << "\nDrugi ułamek, wyrażony jako wartość double, to: "
        << f2.toDouble() << endl;
    return 0;
} //4
```

//1 Zakres zagnieżdżony, blok wewnętrzny.

//2 Błąd: i już nie istnieje, więc nie jest widoczne w tym zakresie.

//3 Ustawienie wartości przy użyciu funkcji składowej.

//4 Błąd, `m_Numerator` jest widoczny, ale niedostępny.

Związek pomiędzy `struct` a `class` w C++ jest prosty do zrozumienia. Stroustrup definiuje `struct` jako klasę, której składowe są domyślnie publiczne. Zapis:

```
struct T { ...  
odpowiada  
class T {public: ...
```

W C++ struct może zawierać i dane, i funkcje składowe. Programiści C++ preferują klasy, być może z powodu domyślnej prywatności danych. Struktury najczęściej stosuje się tam, gdzie potrzebne jest grupowanie danych, ale są wymagane funkcje składowe.

2.4. Enkapsulacja

Enkapsulacja (w szerszym rozumieniu: hermetyzacja) stanowi pierwszy krok na drodze do programowania obiektowego. Obejmuje ona:

- opakowywanie danych w funkcje, które wykonują na nich operacje w ramach dobrze nazwanych klas;
- tworzenie dobrze udokumentowanych i zrozumiale nazwanych publicznych funkcji pozwalających użytkownikom klasy na wykonanie wszystkich operacji, których można po klasie oczekwać;
- ukrywanie szczegółów implementacyjnych.

Zbiór publicznych prototypów funkcji danej klasy jest nazywany jej **publicznym interfejsem**. Zbiór niepublicznych funkcji i pól składowych jest nazywany **implementacją**.

Jedną z zalet enkapsulacji jest to, że umożliwia ona programistę stosowanie spójnego schematu nazewnictwa składowych. Przykładowo: istnieje wiele różnych klas, w których miałoby sens dodanie pola przechowującego koszt instancji klasy lub — jak wspomnieliśmy wcześniej — funkcji składowej o nazwie `toString()`. Ponieważ nazwy składowych nie są widoczne poza zakresem klasy, możesz bezpiecznie przyjąć konwencję stosowania tych samych nazw `m_unitCost` (koszt jednostkowy) i `toString()` we wszystkich klasach, które potrzebują tych elementów⁵.

2.5. Wprowadzenie do UML

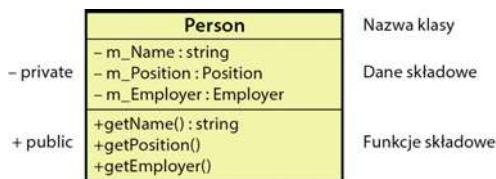
Współczesne aplikacje obiektowe opierają się na solidnej podstawie dobrze zaprojektowanych klas. W większości projektów część klas pochodzi z bibliotek ogólnego przeznaczenia (jak Qt) i bibliotek przeznaczonych do realizacji określonych zadań. Resztę klas musi napisać programista. Język UML (z ang. *Unified Modeling Language*, zunifikowany język modelowania) jest najczęściej stosowanym narzędziem projektowania obiektowego. Umożliwia on opisywanie projektów za pomocą wielu rozbudowanych typów diagramów. Używamy

⁵ Kompilator się nie pogubi, ponieważ osadza on nazwę klasy w pełnej nazwie składowej.

diagramów, ponieważ obraz jest wart tysiąca słów. Na zwięzłym, intuicyjnym diagramie można pokazać najważniejsze klasy i elementy oraz relacje pomiędzy nimi. Istnieją specjalne diagramy do opisu współpracy pomiędzy klasami, a także interakcji, w jakie użytkownicy wchodzą z systemem. W tej książce będziemy korzystali jedynie z niewielkiego podzbioru języka UML.

Większość prezentowanych tu diagramów została wykonana za pomocą narzędzia o nazwie Umbrello⁶. W menu pomocy tej aplikacji znajdziesz podręcznik „The Umbrello Modeller Handbook”, który stanowi zwięzły przegląd UML. Bardziej kompletną i zdecydowanie wartą polecenia pozycją na temat UML, zapewniającą maksimum treści przy możliwie minimalnej objętości jest [Fowler04].

Rysunek 2.1 przedstawia diagram klas z jedną tylko klasą: Person (osoba). Zwróć uwagę, że deklaracje podaje się w kolejności znanej z Pascal'a, czyli *nazwa : typ*, a nie w prawdopodobnie bardziej znajomej składni C++/Java. Ma to ułatwić czytanie diagramów. Ponieważ czytamy od lewej do prawej, w ten sposób szybciej widzimy nazwy składowych. Zwróć uwagę na to, że składowe publiczne poprzedzamy znakiem plus (+), a prywatne znakiem minus (-).

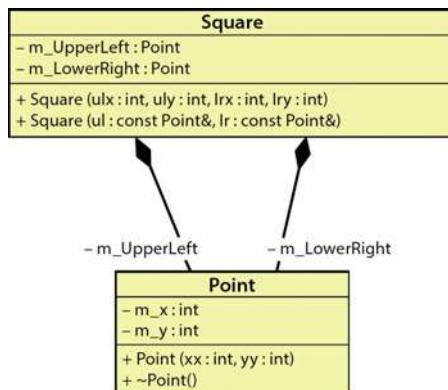


RYSUNEK 2.1. Klasa Person

2.5.1. Relacje w UML

UML wyjątkowo dobrze nadaje się do opisywania relacji pomiędzy klasami. Kolejny diagram (związany z nieco bardziej odległym przykładem 2.22) pokazuje bardzo ważną relację. Widac na nim klasę Point reprezentującą punkt na płaszczyźnie geometrycznej (ekranie) i klasę Square reprezentującą kwadrat (ekran). Klasa Point posiada dwa pola typu int, a Square dwa pola typu Point. Składowe Point są **podobiektemi** Square. Obiekt Square jest tu w pewnym sensie **rodzicem** dla podobiektów Square. Gdy obiekt Square zostanie usunięty, usunięte zostaną również jego podobiekty. Oznacza to, że podobiekty te są **komponentami** rodzica, a sama relacja jest nazywana **kompozycją**. Wypełnione romby na rysunku 2.2 oznaczają właśnie to, że mamy do czynienia z relacją kompozycji. Instancje jednej z klas są złożone bądź *skomponowane* (przynajmniej częściowo) z instancjami klasy po drugiej stronie strzałki.

⁶ <http://uml.sourceforge.net>



RYSUNEK 2.2. Kompozycja

2.6. Przyjaciele klasy

Skoro znasz już reguły dostępności, pora dowiedzieć się, jak czasami je łamać. Mechanizm **przyjaciół** pozwala na to, by funkcje, które nie są funkcjami składowymi danej klasy, miały dostęp do jej prywatnych danych. Słowo kluczowe **friend** (przyjaciel) poprzedza nazwę klasy lub funkcji. Deklaracje te pojawiają się wewnątrz definicji klasy. Poniżej kilka deklaracji przyjaciół:

```

class Timer {
    friend class Clock;
    friend void Time::toString();
    friend ostream& operator <<(ostream& os, const Timer& obj);
    [...]
private:
    long m_Elapsed;
};
  
```

Przyjaciel może być klasą, funkcją składową innej klasy, zwykłą funkcją. W poprzedzającym przykładzie przyjacielem jest `Clock`, zatem funkcje składowe tej klasy mają dostęp do składowej `Timer::m_Elapsed`. Funkcja składowa `Time::toString` jest przyjacielem `Timer`. Kompilator uważa ją za poprawną składową klasy `Time`. Trzeci przyjaciel to funkcja niebędąca niczymą składową — przeładowany **operator wstawienia**, który wstawia swój drugi argument do strumienia wyjściowego i zwraca referencję do tego strumienia, dzięki czemu wywołania można łączyć w łańcuchy.

Łamanie zasad enkapsulacji może utrudnić utrzymywanie programów, więc używaj mechanizmu przyjaciół rzadko i z dużą ostrożnością. Funkcji **friend** najczęściej używa się w dwóch celach:

1. w obiektach fabrykach w celu wymuszenia reguł wytwarzczych (podrozdział 16.1);
2. w globalnych funkcjach-operatorach, takich jak `operator<<()` i `operator>>()`, gdy nie chcesz, by operator był funkcją składową, lub nie możesz dopisać kodu do definicji istniejącej klasy.

2.7. Konstruktory

Konstruktor to specjalna funkcja składowa kontrolująca proces inicjalizacji obiektu. Konstruktor musi mieć nazwę taką samą jak nazwa klasy. Konstruktor nic nie zwraca i nie ma typu zwracanego.

Konstruktory definiuje się przy użyciu specjalnej składni:

```
NazwaKlasy::NazwaKlasy( listaParametrow )
: listaInicjalizacyjna
{
    ciałoKonstruktora
}
```

//1 Element opcjonalny, ale bardzo istotny. Nie pomijaj go, nawet jeśli kompilator nie narzeka.

Pomiędzy nawiasem zamykającym listę parametrów a klapką otwierającą ciało funkcji można podać opcjonalną listę inicjalizacyjną. **Lista inicjalizacyjna składowych** rozpoznaje się dwukropkiem (:), po którym następuje rozdzielona przecinkami lista inicjalizatorów postaci:

nazwaSkładowej(wyrażenieInicjalizujące)

Wtedy (i tylko wtedy) gdy definicja klasy nie zawiera konstruktora, kompilator podstawi następujący konstruktor:

```
NazwaKlasy::NazwaKlasy()
{}
```

Konstruktor, który można wywołać bez argumentów, to **konstruktor domyślny**. Konstruktor domyślny przeprowadza **domyślną inicjalizację** obiektu swojej klasy. Każde pole, które nie zostanie jawnie zainicjalizowane na liście inicjalizacyjnej konstruktora, otrzyma wartość początkową od kompilatora.

Klasy mogą mieć wiele konstrktorów, z których każdy inicjalizuje klasę na inny (w założeniu — przydatny) sposób. Przykład 2.7 przedstawia definicję klasy z trzema konstruktormi.

PRZYKŁAD 2.7. src/ctor/complex.h

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

class Complex {
public:
    Complex(double realPart, double imPart);
    Complex(double realPart);
    Complex();
    string toString() const;
private:
    double m_R, m_I;
};
```

Przykład 2.8 to implementacja klasy i kod kliencki.

PRZYKŁAD 2.8. src/ctor/complex.cpp

```
#include "complex.h"
#include <iostream>
#include <sstream>
using namespace std;

Complex::Complex(double realPart, double imPart)
    : m_R(realPart), m_I(imPart) //1
{
    cout << "complex(" << m_R << "," << m_I << ")" << endl;
}

Complex::Complex(double realPart) :
    m_R(realPart), m_I(0) {}

Complex::Complex() : m_R(0.0), m_I(0.0) {}

string Complex::toString() const {
    ostringstream strbuf;
    strbuf << '(' << m_R << ", " << m_I << ')';
    return strbuf.str();
}

int main() {
    Complex C1;
    Complex C2(3.14);
    Complex C3(6.2, 10.23);
    cout << C1.toString() << '\t' << C2.toString()
        << C3.toString() << endl;
}
```

//1 Lista inicjalizacyjna składowych.

Konstruktor domyślny tej klasy przypisuje obu polom obiektu C1 wartości domyślne. Inicjalizacja odpowiada następującej inicjalizacji pary zmiennych double:

```
double x, y;
cout << x << '\t' << y << endl;
```



PYTANIE

Jaki Twoim zdaniem będzie wynik działania tego kodu?

Co się stanie, jeśli pominiiesz listę inicjalizacyjną? Rozważ następującą definicję konstruktora:

```
Complex(double realPart, double imPart) {
    m_R = realPart;
    m_I = imPart;
}
```

Każde z pól jest inicjalizowane wartością domyślną, a następnie otrzymuje właściwą wartość. Nie wystąpił żaden błąd, ale taka inicjalizacja to marnowanie czasu procesora.

Spójrz na jeszcze jeden przykład związany z rysunkiem 2.2. Na jego potrzeby założmy, że tak jak na diagramie w definicji klasy `Point` istnieje tylko jeden konstruktor. Założymy też, że definiujemy konstruktor `Square` bez listy inicjalizacyjnej:

```
Square::Square(const Point& ul, const Point& lr) {  
    m_UpperLeft = ul;  
    m_LowerRight = lr;  
}
```

Ponieważ *nie zdefiniowaliśmy* w sposób jawnego domyślnego konstruktora klasy `Point`, ale zdefiniowaliśmy konstruktor dwuparametryowy, klasa `Point` nie ma konstruktora domyślnego. W konsekwencji kompilator zgłosi błąd w konstruktorze klasy `Square`.

2.8. Destruktory

Destruktor to specjalna funkcja składowa, która automatyzuje operacje związane ze sprzątaniem po obiekcie *na moment przed tym, jak jest on niszczyony*.



Kiedy obiekt jest niszczyony?

- Gdy lokalny (automatyczny) obiekt wyjdzie z zakresu (np. po wyjściu z funkcji).
- Gdy obiekt stworzony przy użyciu operatora `new` jest usuwany przy użyciu operatora `delete`.
- Tuż przed zakończeniem działania programu usuwane są wszystkie obiekty z pamięcią statyczną.

Nazwa destruktora to nazwa klasy poprzedzona tyldą (~). Nie ma on typu zwracanego i nie pobiera parametrów. Z tego powodu klasa może mieć tylko jeden destruktor. Jeśli definicja klasy nie zawiera definicji destruktora, kompilator podstawi następujący destruktor:

```
NazwaKlasy::~NazwaKlasy()  
{ }
```

W podrozdziale 2.9 pojawi się mniej trywialny przykład destruktora.



Kiedy trzeba napisać destruktor?

W ogólności każda klasa, która bezpośrednio zarządza zewnętrznym zasobem lub go współdzieli (otwiera plik, otwiera połączenie sieciowe, tworzy proces itp.), musi w odpowiednim momencie ten zasób zwolnić. Często tworzy się klasy opakowujące, które odpowiadają za posprzątanie po obiekcie.

Kontenery w Qt pozwalają uniknąć pisania kodu, który bezpośrednio zarządza pamięcią dynamiczną.

Nie potrzebujesz destruktora, jeżeli klasa:

- ma pola typów prostych niebędące wskaźnikami;
- ma pola, w których poprawnie zdefiniowano destruktory;

- jest klasą z Qt, spełniającą pewne warunki⁷.

Domyślny destruktor wygenerowany przez kompilator wywołuje — tuż przed usunięciem obiektu — destruktory wszystkich składowych w kolejności, w jakiej zostały one wymienione w definicji klasy. Domyślny destruktor nie zwalnia pamięci w przypadku składowych będących wskaźnikami.

2.9. Słowo kluczowe static

Słowo kluczowe **static** można stosować do zmiennych lokalnych, składowych klas oraz zmiennych i funkcji globalnych. W każdym przypadku **static** oznacza coś innego.

Statyczne zmienne lokalne

Słowo kluczowe **static** można zastosować podczas deklaracji zmiennej lokalnej, by przydzielić zmiennej **statyczną klasę pamięci** (podrozdział 20.3).

Taka zmienna jest tworzona tylko raz. Jest inicjalizowana, gdy program po raz pierwszy natknie się na jej deklarację. Jest niszczona przy wyłączeniu programu. Nielokalna zmienna **static** jest tworzona raz, gdy moduł obiektu jest ładowany do pamięci, a usuwana jest podczas wyłączenia programu.

Statyczne składowe klas

Pole statyczne to fragment danych powiązanych bardziej z klasą niż z konkretnym obiektem. Nie ma ono wpływu na wynik `sizeof()` na obiekcie klasy. Każdy obiekt klasy utrzymuje własny zestaw pól niestatycznych, jednak w przypadku pól statycznych istnieje tylko jedna instancja, wspólnie dzielona przez wszystkie obiekty danej klasy.

Skadowe statyczne są preferowane ponad zmienne globalne (i z powodzeniem mogą je zastąpić), ponieważ nie dodają niepotrzebnych nazw do przestrzeni globalnej.



Zanieczyszczanie globalnej przestrzeni nazw

Dodawanie nazw do zakresu globalnego (np. poprzez deklarowanie zmiennych i funkcji globalnych) jest określane mianem **zanieczyszczania globalnej przestrzeni nazw** i jest uważane za złą praktykę. Istnieje wiele istotnych powodów, by unikać deklarowania zmiennych globalnych. Jeden z nich to zwiększone ryzyko wystąpienia kolizji nazw i związanych z tym nieporozumień. Niektórzy eksperci twierdzą, że liczba zmiennych globalnych w programie jest odwrotnie proporcjonalna do jego jakości (im więcej zmiennych globalnych, tym gorsza jakość).

Statyczne składowe muszą zostać zadeklarowane jako **static** w definicji klasy (i tylko tam, zobacz przykład 2.9).

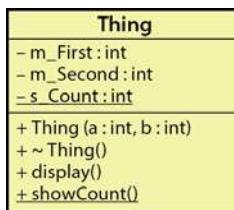
⁷ Klasa z tej grupy zostaną szczegółowo omówione, począwszy od rozdziału 8.

PRZYKŁAD 2.9. src/statics/static.h

```
[...]
class Thing {
public:
    Thing(int a, int b);
    ~Thing();
    void display() const;
    static void showCount();
private:
    int m_First, m_Second;
    static int s_Count;
};

[...]
```

Rysunek 2.3 pokazuje na diagramie klas UML diagram klasy `Thing` z przykładu 2.9.



RYSUNEK 2.3. Diagram klas UML zawierający składowe statyczne

Zwróć uwagę na to, że składowe statyczne na diagramie są podkreślone.

Funkcja składowa, która w żaden sposób nie odwołuje się do niestatycznych danych składowych, może (i powinna) zostać zadeklarowana jako `static`. W przykładzie 2.9 statyczne pole to prywatny licznik, który pamięta liczbę obiektów `Thing` istniejących w danej chwili. Funkcja składowa `public static` wyświetla aktualną wartość tego statycznego licznika.

Każda składowa statyczna musi zostać zainicjalizowana (zdefiniowana) raz *poza* definicją klasy — najlepiej w odpowiednim pliku implementacyjnym⁸. Przykład 2.10 pokazuje, jak inicjalizować składowe statyczne i korzystać z nich.

PRZYKŁAD 2.10. src/statics/static.cpp

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

int Thing::s_Count = 0; //1

Thing::Thing(int a, int b)
    : m_First(a), m_Second(b) {
    ++s_Count;
}

Thing::~Thing() {
    --s_Count;
}
```

⁸ Wyjątkiem od tej reguły jest `static const int`, które można inicjalizować wewnątrz definicji klasy.

```

void Thing::display() const {
    using namespace std;
    cout << m_First << " $$ " << m_Second;
}

void Thing::showCount() {                                ///2
    using namespace std;
    cout << "Count = " << s_Count << endl;
}

```

///1 Inicjalizacja składowych statycznych jest obowiązkowa!

///2 Funkcja statyczna.



UWAGA

Zwróć uwagę na to, że słowo `static` nie pojawia się w definicjach `s_Count` i `showCount()`. W definicji `s_Count` słowo `static` znaczyłoby coś odmiennego: zmieniłby się zakres zmiennej (z globalnej na zakres pliku, więcej w rozdziale 20.2). W przypadku definicji funkcji byłoby to nadmiarowe.

Static o zakresie blokowym

Zmienne `static` definiowane wewnątrz funkcji lub bloku kodu są inicjalizowane podczas pierwszego wykonania.

```

long nextNumber() {
    int localvar(24);
    static long statNum = 1000;
    cout << statNum + localvar;
    return ++statNum;
}

```

Pierwsze wywołanie `nextNumber()` inicjalizuje `localvar` wartością 24, a `statNum` wartością 1000, wyświetla na ekranie 1024 i zwraca 1001. Po zwróceniu wartości `localvar` jest niszczona, ale `statNum` nie. Przy każdym wywołaniu funkcji `localvar` jest tworzone i inicjalizowane z powrotem na 24. Statyczna zmienna `statNum` istnieje również pomiędzy wywołaniami, przechowując wartość, którą uzyskała podczas ostatniego wywołania. Zatem przy kolejnym wywołaniu wyświetlona zostanie wartość 1025, a zwrócona 1002.

Inicjalizacja obiektów statycznych

Obiekt `static`, który nie został zdefiniowany w bloku ani w funkcji, jest inicjalizowany, gdy po raz pierwszy ładuje się związany z nim moduł obiektowy⁹. Najczęściej ma to miejsce przy uruchomieniu programu, zanim rozpoczęcie się wykonywanie funkcji `main()`. Kolejność ładowania modułów i inicjalizacji zmiennych zależy od implementacji, dlatego nigdy nie polegaj na inicjalizacji wartości z innego pliku, nawet jeśli podczas komplikacji został on wymieniony jako pierwszy.

⁹ Więcej na ten temat w rozdziale 7., poświęconym bibliotekom i wzorcom projektowym.

Obiekt static jest konstruowany tylko raz i istnieje aż do końca działania programu. Statyczne pole to obiekt statyczny z zakresem klasy.

W przykładzie 2.11 korzystamy z bloku wewnętrznego w celu wprowadzenia pewnych lokalnych obiektów, które zostaną usunięte przed zakończeniem programu.

PRZYKŁAD 2.11. src/statics/static-test.cpp

```
#include "static.h"

int main() {
    Thing::showCount();                                ///1
    Thing t1(3,4), t2(5,6);
    t1.showCount();                                    ///2
    {
        Thing t3(7,8), t4(9,10);
        Thing::showCount();                            ///3
    }
    Thing::showCount();                                ///4
    return 0;                                         ///5
}
```

///1 W tej chwili nie istnieją żadne obiekty, ale wszystkie elementy statyczne zostały już zainicjalizowane.

///2 Dostęp poprzez obiekt.

///3 Wejście do wewnętrznego bloku kodu.

///4 Dostęp poprzez klasowy operator zakresu.

///5 Koniec bloku wewnętrznego.

Wynik komplikacji i uruchomienia programu:

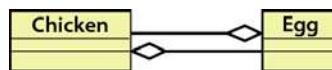
```
src/statics> g++ -Wall static.cpp static-test.cpp
src/statics> ./a.out
Count = 0
Count = 2
Count = 4
Count = 2
src/statics>
```

Statyczne zmienne globalne

Słowo static zastosowane do zmiennych i funkcji globalnych działa inaczej, niż można by się spodziewać. Nie zmienia klasy pamięci, tylko wskazuje konsolidatorowi, żeby nie eksportował symbolu na pozostałą część programu. W ten sposób symbol uzyskuje **zakres plikowy**, o którym więcej napisano w sekcji 20.2.

2.10. Deklaracje i definicje klas

Relacje dwukierunkowe, podobne do tej z rysunku 2.4, nie są niczym nadzwyczajnym w przypadku klas. By je zaimplementować, kompilator musi mieć jakąś wiedzę na temat każdej z klas przed zdefiniowaniem klasy po drugiej stronie relacji. Być może przyszło Ci do głowy, że dobrym rozwiążaniem byłoby wzajemne załączenie plików nagłówkowych obu klas, jak w przykładzie 2.12.



RYSUNEK 2.4. Relacja dwukierunkowa (Chicken — kura, Egg — jajko)

PRZYKŁAD 2.12. src/circular/badegg/egg.h

```
[...]
#include "chicken.h"
class Egg {
public:
    Chicken* getParent();
};

[...]
```

Problem uwidocznii się, gdy spojrzysz na przykład 2.13, który w analogiczny sposób załącza *egg.h*.

PRZYKŁAD 2.13. src/circular/badegg/chicken.h

```
[...]
#include "egg.h"

class Chicken {
public:
    Egg* layEgg();
};

[...]
```

Preprocesor nie pozwala na tego typu cykliczne zależności. W tym przykładzie żaden z plików nie musi załączać drugiego. W obu przypadkach niepotrzebnie stworzyliśmy silną zależność pomiędzy plikami nagłówkowymi.

Pod odpowiednimi warunkami C++ pozwala na stosowanie **zapowiedzi klasy** zamiast załączania pliku nagłówkowego (przykład 2.14).

PRZYKŁAD 2.14. src/circular/goodegg/egg.h

```
[...]
class Chicken;
class Egg {
public:
```

///1

```
    Chicken* getParent(); //2
}; [...]
```

//1 Zapowiedź klasy.

//2 Dozwolone w deklaracjach, jeśli mamy do czynienia ze wskaźnikami.

Zapowiedź klasy pozwala na odwołanie się do symbolu, nawet jeśli nie jest dostępna jego pełna definicja. To pewna obietnica składana kompilatorowi, że definicja klasy zostanie załączona wtedy, gdy naprawdę będzie potrzebna. Klassy, które zostały zadeklarowane, ale nie zdefiniowane, mogą służyć za typy wskaźników lub referencji, o ile w danym pliku nie zostaną wyłuskanie.

W module źródłowym *egg.cpp* definiujemy funkcję *getParent()*, pokazaną jako przykład 2.15. Zwróć uwagę na to, że plik *.cpp* może załączyć oba pliki nagłówkowe bez tworzenia zależności cyklicznej. Plik *.cpp* jest silnie uzależniony od obu plików nagłówkowych, ale pliki nagłówkowe nie są zależne od siebie.

PRZYKŁAD 2.15. *src/circular/goodegg/egg.cpp*

```
#include "chicken.h"
#include "egg.h"

Chicken* Egg::getParent() {
    return new Chicken(); //1
}
```

//1 Wymaga definicji *Chicken*

Podsumowując: zapowiedzi klas umożliwiają definiowanie relacji dwukierunkowych, takich jak w przykładzie 2.15, bez tworzenia cyklicznych zależności. Umieściliśmy zależności w modułach źródłowych, gdzie rzeczywiście były potrzebne, a nie w plikach nagłówkowych.

W Javie jest możliwe tworzenie silnych zależności cyklicznych pomiędzy (lub więcej) klasami. Innymi słowy, każda klasa może importować i używać (wyłuskiwać referencje do) wszystkich innych. Takie cykliczne referencje utrudniają utrzymywanie kodu, ponieważ zmiana w jednej klasie może zepsuć wszystkie inne. Jest to jedna z nielicznych sytuacji, w których C++ chroni Cię lepiej niż Java: nie da się wprowadzić takiej relacji przypadkowo.

W Javie również istnieje mechanizm zapowiedzi klas, jednak jest rzadko stosowany, ponieważ programy w tym języku nie wyróżniają plików nagłówkowych i implementacyjnych.

Więcej informacji znajdziesz w podrozdziale C.2.

2.11. Konstruktory kopiące i operatory przypisania

C++ daje projektantowi klasy niemalże boską władzę. „Zarządzanie cyklem życia obiektu” oznacza sprawowanie całkowitej kontroli nad zachowaniem obiektów w czasie ich powstawania, reprodukowania oraz śmierci. Widziałeś już, jak konstruktory kontrolują narodziny obiektu, a destruktory jego śmierć. W tej sekcji zajmiemy się procesem reprodukcji: wykorzystaniem konstruktorów kopiących i operatorów przypisania.

Konstruktor kopiący to konstruktor o następującym prototypie:

NazwaKlasy(const *NazwaKlasy* & x)

Celem konstruktora kopiącego jest stworzenie obiektu będącego dokładną kopią istniejącego już obiektu danej klasy.

Operator przypisania dla danej klasy przeładowuje symbol =, nadając mu znaczenie właściwe danej klasie. Istnieje szczególna wersja operatora przypisania o następującym prototypie:

NazwaKlasy & operator=(const *NazwaKlasy* & x);

Ponieważ możliwe jest istnienie kilku przeładowanych wersji operator=() wewnętrz klasy, tę konkretną wersję będziemy nazywać **kopiującym operatorem przypisania**.

Wersja Fraction z przykładu 2.16 ma trzy liczniki statyczne, zdefiniowane w przykładzie 2.17. Pozwalają one na zliczenie, ile razy wywołane zostały poszczególne funkcje składowe. Te fragmenty kodu powinny ułatwić Ci zrozumienie, kiedy obiekty są kopiowane.

PRZYKŁAD 2.16. src/lifecycle/copyassing/fraction.h

```
[...]
class Fraction {
public:
    Fraction(int n, int d);                                ///1
    Fraction(const Fraction& other);                      ///2
    Fraction& operator=(const Fraction& other);          ///3
    Fraction multiply(Fraction f2);
    static QString report();
private:
    int m_Numer, m_Denom;
    static int s_assigns;
    static int s_copies;
    static int s_ctors;
};

[...]
```

///1 Zwykły konstruktor.

///2 Konstruktor kopiący.

///3 Kopiący operator przypisania.

PRZYKŁAD 2.17. src/lifecycle/copyassign/fraction.cpp

```
[...]
int Fraction::s_assigns = 0;                                ///1
int Fraction::s_copies = 0;
int Fraction::s_ctors = 0;

Fraction::Fraction(const Fraction& other)
    : m_Numer(other.m_Numer), m_Denom(other.m_Denom) {
    ++s_copies;
}

Fraction& Fraction::operator=(const Fraction& other) {
    if (this != &other) {                                       ///2
        m_Numer = other.m_Numer;
        m_Denom = other.m_Denom;
        ++s_assigns;
    }
    return *this;                                              ///3
}
[...]
```

///1 Statyczne definicje składowych.

///2 Przeładowany operator=() nie powinien wykonywać żadnej akcji podczas przypisywania wartości samej sobie.

///3 Przeładowany operator=() powinien zawsze zwracać *this, by pozwolić na łączenie wywołań w łańcuchy, np. a=b=c.

Przykład 2.18 korzysta ze zdefiniowanych przed chwilą klas do tworzenia, kopowania i przypisywania obiektów.

PRZYKŁAD 2.18. src/lifecycle/copyassign/copyassign.cpp

```
#include <QTextStream>
#include "fraction.h"

int main() {
    QTextStream cout(stdout);
    Fraction twothirds(2,3);                                ///1
    Fraction threequarters(3,4);
    Fraction acopy(twothirds);
    Fraction f4 = threequarters;                            ///2
    ///3
    cout << "po deklaracjach - " << Fraction::report();
    f4 = twothirds;                                         ///4
    cout << "\nprzed mnożeniem - " << Fraction::report();
    f4 = twothirds.multiply(threequarters);                ///5
    cout << "\npo mnożeniu - " << Fraction::report() << endl;
    return 0;
}
```

///1 Konstruktor dwuargumentowy.

///2 Konstruktor kopujący.

///3 Znów konstruktor kopujący.

///4 Przypisanie.

///5 W tym miejscu tworzonych jest kilka obiektów.

Oto wyjście z programu:

```
copyassign> ./copyassign
po deklaracji - [assigns: 0 copies: 2 ctors: 2]
przed mnożeniem - [assigns: 1 copies: 2 ctors: 2]
po mnożeniu - [assigns: 2 copies: 3 ctors: 3]
copyassign>
```



PYTANIE

Jak widzisz, wywołanie `multiply` tworzy trzy obiekty `Fraction`. Czy potrafisz wyjaśnić dlaczego?

Konstruktor kopiujący autorstwa kompilatora

Ważne, by wiedzieć, że kompilator sam z siebie przygotuje wersje konstruktora kopiującego lub kopiującego operatora przypisania, jeśli definicja klasy nie zawiera żadnego z tych elementów. Wersje stworzone przez kompilator są publiczne i mają następujące prototypy dla klasy `T`:

```
T::T(const T & other);
T& T::operator=(const T & other);
```

Obie wersje tworzą dokładną kopię każdego z pól. W przypadku klas, których wszystkie składowe są typów prostych lub przechowują proste wartości, jak `int`, `double`, `QString` itp., wersja dostarczona przez kompilator okaże się wystarczająca. Jeśli jednak klasa zawiera składowe będące wskaźnikami lub obiektami¹⁰, to istotne jest napisanie zarówno konstruktora kopiującego, jak i operatora przypisania dla tej klasy. Wkrótce¹¹ przekonasz się, że czasami należy zablokować możliwość tworzenia kopii obiektów danej klasy. W takim przypadku i konstruktor kopiujący, i operator przypisania muszą zostać zadeklarowane jako prywatne i muszą otrzymać odpowiednie definicje niekopiowalne, by kompilator nie mógł podstawić swoich wersji publicznych.

2.12. Konwersje

Konstruktor, który może zostać wywołany z jednym argumentem (innego typu) i tworzy nowy obiekt poprzez **konwersję** z typu argumentu do typu klasy, jest nazywany **konstruktorem konwertującym**. Przykład 2.19 przedstawia taki właśnie konstruktor.

PRZYKŁAD 2.19. src/ctor/conversion/fraction.cpp

```
class Fraction {
public:
    Fraction(int n) : m_Numer(n), m_Denom(1) {} ///1
    Fraction(int n, int d) : m_Numer(n), m_Denom(d) {}
```

¹⁰ Podrozdział 8.1.

¹¹ Rozdział 8.

```
Fraction times(const Fraction& other) {
    return Fraction(m_Numer * other.m_Numer, m_Denom* other.m_Denom);
}
private:
    int m_Numer, m_Denom;
};
int main() {
    int i;
    Fraction frac(8); //1
    Fraction frac2 = 5; //2
    frac = 9; //3
    frac = (Fraction) 7; //4
    frac = Fraction(6); //5
    frac = static_cast<Fraction>(6); //6
    frac = frac2.times(19); //7
    return 0;
} //8
```

- //1 Konstruktor jednoargumentowy definiuje konwersję z `int`.
- //2 Wywołanie konstruktora konwertującego.
- //3 Kopiowanie `int` (również wywołuje konstruktor konwertujący).
- //4 Konwersja z przypisaniem.
- //5 Rzutowanie w stylu C (przestarzałe).
- //6 Jawny element pośredni; rzutowanie w stylu C++.
- //7 Preferowane rzutowanie w stylu ANSI.
- //8 Jawne wywołanie konstruktora konwertującego.

W funkcji `main()` w przykładzie 2.19 zmienna `frac` jest inicjalizowana za pomocą wartości `int`. Pasuje do niej konstruktor jednoargumentowy. Zamieni on liczbę całkowitą 8 na ułamek 8/1.

Prototyp konstruktora konwertującego najczęściej wygląda tak:

```
ClassA::ClassA(const ClassB& bobj);
```

Konstruktor konwertujący klasy `ClassA` zostanie wywołany automatycznie, gdy niezbędny okaże się obiekt tej klasy i gdy istnieje możliwość utworzenia odpowiedniej wartości przez konstruktor w oparciu o klasę `ClassB`, której obiekt został przekazany jako wartość inicjalizacyjna lub przez przypisanie.

Przykładowo: jeśli `frac` to odpowiednio zainicjalizowany obiekt `Fraction` zdefiniowany jak w przykładzie 2.19, to możesz wydać następującą instrukcję:

```
frac = frac.times(19);
```

Ponieważ liczba 19 nie jest obiektem `Fraction` (czego wymaga definicja funkcji `times()`), kompilator sprawdzi, czy można tę wartość przekształcić na `Fraction`. Ponieważ mamy konstruktor konwertujący, jest to jak najbardziej możliwe.

Nie zdefiniowano przeładowanego operatora, zatem kompilator użyje domyślnego operatora przypisania:

```
Fraction& operator=(const Fraction& fobj);
```

Operator przypisania przechodzi przez wszystkie pola `fobj`, przypisując odpowiednie wartości polom obiektu gospodarza.

Zatem nasza jedna instrukcja wywołuje trzy funkcje składowe `Fraction`:

1. `Fraction::operator=()` w celu przypisania wartości.
2. `Fraction::times()` w celu mnożenia.
3. `Fraction::Fraction(19)` w celu przekształcenia wartości 19 na `Fraction`.

Tymczasowy obiekt `Fraction` zwracany przez funkcję `times()` istnieje na tyle długo, by zakończyć przypisywanie. Następnie jest automatycznie niszczony.

Móżesz uprościć definicję klasy `Fraction`, eliminując konstruktor jednoargumentowy i dostarczając domyślną wartość drugiego parametru w wersji dwuargumentowej — jak w przykładzie 2.20. Ponieważ taki konstruktor można wywołać z jednym argumentem, jest zgodny z definicją konstruktora konwertującego.

PRZYKŁAD 2.20. src/ctor/conversion/fraction.h

```
class Fraction {
public:
    Fraction(int n, int d = 1)
        : m_Numer(n), m_Denom(d) {}
    Fraction times(const Fraction& other) {
        return Fraction(m_Numer * other.m_Numer, m_Denom * other.m_Denom);
    }
private:
    int m_Numer, m_Denom;
};
```

Zasadniczo każdy konstruktor, który może zostać wywołany z jednym argumentem innego typu, jest uważany za konstruktor konwertujący. Jeśli mechanizm konwersji w danym przypadku nie ma zastosowania, można go zablokować. Słowo kluczowe `explicit` zapobiega automatycznemu wykorzystaniu danego konstruktora jako konstruktora konwertującego przez kompilator, a co za tym idzie — zapobiega niejawnym konwersjom¹².

Słowo kluczowe `explicit`

Słowo kluczowe `explicit` można umieścić po argumencie konstruktora jednoargumentowego w definicji klasy, by uniknąć stosowania tego konstruktora do automatycznej konwersji. Przydaje się to, gdy argument nie przypomina klasy, którą tworzy, lub jeśli pomiędzy obiektami istnieje relacja przypominająca relację rodzic – dziecko. Klasy generowane przez Qt Creator umieszczają to słowo przed wygenerowanymi konstruktorami widżetów użytkownika. Zalecamy to samo zachowanie w przypadku klas dziedziczących z `QWidget`.

¹² Związan z tym tematem przykład znajdziesz w sekcji 19.8.4.

2.13. Funkcje składowe const

Jeśli funkcja składowa klasy jest wywoływana poprzez obiekt objx:

```
objx.f();
```

nazywamy objx **gospodarzem**.

Słowo kluczowe **const** ma specjalne znaczenie, gdy jest stosowane do niestatycznej funkcji składowej. Jeśli jest umieszczone po liście parametrów, **const** staje się częścią sygnatury funkcji. Gwarantuje ono, że funkcja nie zmieni stanu gospodarza.

Można na to spojrzeć tak: każda niestatyczna funkcja składowa ma niejawny parametr o nazwie **this**, będący wskaźnikiem do obiektu gospodarza. Gdy deklarujesz funkcję składową jako **const**, mówisz kompilatorowi, że z punktu widzenia tej funkcji **this** to wskaźnik na **const**.

W celu zrozumienia, jak zmienia to sposób wywoływanego funkcji, przyjrzyj się, jak oryginalny preprocesor przetwarzający C++ do C radził sobie z funkcjami składowymi. Jako że C nie pozwala na przeładowywanie funkcji ani na funkcje składowe, preprocesor tłumaczył funkcję na funkcję języka C ze specjalną nazwą, która od innych funkcji różniła się tym, że jej pełna sygnatura była zakodowana w nazwie. Proces ten wprowadzał na listę parametrów dodatkowy parametr niejawny: **this**, czyli wskaźnik na obiekt gospodarza. Przykład 2.21 pokazuje, jak funkcje składowe mogą być widziane przez konsolidator po ich przetłumaczeniu na C.

PRZYKŁAD 2.21. src/const/constmembers/constmembers.cpp

```
#include <QTextStream>
#include <QString>

class Point {
public:
    Point(int px, int py)
        : m_X(px), m_Y(py) {}

    void set(int nx, int ny) { //1
        m_X = nx;
        m_Y = ny;
    }

    QString toString() const { //2
        //m_X = 5; //3
        m_Count++; //4
        return QString("[%1,%2]").arg(m_X).arg(m_Y);
    }

private:
    int m_X, m_Y; //5
    mutable int m_Count;
};

int main() {
    QTextStream cout(stdout);
```

```

Point p(1,1);
const Point q(2,2);
p.set(4,4);                                ///6
cout << p.toString() << endl;
// q.set(4,4);                                ///7
return 0;
}

```

///1 Wersja C: `_Point_set_int_int(int(Point* this, int nx, int ny)`.

///2 Wersja C: `_Point_toString_string_const(const Point* this)`.

///1 Błąd: `this->m_X = 5`, `*this` jest stałe (`const`).

///4 W porządku, składowa jest modyfikowalna.

///5 Zmienna `mutable` może zostać zmodyfikowana wewnątrz metody `const`.

///6 W porządku, można ponownie przypisać `p`.

///7 Błąd! Obiekt `const` nie może wywoływać metod `nie-const`.

W rzeczywistości zmienione nazwy `set` i `print` zostałyby skompresowane, by zaoszczędzić miejsce, a przy okazji stałyby się mniej czytelne dla człowieka. Zauważ, że **modyfikowalne** (`mutable`) składowe można zmieniać wewnątrz funkcji składowych `const`, jednak nie można zmieniać zwykłych składowych.

Słowo `const` w sygnaturze `print()` można traktować jako modyfikator niewidzialnego parametru `this` wskazującego na obiekt gospodarza. Oznacza to, że pamięci, na którą wskazuje `this`, nie można zmienić z wnętrza funkcji `print()`. Powodem, dla którego przypisanie `x = 5`; generuje błąd, jest to, że zapis ten odpowiada `this->x = 5`. Łamie to reguły `const`.

Założmy, że musisz zająć się projektem, w którym istniejące klasy w nieodpowiedni sposób korzystają z `const`. Kiedy zacznesz dodawać `const` do składowych, parametrów i wskaźników, które tego wymagają, może się okazać, że Twoje zmiany generują kaskadę błędów kompilatora uniemożliwiającą zbudowanie projektu aż do chwili, gdy poprawnie pododajesz `const` w całym projekcie.

2.14. Podobiekty

Obiekty mogą zawierać inne obiekty. W takim przypadku obiekt wewnętrzny (zawarty w innym) jest nazywany **podobiektem**. W przykładzie 2.22 każdy obiekt `Square` ma dwa podobiekty `Point`.

PRZYKŁAD 2.22. src/subobject/subobject.h

```

[...]
class Point {
public:
    Point(int xx, int yy) : m_x(xx), m_y(yy){}
    ~Point() {
        cout << "punkt usunięty: (" 
            << m_x << "," << m_y << ")" << endl;
    }
}

```

```
private:  
    int m_x, m_y;  
};  
  
class Square {  
public:  
    Square(int ulx, int uly, int lrx, int lry)  
        : m_UpperLeft(ulx, uly), m_LowerRight (lrx, lry) //1  
    {}  
  
    Square(const Point& ul, const Point& lr) :  
        m_UpperLeft(ul), m_LowerRight(lr) {} //2  
private:  
    Point m_UpperLeft, m_LowerRight; //3  
};  
[...]
```

//1 Inicjalizacja składowych jest tu wymagana, ponieważ nie ma konstruktora domyślnego.

//2 Inicjalizacja składowych przy użyciu domniemanego konstruktora kopiącego Point.

//3 Osadzone podobiekty.

Z każdym razem, gdy tworzona jest instancja `Square`, wraz z nią tworzone są instancje podobiektów, dzięki czemu trzy obiekty zajmują spójny fragment pamięci. Podczas niszczenia instancji `Square` niszczone są wszystkie jej podobiekty.

Obiekt `Square` jest złożony (*relacja kompozycji*) z dwóch obiektów `Point`. Ponieważ `Point` nie ma konstruktora domyślnego¹³, musimy poprawnie zainicjalizować podobiekty `Point` na liście inicjalizacyjnej `Square`¹⁴.

Przykład 2.23 to kod kliencki tworzący instancje interesujących nas klas.

PRZYKŁAD 2.23. src/subobject/subobject.cpp

```
#include "subobject.h"  
  
int main() {  
    Square sq1(3,4,5,6);  
    Point p1(2,3), p2(8, 9);  
    Square sq2(p1, p2);  
}
```

Mimo że nie zdefiniowaliśmy destruktora dla `Square`, każdy z podobiektów został poprawnie zniszczony niezależnie od tego, co było jego obiektem rodzicem. Jak zaobserwowaliśmy w sekcji 2.5.1, jest to przykład kompozycji, do której wróćmy jeszcze w podrozdziale 6.9.

¹³ Dlaczego nie ma?

¹⁴ Dlaczego nie możemy po prostu zainicjalizować `m_x` i `m_y`?

```
punkt usunięty: (8,9)
punkt usunięty: (2,3)
punkt usunięty: (8,9)
punkt usunięty: (2,3)
punkt usunięty: (5,6)
punkt usunięty: (3,4)
```

2.15. Ćwiczenia: klasy

- Przykłady od 2.24 do 2.26 tworzą jeden program. Wykorzystaj go do rozwiązania zdefiniowanych poniżej problemów.

PRZYKŁAD 2.24. src/early-examples/thing/thing.h

```
#ifndef THING_H_
#define THING_H_

class Thing {
public:
    void set(int num, char c);
    void increment();
    void show();
private:
    int m_Number;
    char m_Character;
};
#endif
```

PRZYKŁAD 2.25. src/early-examples/thing/thing.cpp

```
#include <QTextStream>
#include "thing.h"

void Thing::set(int num, char c) {
    m_Number = num;
    m_Character = c;
}

void Thing::increment() {
    ++m_Number;
    ++m_Character;
}

void Thing::show() {
    QTextStream cout(stdout);
    cout << m_Number << '\t' << m_Character << endl;
}
```

PRZYKŁAD 2.26. src/early-examples/thing/thing-demo.cpp

```
#include <QTextStream>
#include "thing.h"
```

```

void display(Thing t, int n) {
    int i;
    for (i = 0; i < n; ++i)
        t.show();
}

int main() {
    QTextStream cout(stdout);
    Thing t1, t2;
    t1.set(23, 'H');
    t2.set(1234, 'w');
    t1.increment();
    // cout << t1.m_Number;
    display(t1, 3);
    // cout << i << endl;
    t2.show();
    return 0;
}

```

- a. Usuń znaki komentarza z dwóch linii w kodzie przykładu 2.26 i spróbuj zbudować program za pomocą komend:

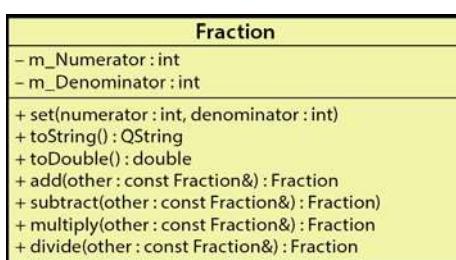
```

qmake -project
qmake
make

```

Wyjaśnij różnicę w błędach zgłaszanych przez kompilator.

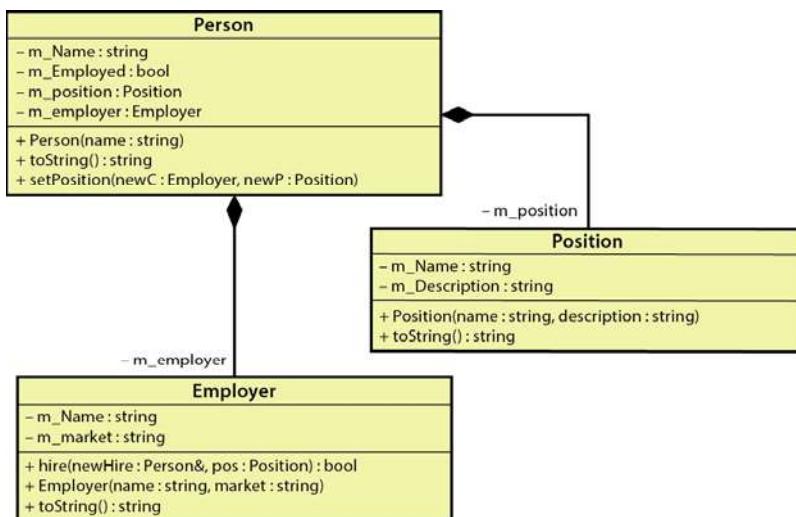
- b. Dodaj publiczne funkcje składowe do definicji klasy `Thing`, by dane składowe były prywatne, ale kod kliencki ciągle mógł wyświetlić ich wartości.
2. W oparciu o diagram UML na rysunku 2.5 zdefiniuj klasę oraz wszystkie wyspecyfikowane funkcje składowe, tworząc rozszerzoną klasę `Fraction`. Za punkt wyjścia możesz przyjąć kod przykładu 2.4.



RYSUNEK 2.5. Diagram klasy `Fraction` (ułamek) z operacjami arytmetycznymi (dodawanie, odejmowanie, mnożenie, dzielenie)

Napisz kod kliencki sprawdzający poprawność obliczeń.

3. Założmy, że piszesz aplikację dla urzędu pracy. W pierwszym kroku musisz zaprojektować odpowiednie klasy. Za punkt wyjścia przyjmij rysunek 2.6. Na diagramie klasa `Person` (osoba) ma dwie podklasy: `Employer` (pracodawca) i `Position` (stanowisko). Najważniejsza funkcja składowa to `hire` (zatrudnij)!



RYSUNEK 2.6. Struktura firmy

W tym ćwiczeniu przydadzą się zapowiedzi klasy (podrozdział 2.10).

- Napisz klasy Person, Position i Employer wyspecyfikowane na rysunku 2.6.
 - Jeśli funkcje składowe Person::getPosition() (pobierz stanowisko osoby) i Person::getEmployer (pobierz pracodawcę) nie mogą zwrócić wyniku, ponieważ osoba jeszcze nie ma pracy, niech utworzą i zwrócą coś zabawnego.
 - Niech wywołanie funkcji hire (zatrudnij) tak zmieni stan obiektu Person, by w przyszłości wywołania getPosition() i getEmployer() zwracały poprawny wynik.
 - W programie main() utwórz dwóch pracodawców (instancje Employer): Gwiezdną Flotę oraz Borg.
 - Stwórz przynajmniej dwóch pracowników. Niech będą to Jean-Luc Picard i Wesley Crusher.
 - Dla każdej klasy napisz funkcję `toString()` zwracającą tekstową reprezentację obiektu.
 - Napisz główny program, który tworzy kilka obiektów, a potem wypisuje na ekranie listę pracowników każdej z firm.
- Skrytykuj projekt widoczny na rysunku 2.6. Co jest z nim nie tak? Jak można napisać Employer::getEmployees() i Position::getEmployer()? Zaproponuj poprawki.
 - Zdefiniuj klasę reprezentującą nowoczesny samochód, opartą na diagramie z rysunku 2.7.
- Oto kilka właściwości tej klasy:
- Poza składowymi nazwanymi konstruktor powinien inicjalizować prędkość. Zero wydaje się tu rozsądny wyborem — dość dziwnie byłoby skonstruować już jadący pojazd.
 - Funkcja `drive()` powinna być dość inteligentna:

Hondurota
<ul style="list-style-type: none"> - m_Fuel : double - m_Odometer : double - m_TankCapacity : double - m MPG : double - m_Speed : double <ul style="list-style-type: none"> + addFuel(gal : double) : double + getSpeed() : double + Hondurota(fuel : double, odom : double, capacity : double, mpg : double) + getTankCapacity() : double + getMPG() : double + drive(speed : double, minutes : int) : double + getFuel() : double + getOdometer() : double

RYSUNEK 2.7. Hondurota (pola: paliwo, drogomierz, pojemność baku, spalanie, prędkość; funkcje: metody dostępowe i konstruktor [„pobierz ...”])

- samochód nie może jechać, jeśli nie ma paliwa;
- drogomierz i poziom paliwa powinny zmieniać wartości podczas jazdy;
- funkcja zwraca ilość paliwa pozostałą w baku;
- funkcja addFuel() (dolej paliwa) powinna zmieniać ilość paliwa w baku i zwracać stan po dolaniu; addFuel(0) napełnia bak.

Napisz kod kliencki testujący tę klasę.

6. W poprzednim przykładzie nie korzystaliśmy właściwie z pomiaru prędkości. Funkcja `drive()` zakładała średnią prędkość i średnie zużycie paliwa. Spróbuj teraz dodać prędkość, by trochę urealnić działanie auta.

Dodaj do klasy Hondurota funkcję składową o prototypie:

```
double highwayDrive(double distance, double speedLimit);
```

Funkcja realizuje podróż autostradą. Pobiera odległość i ograniczenie prędkości, zwraca szacowany czas podróży.

Podczas jazdy po autostradzie z reguły można poruszać się z prędkością bliską ograniczenia prędkości lub mu równą. Niestety, zdarza się, że w wyniku pewnych wydarzeń ruch odbywa się wolniej — czasami dużo wolniej.

Inny interesujący czynnik to wpływ prędkości na spalanie. W przypadku większości współczesnych samochodów wyznaczono prędkość optymalną pod względem spalania (np. 70 km/h). Rolą tej nowej funkcji jest obliczenie, ile zajmie przebycie wyznaczonej odległości po autostradzie i ile paliwa zostanie zużyte.

- Napisz funkcję, która co minutę aktualizuje prędkość, drogomierz i ilość paliwa aż do momentu pokonania wyznaczonej odległości.
- Ustaw 70 km/h jako prędkość, przy której zużywana jest dokładnie ta ilość paliwa, którą ustalono w `m_FuelConsumptionRate`.
- W przypadku innych prędkości użądź odpowiednio zmodyfikowanej funkcji spalania, dodając 1% za każdy kilometr na godzinę ponad wyznaczoną wartość 70 km/h.
- Samochód powinien się zatrzymać, gdy skończy się paliwo.

- Załóż, że podróżujesz z prędkością odpowiadającą ograniczeniu prędkości, z wyjątkiem wyznaczanych co minutę losowych odchyлеń o wartość od minus do plus 5 km/h. Samochód nie może jechać szybciej niż 40 km/h ponad limit. Z oczywistych powodów nie może jechać wolniej niż 0 km/h. Jeśli losowe odchylenie prowadzi do prędkości spoza dozwolonego przedziału, wygeneruj nową wartość.

Napisz kod kliencki testujący tę funkcję.

7. Zabaw się w komputer i spróbuj przewidzieć wynik działania kodu z przykładu 2.28 (jest to kod kliencki dla kodu z przykładu 2.27).

PRZYKŁAD 2.27. src/statics/static3.h

```
#ifndef _STATIC_3_H_
#define _STATIC_3_H_

#include <string>
using namespace std;

class Client {
public:
    Client(string name) : m_Name(name), m_ID(s_SavedID++)
    {
    }
    static int getSavedID() {
        if(s_SavedID > m_ID) return s_SavedID;
        else return 0;
    }
    string getName() {return m_Name;}
    int getID() {return m_ID;}
private:
    string m_Name;
    int m_ID;
    static int s_SavedID;
};

#endif
```

PRZYKŁAD 2.28. src/statics/static3.cpp

```
#include "static3.h"
#include <iostream>

int Client::s_SavedID(1000);

int main() {
    Client cust1("George");
    cout << cust1.getID() << endl;
    cout << Client::getName() << endl;
}
```

8. Zaprojektuj i zaimplementuj klasę Date (data) w oparciu o rysunek 2.8, stosującą się do następujących ograniczeń i wytycznych:



RYSUNEK 2.8. Diagram UML przedstawiający klasę Date

- Pole `m_DaysSinceBaseDate` przechowuje wartość całkowitą równą liczbie dni, które minęły od ustalonej daty 1 stycznia 1000 (lub dowolnej innej, wedle Twojego uznania).
- Rok będący punktem odniesienia powinien być reprezentowany przez składową `static int` (o wartości np. 1000 lub 1900).
- Klasa ma konstruktor oraz funkcję `set` (ustaw) z parametrami reprezentującymi rok, miesiąc i dzień. W oparciu o te wartości wyznaczana jest liczba dni od daty bazowej. Przekształcenie jest realizowane przez prywatną funkcję `ymd2dsbd()`, wywoływaną i przez konstruktora, i przez funkcję ustawiającą wartość.
- Funkcja `toString()` zwraca reprezentację daty w postaci tekstowej zgodnej z wybranym standardem (np. `rrrr/mm/dd`).

Wymaga to wykonania obliczeń, które wykonuje wspomniana wcześniej funkcja `ymd2dsbd()`. Realizuje ją kolejna prywatna funkcja składowa — `getYMD()`. Diagram sugeruje wersję funkcji `toString` z parametrem `brief` (skrócony), pozwalającym wybrać format daty.

- Na diagramie pojawia się także kilka statycznych funkcji pomocniczych, takich jak `leapYear` (rok przestępny). Są one oznaczone jako `static`, ponieważ nie zmieniają stanu obiektu `Date`.
- Upewnij się, że znasz reguły określania, które lata są przestępne!
- Definicję klasy umieść w pliku `date.h`.
- Stwórz plik `date.cpp` zawierający definicje wszystkich funkcji zadeklarowanych w `date.h`.
- Napisz kod kliencki dogłębnie testujący klasę `Date`.
- Klasa powinna w rozsądny sposób obsługiwać „niepoprawne” daty (np. rok wcześniejszy niż rok bazowy, niedozwolony numer miesiąca lub daty).
- Oto kod funkcji `setToToday()` korzystającej z zegara systemowego w celu ustalenia dzisiejszej daty. Do jego działania niezbędne jest załączenie `time.h` z biblioteki standardowej:

```

void Date::setToToday() {
    struct tm *tp = 0;
    time_t now;
    now = time(0);
    tp = localtime(&now);
    set(1 + tp->tm_mon, tp->tm_mday, 1900 + tp->tm_year);
}

```

- Funkcja `getWeekDay()` zwraca nazwę dnia tygodnia odpowiadającego danej dacie. Możesz ją wykorzystać w bogatszej wersji funkcji `toString()`. Wskazówka: 1 stycznia 1900 to poniedziałek.

9. Rozważ klasę z przykładu 2.29:

PRZYKŁAD 2.29. src/destructor/demo/thing.h

```

#ifndef THING_H_
#define THING_H_

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

class Thing {
public:
    Thing(int n) : m_Num(n) {

    }
    ~Thing() {
        cout << "wywołanie destruktora: "
            << m_Num << endl;
    }

private:
    string m_String;
    int m_Num;
};

#endif

```

Kod kliencki z przykładu 2.30 na różne sposoby tworzy kilka obiektów i usuwa większość z nich.

PRZYKŁAD 2.30. src/destructor/demo/destructor-demo.cpp

```

#include "thing.h"

void function(Thing t) {
    Thing lt(106);
    Thing* tp1 = new Thing(107);
    Thing* tp2 = new Thing(108);
    delete tp1;
}

int main() {
    Thing t1(101), t2(102);
    Thing* tp1 = new Thing(103);
    function(t1);
}

```

```
{ //1
    Thing t3(104);
    Thing* tp = new Thing(105);
}
delete tp1;
return 0;
}
```

//1 Blok (zakres) zagnieżdżony.

Oto wynik działania programu:

```
wywołanie destruktora: 107
wywołanie destruktora: 106
wywołanie destruktora: 101
wywołanie destruktora: 104
wywołanie destruktora: 103
wywołanie destruktora: 102
wywołanie destruktora: 101
```

- a. Ile obiektów zostało utworzonych, ale nie zniszczonych?
- b. Dlaczego 101 pojawia się na liście dwukrotnie?

2.16. Pytania sprawdzające znajomość rozdziału

1. Opisz przynajmniej jedną różnicę pomiędzy class a struct.
2. W jaki sposób zakres klasy różni się od zakresu blokowego?
3. Wymień dwie sytuacje, w których dozwolone jest korzystanie z funkcji zaprzyjaźnionych.
4. Jak pole statyczne różni się od niestatycznego?
5. Jaka jest różnica pomiędzy funkcją składową static a zwykłą funkcją składową?
6. Jakie jest znaczenie słowa const użytego podczas deklaracji funkcji składowej?
7. Wyjaśnij, co by się stało (i dlaczego), gdyby klasa T miała konstruktor kopiąjący o następującym prototypie:
`T::T(T other);`
8. Niektóre spośród linii oznaczonych w przykładzie 2.31 zawierają błędy. Przykład 2.32 zawiera pytania wielokrotnego wyboru.

PRZYKŁAD 2.31. src/quizzes/constquiz.cpp

```
#include <iostream>

class Point {
public:
    Point(int px, int py)
        : m_X(px), m_Y(py) {} //1

    void set(int nx, int ny) {
        m_X = nx;
        m_Y = ny;
    }
}
```

```

void print() const {
    using namespace std;
    cout << "[" << m_X << "," << m_Y << "]";
    m_printCount++;
} //2
private:
    int m_X, m_Y;
    int m_printCount; //3
};

int main() {
    Point p(1,1);
    const Point q(2,2);
    p.set(4,4); //4
    p.print();
    q.set(4,4); //5
    q.print(); //6
    return 0;
}

```

//1 _____
//2 _____
//3 _____
//4 _____
//5 _____
//6 _____

PRZYKŁAD 2.32. src/quizzes/constquiz-questions.txt

1. Jakie błędy wystąpią w tej linii?
 a. Niedozwolone w tym miejscu.
 b. Brakuje `m_PointCount` – błąd kompilacji.
 c. Brak średnika w {}.
 d. Brakuje `m_pointCount` – błąd czasu wykonania.
 e. Wszystko działa bez zarzutu.
2. Jakie błędy wystąpią w tej linii?
 a. Wszystko działa bez zarzutu.
 b. `m_printCount` musi być `const`.
 c. `m_printCount` musi być `explicit`.
 d. Błąd kompilacji – nie można zmienić `m_printCount`.
 e. `m_printCount` musi być `volatile`.
3. Jaki błąd wystąpi w tej linii?
 a. Wszystko działa bez zarzutu.
 b. `m_printCount` musi być `volatile`.
 c. `m_printCount` musi być `const`.
 d. `m_printCount` musi być `mutable`.
 e. `m_printCount` musi być `explicit`.
4. Jakie błędy wystąpią w tej linii?
 a. Nie można wywołać składowej `const`.
 b. Nie można wywołać składowej `nie-const`.
 c. Wszystko działa bez zarzutu.
 d. Funkcja `set` musi być `const`.

- e. Funkcja set musi być volatile.
5. Jakie błędy wystąpią w tej linii?
- a. Nie można wywołać składowej const.
 - b. Nie można wywołać składowej nie-const.
 - c. Funkcja print musi być const.
 - d. q nie może być const.
 - e. Funkcja set musi być volatile.
6. Jaki błąd wystąpi w tej linii?
- a. Wszystko działa bez zarzutu.
 - b. Nie można wywołać składowej nie-const.
 - c. Funkcja print musi być const.
 - d. q musi być explicit.
 - e. Nie można wywołać składowej const.
-

9. Znajdź błędy w przykładzie 2.33 i odpowiedz na pytania z przykładu 2.34.

PRZYKŁAD 2.33. src/quizzes/statics-quiz.cpp

```
// wadget.h:  
  
class Wadget {  
public:  
    Wadget(double a, double b);  
    void print();  
    static double calculation();  
    static int wadgetCount();  
  
private:  
    double m_d1, m_d2;  
    static int m_wadgetCount;  
};  
  
// wadget.cpp:  
  
Wadget::Wadget(double a, double b)  
: m_d1(a), m_d2(b) {  
    m_wadgetCount++;  
}  
  
static int wadgetCount() {  
    return m_wadgetCount;  
}  
  
double Wadget::calculation() {  
    return d1*d2 + m_wadgetCount;  
}  
[...]
```

PRZYKŁAD 2.34. src/quizzes/statics-quiz.txt

1. W kodzie przykładu 2.33 pojawia się kilka problemów. Pierwszy z nich polega na tym, że nie powiedzie się konsolidacja, ponieważ brakuje definicji `m_wadgetCount`. Jak możemy naprawić ten problem?
- a. W `wadget.h` wewnątrz definicji klasy:
`static int m_wadgetCount = 0;`

b. W wadget.h poza definicją klasy:
 static int Wadget::m_wadgetCount = 0;
c. W wadget.cpp na szczytce pliku:
 int Wadget::m_wadgetCount = 0;
d. W wadget.cpp na szczytce pliku:
 static int Wadget::m_wadgetCount = 0;
e. W wadget.cpp na liście inicjalizacji składowych:
 Wadget():Wadget()
 : m_d1(a), m_d2(b), m_wadgetCount(0)
 { m_wadgetCount ++; }

2. Rozważ deklarację:

```
static int Wadget::wadgetCount()  
Co oznacza w tym przypadku słowo kluczowe static?  
a. Funkcja musi zostać zdefiniowana w pliku .cpp.  
b. Funkcja może być wywoływana jedynie przez obiekty statyczne.  
c. Funkcja musi zostać wywołana z operatorem zakresu Wadget::.  
d. Nazwa funkcji ma zakres plikowy.  
e. Funkcja ma dostęp jedynie do składowych statycznych.
```

3. Rozważ definicję:

```
static int Wadget::wadgetCount()  
Co oznacza w tym przypadku słowo kluczowe static?  
a. Funkcja ma dostęp jedynie do statycznych składowych.  
b. Funkcję można wywołać jedynie na obiektach statycznych.  
c. Funkcja musi zostać wywołana z operatorem zakresu Wadget::.  
d. Nazwa funkcji jest eksportowana do konsolidatora.  
e. Nazwa funkcji nie jest eksportowana do konsolidatora.
```

4. Co możemy powiedzieć o definicji Wadget::calculation?

- a. d1 i d2 nie są dostępne z metody static.
- b. Brakuje static przed definicją funkcji.
- c. Wszystko działa bez zarzutu.
- d. Prawdziwe są odpowiedzi a i b.
- e. Nazwa funkcji nie jest eksportowana do konsolidatora (błąd).