6
OBIEKTY

Łyżka nie istnieje…
Neo w filmie „Matrix”

Lektura kilku ostatnich rozdziałów dała ci spore pojęcie o programowaniu w języku C++, ze szczególnym uwzględnieniem sposób realizacji w nim pewnych algorytmów oraz użycia takich konstrukcji jak pętle czy instrukcje warunkowe. Zapoznałeś się także z możliwościami, jakie oferuje ten język w zakresie manipulowania bardziej złożonymi porcjami informacji.

Wreszcie, miałeś sposobność realizacji konkretnej aplikacji - poczynając od jej zaprojektowania, a na kodowaniu i ostatecznej kompilacji skończywszy. Wierzę, iż samo programowanie było wtedy raczej zrozumiałe - chociaż nie pisaliśmy już wówczas trywialnego kodu.
Podejrzewam jednak, że wstępne konstruowanie programu nosiło dla ciebie znamiona co najmniej dziwnej czynności; wspominałem o tym zresztą w podsumowaniu całego naszego projektu, obiecując pokazanie w niniejszym rozdziale znacznie przyjaźniejszej, naturalniejszej i, jak sądzę, przyjemniejszej techniki programowania. Przyszedł czas, by spełnić tę obietnicę.

Zatem nie tracąc czasu, zajmijmy się tym wyczekiwanym tęsknie zagadnieniem :)

Przedstawiamy klasy i obiekty

Poznamy teraz sposób kodowania znany jako **programowanie obiektowe** (ang. *object-oriented programming* - OOP), które spełnia wszystkie niedawno złożone przeze mnie obietnice. Nie dziwi więc, iż jest to najszerzej stosowana przez dzisiejszych programistów technika projektowania i implementacji programów.

Nie zawsze jednak tak było; myślę zatem, że warto przyjrzeć się drodze, jaką pokonał fach o nazwie programowanie komputerów - od początku aż do chwili obecnej. Dzięki temu będziemy mogli lepiej docenić używane przez siebie narzędzia, z C++ na czele :)

Skrawek historii

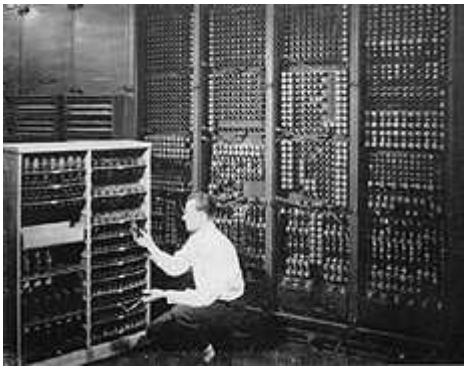
Pomysł programowania komputerów jest nawet starszy niż one same. Zanim bowiem powstały pierwsze maszyny zdolne do wykonywania sekwencji obliczeń, istniało już wiele teoretycznych modeli, wedle których miałyby funkcjonować[64].

Mało zachęcające początki

Nie dziwi więc, iż pojawienie się „mózgów elektronowych”, jak wtedy nazywano komputery, na wielu uniwersytetach w latach 50. wywołało spory entuzjazm. Mnóstwo

[64] Najbardziej znanym jest maszyna Turinga.

ludzi zaczęło zajmować się oprogramowywaniem tych wielkich i topornych urządzeń. Była to praca na wskroś heroiczna - zważywszy, że „pisanie" programów oznaczało wtedy odpowiednie dziurkowanie zwykłych papierowych kart i przepuszczanie je przez wnętrzności maszyny. Najmniejszy błąd zmuszał do uruchamiania programu od początku, co zazwyczaj skutkowało trafieniem na koniec kolejki oczekujących na możliwość skorzystania z drogocennej mocy obliczeniowej.



Fotografia 1. ENIAC - pierwsza maszyna licząca nazwana komputerem, skonstruowana w 1946 roku. Był to doprawdy cud techniki - przy poborze mocy równym zaledwie 130 kW mógł wykonać aż 5 tysięcy obliczeń na sekundę (ok. milion razy mniej niż współczesne komputery). (zdjęcie pochodzi z serwisu Internetowe Muzeum Starych Programów i Komputerów)

Zwyczaj jej starannego wydzielania utrzymał się przez wiele lat, choć z czasem techniki programistyczne uległy usprawnieniu. Kiedy koderzy (a właściwie hakerzy, bo w tych czasach głównie maniacy zajmowali się komputerami) dostali wreszcie do dyspozycji monitory i klawiatury (prymitywne i prawie w ogóle niepodobne do dzisiejszych cacek), programowanie zaczęło bardziej przypominać znajomą nam czynność i stało się nieco łatwiejsze. Jednakże określenie „przyjazne" było jeszcze zdecydowanie przedwczesne :) Zakodowanie programu oznaczało najczęściej konieczność wklepywania długich rzędów numerków, czyli jego **kodu maszynowego**. Dopiero później pojawiły się bardziej zrozumiałe, lecz nadal niezbyt przyjazne **języki asemblera**, w których liczbowe instrukcje procesora zastąpiono ich słownymi odpowiednikami. Cały czas było to jednak operowanie na bardzo **niskim poziomie abstrakcji**, ściśle związanym ze sprzętem. Listingi były więc mało czytelne i podobne np. do poniższego:

```
mov   ah, 4Ch
int   21h
```

Przyznasz chyba, że odgadnięcie działania tegoż kodu wymaga nielichych zdolności profetycznych[65] ;)

Wyższy poziom

Nie dziwi więc, że kiedy tylko potencjał komputerów na to pozwolił (a stało się to na początku lat 70.), powstały znacznie wygodniejsze w użyciu języki programowania **wysokiego poziomu** (algorytmiczne), zwane też językami **drugiej generacji**. Zawierały one, tak oczywiste dla nas, lecz wówczas nowatorskie, konstrukcje w rodzaju instrukcji warunkowych czy pętli. Nie były też zależne od konkretnej platformy sprzętowej, co czyniło programy w nich napisane wielce przenośnymi. Tak narodziło się programowanie strukturalne.

[65] Nie robi on jednak nic szczególnego, gdyż po prostu kończy działanie programu :) O dziwo, te dwie linijki powinny funkcjonować na prawie wszystkich dzisiejszych pecetach z systemami DOS lub Windows!

W tym okresie stworzone zostały znane i używane do dziś języki - Pascal, C czy BASIC. Programowanie stało się łatwiejsze, bardziej dostępne i popularniejsze - również wśród niewielkiej jeszcze grupy użytkowników domowych komputerów. Pociągnęło to za sobą także rozwój oprogramowania: pojawiły się systemy operacyjne w rodzaju Unixa, DOSa czy Windows (wszystkie napisane w C), rosła też liczba przeznaczonych dlań aplikacji. Chociaż niekiedy pisano jeszcze drobne fragmenty kodu w asemblerze, ogromna większość projektów była już realizowana wedle zasad programowania strukturalnego.

Można w zasadzie powiedzieć, że z posiadanymi umiejętnościami sytuujemy się właśnie w tym punkcie historii. Wprawdzie używamy języka C++, ale dotychczas korzystaliśmy jedynie z tych jego możliwości, które były dostępne także w C.
To się oczywiście wkrótce zmieni :)

Skostniałe standardy

Czasy świetności metod programowania strukturalnego trwały zaskakująco długo, bo aż kilkanaście lat. Może to się wydawać dziwne - szczególnie w odniesieniu do, przywoływanego już niejednokrotnie, wyjątkowo sztucznego projektowania kodu przy użyciu tychże metod. Jeżeli dodamy do tego fakt, iż już wtedy istniała całkiem pokaźna liczba języków **trzeciej generacji**, pozwalających na programowanie obiektowe[66], sytuacja jawi się wręcz niedorzecznie. Dlaczego koderzy nie porzucili swych wysłużonych i topornych instrumentów przez tak długi okres?…

„Winowajcą" jest głównie język C, który zdążył przez ten czas urosnąć do rangi niemal jedynego słusznego języka programowania. Jako że był on narzędziem, którego używano nawet do pisania systemów operacyjnych, istniało mnóstwo jego kompilatorów oraz ogromna liczba stworzonych w nim programów. Zmiana tak silnie zakorzenionego standardu była w zasadzie niemożliwa, toteż przez wiele lat nikt się jej nie podjął.

Obiektów czar

Aż tu w 1983 roku duński programista Bjarne Stroustrup zaprezentował stworzony przez siebie język C++. Miał on niezaprzeczalną zaletę (język, nie jego twórca ;D): łączył składnię C (przez co zachowywał kompatybilność z istniejącymi aplikacjami) z możliwościami programowania zorientowanego obiektowo.
Fakt ten sprawił, że C++ zaczął powoli wypierać swego poprzednika, zajmując czołowe miejsce wśród używanych języków programowania. Zajmuje je zresztą do dziś.

Obiektowych następców dorobiły się też dwa pozostałe języki strukturalne. Pascal wyewoluował w Object Pascala, który jest podstawą dla popularnego środowiska Delphi. BASIC'iem natomiast zaopiekował się Microsoft, tworząc z niego Visual Basic; dopiero jednak ostatnie wersje tego języka (oznaczone jako .NET) można nazwać w pełni obiektowymi.

Co dalej?

Zaraz, w takim razie programowanie obiektowe i nasz ulubiony język C++ mają już z górą dwadzieścia lat - w świecie komputerów to przecież cały eon! Czy zatem technologii tej nie czeka rychły schyłek?…

Możnaby tak przypuszczać, gdyby istniała inna, równorzędna wobec OOPu technika programowania. Dotychczas jednak nikt nie wynalazł niczego takiego i nie zanosi się na to w przewidywalnej przyszłości :) Programowanie obiektowe ma się dzisiaj co najmniej

[66] Były to na przykład LISP albo Smalltalk.

tak samo dobrze (a nawet znacznie lepiej), jak w chwili swego powstania i trudno sobie nawet wyobrazić jego ewentualny zmierzch.

Naturalnie, zawsze można się z tym nie zgodzić :) Niektórzy przekonują nawet, iż istnieje coś takiego jak języki **czwartej generacji**, zwane również deklaratywnymi. Zaliczają do nich na przykład SQL (język zapytań do baz danych) czy XSL (transformacje XML).
Nie da się jednak ukryć faktu, że obszar zastosowań każdego z tych języków jest bardzo specyficzny i ograniczony. Jeżeli bowiem kiedykolwiek będzie możliwe tworzenie zwykłych aplikacji przy pomocy następców tychże języków, to lada dzień zbędni staną się także sami programiści ;))

Pierwszy kontakt

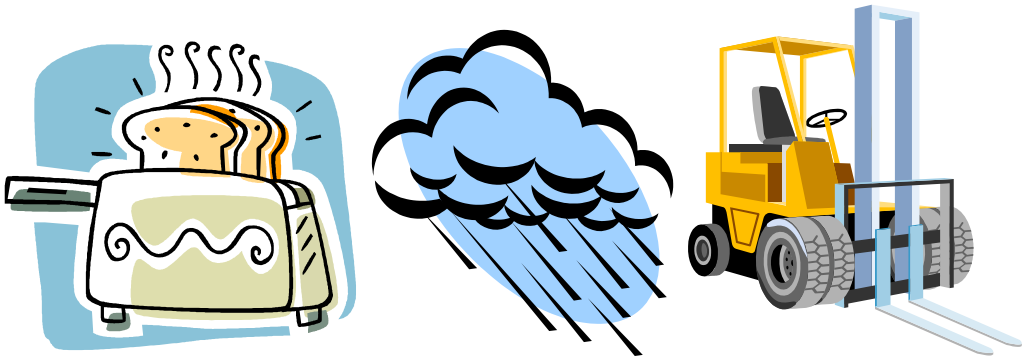
Nadeszła wreszcie pora, kiedy poznamy podstawowe założenia osławionego programowania obiektowego. Być może dowiemy się też, dlaczego jest takie wspaniałe ;)

Obiektowy świat

Z nazwy tej techniki programowania nietrudno wywnioskować, że jej najważniejszym pojęciem jest **obiekt**. Tworząc obiekty i definiując ich nowe rodzaje można zbudować dowolny program.

Wszystko jest obiektem

Ale czym w istocie jest taki obiekt? W języku potocznym słowo to może przecież oznaczać w zasadzie wszystko. Obiektem można nazwać lampę stojącą na biurku, drzewo za oknem, sąsiedni dom, samochód na ulicy, a nawet całe miasto. Jakkolwiek czasem będzie to dość dziwny sposób nazewnictwa, ale jednak należy go uznać za całkowicie dopuszczalny.



Rysunek 3, 4 i 5. Obiekty otaczają nas z każdej strony

Myśląc o programowaniu, znaczenie terminu ‘obiekt’ nie ulega zasadniczej zmianie. Także tutaj obiektem może być praktycznie wszystko. Różnica polega jednak na tym, iż programista występuje wówczas w roli stwórcy, pana i władcy wykreowanego „świata”. Wprowadzając nowe obiekty i zapewniając współpracę między nimi, tworzy działający system, podporządkowany realizacji określonego zadania.

Zanotujmy więc pierwsze spostrzeżenie:

Obiekt może reprezentować **cokolwiek**. Programista wykorzystuje obiekty jako cegiełki, z których buduje gotowy program.

Określenie obiektu

Przed chwilą wykazaliśmy, że programowanie nie jest wcale tak oderwane od rzeczywistości, jak się powszechnie sądzi :D Faktycznie techniki obiektowe powstały właśnie dlatego, żeby przybliżyć nieco kodowanie do prawdziwego świata.

O ile jednak w odniesieniu do niego możemy swobodnie używać dość enigmatycznego stwierdzenia, że „obiektem może być wszystko", o tyle programowanie nie znosi przecież żadnych nieścisłości. Obiekt musi więc dać się jasno zdefiniować i w jednoznaczny sposób reprezentować w programie.
Wydawać by się mogło, iż to duże ograniczenie. Ale czy tak jest naprawdę?…

Wiele wskazuje na to, że nie. Pojęcie obiektu w rozumieniu programistycznym jest bowiem na tyle elastyczne, że mieści w sobie niemal wszystko, co tylko można sobie wymarzyć. Mianowicie:

Obiekt składa się z opisujących go **danych** oraz może wykonywać ustalone **czynności**.

Podobnie jak omówione niedawno struktury, obiekty zawierają **pola**, czyli zmienne. Ich rolą jest przechowywanie pewnych informacji o obiekcie - jego charakterystyki. Oczywiście, liczba i typy pól mogą być swobodnie definiowane przez programistę. Oprócz tego obiekt może wykonywać na sobie pewne działania, a więc uruchamiać zaprogramowane funkcje; nazywamy je **metodami** albo **funkcjami składowymi**. Czynią one obiekt tworem **aktywnym** - nie jest on jedynie pojemnikiem na dane, lecz może samodzielnie nimi manipulować.

Co to wszystko oznacza w praktyce? Najlepiej będzie, jeżeli prześledzimy to na przykładzie.
Załóżmy, że chcemy mieć w programie obiekt jadącego samochodu (bo może piszemy właśnie grę wyścigową?). Ustalamy więc dla niego pola, które będą go określały, oraz metody, które będzie mógł wykonywać.
Polami mogą być widoczne cechy auta: jego marka czy kolor, a także te mniej rzucające się w oczy, lecz pewnie ważne dla nas: długość, waga, aktualna prędkość i maksymalna szybkość. Natomiast metodami uczynimy czynności, jakie nasz samochód mógłby wykonywać: przyspieszenie, hamowanie albo skręt.
W ten oto prosty sposób stworzymy więc komputerową reprezentację samochodu. W naszej grze moglibyśmy mieć wiele takich aut i nic nie stałoby na przeszkodzie, aby każde miało np. inny kolor czy markę. Kiedy zaś dla jednego z nich wywołalibyśmy metodę skrętu czy hamowania, zmieniłaby się prędkość tylko **tego jednego** samochodu - zupełnie tak, jakby kierowca poruszył kierownicą lub wcisnął hamulec.

Schemat 16. Przykładowy obiekt samochodu

W idei obiektu widać zatem przeciwieństwo programowania strukturalnego. Tam musieliśmy rozdzielać dane programu od jego kodu, co przy większych projektach prowadziłoby do sporego bałaganu. W programowaniu obiektowym jest zgoła odwrotnie: tworzymy niewielkie cząstki, będące połączeniem informacji oraz działania. Są one

niemal „namacalne”, dlatego łatwiej jest nam myśleć o nich o składnikach programu, który budujemy.

Zapiszmy zatem drugie spostrzeżenie:

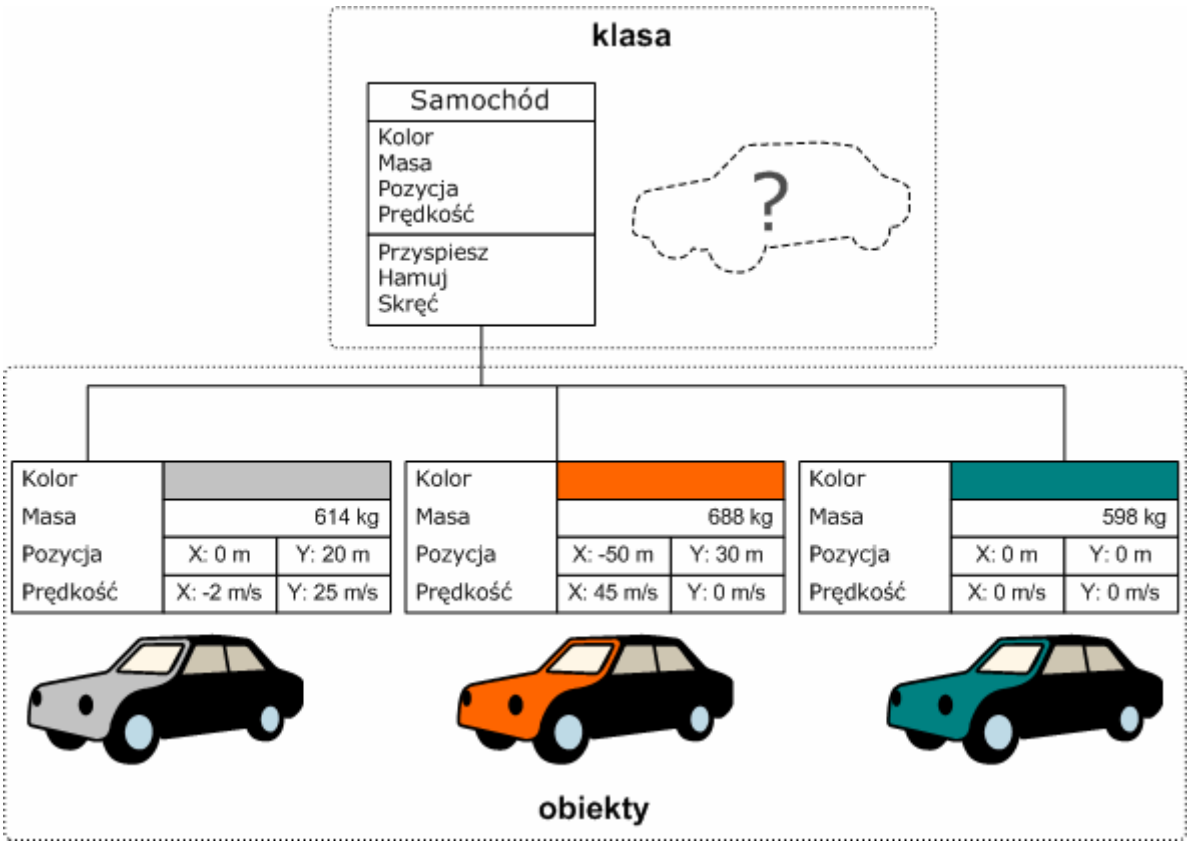
Obiekty zawierają zmienne, czyli **pola**, oraz mogą wykonywać dla siebie ustalone funkcje, które zwiemy **metodami**.

Obiekt obiektowi nierówny

Zestaw pól i metod rzadko jest charakterystyczny dla pojedynczego obiektu. Najczęściej istnieje wiele obiektów, każdy z właściwymi sobie wartościami pól. Łączy je jednak przynależność do jednego i tego samego rodzaju, który nazywamy **klasą**.

Klasy wprowadzają więc pewną systematykę w świat obiektów. Byty należące do tej samej klasy są bowiem do siebie **podobne**: mają ten sam pakiet **pól** oraz mogą wykonywać na sobie te same **metody**. Informacje te zawarte są w **definicji** klasy i wspólne dla wszystkich wywodzących się z niej obiektów.
Klasa jest zatem czymś w rodzaju wzorca - matrycy, wedle którego „produkowane” są kolejne obiekty (**instancje**) w programie. Mogą one różnić się od siebie, ale tylko co do **wartości poszczególnych pól**; wszystkie będą jednak należeć do tej samej klasy i będą mogły wykonywać na sobie te same metody.

Kot o czarnej sierści i kot o białej sierści to przecież jeden i ten sam gatunek *Felis catus*…



Schemat 17. Definicja klasy oraz kilka należących doń obiektów (jej instancji)

W programowaniu obiektowym zadaniem twórcy jest przede wszystkim zaprojektowanie modelu klas programu, zawierającego definicję wszystkich klas występujących w aplikacji. Podczas działania programu będą z nich tworzone obiekty, których współpraca ma zapewnić realizację celów aplikacji (przynajmniej w teorii ;D).

Zatem zamiast zajmować się oddzielnie danymi oraz kodem, bierzemy pod uwagę ich odpowiednie **połączenia** - obiekty, „aktywne struktury”. Definiując odpowiednie klasy oraz umieszczając w programie instrukcje kreujące obiekty tych klas, budujemy nasz program kawałek po kawałku.
Być może brzmi to teraz trochę tajemniczo, lecz niedługo zobaczysz, iż w gruncie rzeczy jest bardzo proste.

Sformułujmy na koniec ostatnie spostrzeżenie:

Każdy obiekt należy do pewnej **klasy**. Definicja klasy zawiera **pola**, z których składa się ów obiekt, oraz **metody**, którymi dysponuje.

Co na to C++?

Zakończmy na razie te nieco zbyt teoretyczne dywagacje i zajmijmy się tym, co programiści lubią najbardziej, czyli kodowaniem :) Zobaczymy, jak C++ radzi sobie z ideą programowania obiektowego. Na razie spojrzymy na to zagadnienie przez kilka prostych przykładów, by później zagłębić się w nie nieco bardziej.

Definiowanie klas

Pierwszym i bardzo ważnym etapem tworzenia kodu opartego na idei OOP jest, jak sobie powiedzieliśmy, zdefiniowanie odpowiednich klas. W C++ jest to całkiem proste.

Klasy są tu *de facto* nowymi typami danych, podobnymi w pewnym sensie do struktur[67]. Dlatego też naturalnym miejscem umieszczania ich definicji są pliki nagłówkowe - umożliwia to łatwe wykorzystanie klasy w obrębie całego programu.
Spójrzmy zatem na przykładową definicję typu obiektów, który pary razy przewijał się w tekście:

```
class CCar
{
   private:
         float m_fMasa;
         COLOR m_Kolor;

         VECTOR2 m_vPozycja;
   public:
         VECTOR2 vPredkosc;

         //---------------------------------------------------------------

         // metody
         void Przyspiesz(float fIle);
         void Hamuj(float fIle);
         void Skrec(float fKat);
};
```

Zastosowanie tu typy danych `COLOR` i `VECTOR2` mają charakter umowny. Powiedzmy, że `COLOR` w jakiś sposób reprezentuje kolor, zaś `VECTOR2` jest dwuwymiarowym wektorem (o współrzędnych `x` i `y`).

Porównanie do struktury jest całkiem na miejscu, chociaż pojawiło nam się kilka nowych elementów, w tym najbardziej oczywiste zastąpienie słowa kluczowego `struct` przez `class`.

[67] W C++ różnica między klasą a strukturą jest zresztą czysto kosmetyczna.

Najważniejsze dla nas jest jednak pojawienie się **deklaracji metod** klasy. Mają one tutaj formę prototypów funkcji, więc będą musiały być zaimplementowane gdzie indziej (jak - o tym niedługo powiemy). Równie dobrze wszak można wpisywać kod krótkich metod bezpośrednio w definicji ich klasy.
Oprócz tego mamy w naszej klasie także pewne pola, które deklarujemy w identyczny sposób jak zmienne czy pola w strukturach. To one stanowią treść obiektów, należących do definiowanej klasy.

Nietrudno zauważyć, że cała definicja jest podzielona na dwie części poprzez etykiety `private` i `public`. Być może domyślasz , cóż mogą one znaczyć; jeżeli tak, to punkt dla ciebie :) A jeśli nie, nic straconego - niedługo wyjaśnimy ich działanie. Chwilowo możesz je więc zignorować.

Implementacja metod

Zdefiniowanie typu obiektowego, czyli klasy, nie jest najczęściej ostatnim etapem jego określania. Jeżeli bowiem umieściliśmy weń **prototypy** jakichś **metod**, nieodzowne jest wpisanie ich **kodu** w którymś z modułów programu. Zobaczmy zatem, jak należy to robić.

Przede wszystkim należy udostępnić owemu modułowi definicję klasy, co prawie zawsze oznacza konieczność dołączenia zawierającego ją pliku nagłówkowego. Jeśli zatem nasza klasa jest zdefiniowana w pliku *klasa.h*, to w module kodu musimy umieścić dyrektywę:

```
#include "klasa.h"
```

Potem możemy już przystąpić do implementacji metod.

Ich kody wprowadzamy w niemal ten sam sposób, który stosujemy dla zwykłych funkcji. Jedyna różnica tkwi bowiem w nagłówkach tychże metod, na przykład:

```
void CCar::Przyspiesz(float fIle)
{
   // tutaj kod metody
}
```

Zamiast więc samej nazwy funkcji mamy tutaj także nazwę odpowiedniej **klasy**, umieszczoną wcześniej. Oba te miana rozdzielamy znanym już skądinąd operatorem zasięgu `::`.
Dalej następuje zwyczajowa lista parametrów i wreszcie zasadnicze ciało metody. Wewnątrz tego bloku zamieszczamy instrukcje, składające się na kod danej funkcji.

Tworzenie obiektów

Posiadając zdefiniowaną i zaimplementowaną klasę, możemy pokusić się o stworzenie paru przynależnych jej obiektów.

Istnieje przynajmniej kilka sposobów na wykonanie tej czynności, z których najprostszy nie różni się niczym od zadeklarowania struktury i wygląda chociażby tak:

```
CCar Samochod;
```

Kod ten spowoduje zadeklarowanie nowej zmiennej `Samochod` typu `CCar` oraz **stworzenie obiektu** należącego do tej klasy. Podkreślam to, gdyż moment tworzenia obiektu nie jest wcale taką błahą sprawą i może powodować różne akcje. Powiemy sobie o tym niedługo.

Mając już obiekt (a więc instancję klasy), jesteśmy w stanie operować na wartościach jego pól oraz wywoływać przynależne jego klasie metody. Posługujemy się tu znajomym operatorem kropki (`.`):

```
// przypisanie wartości polu
Samochod.vPredkosc.x = 100.0;
Samochod.vPredkosc.y = 50.0;

// wywołanie metody obiektu
Samochod.Przyspiesz (10.0);
```

Czy nie spotkaliśmy już kiedyś czegoś podobnego?… Zdaje się, że tak. Przy okazji łańcuchów znaków pojawiła się bowiem konstrukcja typu `strTekst.length()`, której użyliśmy do pobrania długości napisu `strTekst`.
Było to nic innego jak tylko wywołanie metody `length()` dla obiektu `strTekst`! Napisy w C++ są więc obiektami, pochodzącymi od klasy `std::string`. Oprócz `length()` posiadają zresztą wiele innych metod, ułatwiających pracę z nimi. Większość poznamy podczas omawiania Biblioteki Standardowej.

Kod wygląda zatem całkiem logicznie i spójnie; łatwo bowiem znaleźć wszystkie instrukcje dotyczące obiektu `Samochod`, bo zaczynają się one od jego nazwy. To jedna (choć może mało znacząca) z licznych zalet programowania obiektowego, które poznasz wkrótce i na które z utęsknieniem czekasz ;)

***

W tym podrozdziale zaliczyliśmy pierwsze spotkanie z programowaniem zorientowanym obiektowo. Mamy więc już jakieś pojęcie o klasach, obiektach oraz ich polach i metodach - także w odniesieniu do języka C++.

Dalsza część rozdziału będzie miała charakter systematyzacyjno-uzupełniający :) Wyjaśnimy i uporządkujemy sobie większość szczegółów dotyczących definiowania klas oraz tworzenia obiektów. Informuję przeto, iż absencja na tym ważnym wykładzie będzie zdecydowanie nierozsądna!

Obiekty i klasy w C++

Szczycąc się chlubnym mianem języka w pełni obiektowego, C++ posiada wszystko, co niezbędne do praktycznej realizacji idei programowanie zorientowanego obiektowo. Teraz właśnie przyjrzymy się dokładnie tym konstrukcjom językowym - wytłumaczymy sobie ich działanie oraz sposób użycia.

Klasa jako typ obiektowy

Wiemy już, że pisanie programu zgodnie z filozofią OOP polega na definiowaniu i implementowaniu odpowiednich klas oraz tworzeniu z nich obiektów i manipulowaniu nimi. Klasa jest więc dla nas pojęciem kluczowym, które na początek wypadałoby wyjaśnić:

Klasa to złożony typ zmiennych, składający się z **pól**, przechowujących dane, oraz posiadający **metody**, wykonujące zaprogramowane czynności.

Zmienne należące do owych typów **obiektowych** nazywamy oczywiście **obiektami**.

Każdy obiekt posiada swój własny pakiet opisujących go pól, które rezydują w pamięci operacyjnej w identyczny sposób jak pola struktur. Metody są natomiast kodem **wspólnym** dla całej klasy, zatem w czasie działania programu istnieje w pamięci tylko **jedna** ich kopia, wywoływana w razie potrzeby na rzecz **różnych** obiektów. Jest to, jak sądzę, dość oczywiste: tworzenie odrębnych kopii tych samych przecież funkcji dla każdego nowego obiektu byłoby niewątpliwie szczytem absurdu.

Dwa etapy określania klasy

Skoro dowiedzieliśmy się dokładnie, czym są klasy i jak (w teorii) działają, spójrzmy na sposoby ich wykorzystania w języku C++. Zaczniemy rzecz jasna od wprowadzania do programu własnych typów obiektowych, gdyż bez tego ani rusz :)

Na początek warto przypomnieć, iż klasa jest typem (podobnie jak struktura czy `enum`), więc właściwym dla niej miejscem byłby zawsze plik nagłówkowy. Jednocześnie jednak zawiera ona kod swoich funkcji składowych, czyli metod, co czyni ją przynależną do jakiegoś modułu (bo tylko wewnątrz modułów można umieszczać funkcje).
Te dwa przeciwstawne stanowiska sprawiają, że określenie klasy jest najczęściej **rozdzielone** na dwie części:

- **definicję**, wstawianą w pliku nagłówkowym, w której określamy pola klasy oraz wpisujemy prototypy jej metod
- **implementację**, umieszczaną w module, będącą po prostu kodem wcześniej zdefiniowanych metod

Układ ten nie dość, że działa nadzwyczaj dobrze, to jeszcze realizuje jeden z postulatów programowania obiektowego, jakim jest **ukrywanie niepotrzebnych szczegółów**. Tymi szczegółami będzie tutaj kod poszczególnych metod, którego znajomość nie jest wcale potrzebna do korzystania z klasy.

Co więcej, może on nie być w ogóle dostępny w postaci pliku *.cpp*, a jedynie w wersji skompilowanej! Tak jest chociażby w przypadku biblioteki DirectX, o czym przekonasz się za czas jakiś.

Domyślasz się zatem, że za chwilę skoncentrujemy się na tych dwóch etapach określania klasy, a więc na definicji i implementacji. Jakkolwiek nie brzmi to zbyt odkrywczo, jednak masz tutaj całkowitą słuszność :D

Czasem, jeszcze przed definicją klasy musimy poinformować kompilator, że dana nazwa jest faktycznie klasą. Robimy tak na przykład wtedy, gdy obiekt klasy `A` odwołuje się do klasy `B`, zaś `B` do `A`. Używamy wtedy deklaracji zapowiadającej, pisząc po prostu `class A;` lub `class B;`.
Takie przypadki są dosyć rzadkie, ale warto wiedzieć, jak sobie z nimi radzić. O tym sposobie wspomnimy zresztą nieco dokładniej, gdy będziemy zajmować się klasami zaprzyjaźnionymi.

Definicja klasy

Jest to konieczna i często pierwsza czynność przy wprowadzaniu do programu nowej klasy. Jej definicja precyzuje bowiem zawarte w niej pola oraz deklaracje metod, którymi klasa będzie dysponowała.

Informacje te są niezbędne, aby móc utworzyć obiekt danej klasy; dlatego też umieszczamy je niemal zawsze w pliku nagłówkowym - miejscu należnym własnym typom danych.
Składnia definicji klasy wygląda natomiast następująco:

```
class nazwa_klasy
```

```
{
    [specyfikator_dostępu:]
          [pola]
          [metody]
};
```

Nie widać w niej zbytnich restrykcji, gdyż faktycznie jest ona całkiem swobodna. Kolejność poszczególnych elementów (pól lub metod) nie jest ściśle ustalona i może być w zasadzie dowolnie zmieniana. Najlepiej jednak zachować w tym względzie jakiś porządek, grupując np. pola i metody w zwarte grupy.
Na razie wszakże trudno byłoby stosować się do tych rad, skoro nie omówiliśmy dokładnie wszystkich części definicji klasy. Czym prędzej więc naprawiamy ten błąd :)

Kontrola dostępu do składowych klasy

Fraza oznaczona jako *specyfikator_dostępu* pewnie nie mówi ci zbyt wiele, chociaż spotkaliśmy się już z nią w którejś z przykładowych klas. Przyjmowała ona tam formę `private` lub `public`, dzieląc cała definicję na jakby dwie odrębne sekcje. Nietrudno wywnioskować, iż podział ten nie ma jedynie charakteru wizualnego, ale powoduje dalej idące konsekwencje. Jakie?…

Nazwa *specyfikator_dostępu*, chociaż brzmi może nieco sztucznie (jak zresztą wiele terminów w programowaniu :)), dobrze oddaje rolę, jaką ta konstrukcja pełni. Otóż **specyfikuje** ona właśnie prawa **dostępu** do części składowych klasy (czyli pól lub metod), wyróżniając ich dwa rodzaje: **prywatne** (ang. *private*) oraz **publiczne** (ang. *public*).
Różnica między nimi jest znacząca i bardzo ważna, gdyż wpływa na to, które elementy klasy są widoczne tylko w ramach jej samej, a które także na zewnątrz. Te pierwsze nazywamy więc prywatnymi, zaś drugie publicznymi.

Prywatne składowe klasy (wpisane po słowie `private:` w jej definicji) są dostępne jedynie **wewnątrz samej klasy**, tj. tylko dla jej własnych metod.

Publiczne składowe klasy (wpisane po słowie `public:` w jej definicji) widoczne są zawsze i **wszędzie** - nie tylko dla samej klasy (jej metod), ale **na zewnątrz** - np. dla jej obiektów.

Danym specyfikatorem objęte są wszystkie następujące po nim części klasy, aż do jej końca lub… kolejnego specyfikatora :) Ich ilość nie jest bowiem niczym ograniczona.

Nic więc nie stoi na przeszkodzie, aby nie było ich wcale! W takiej sytuacji wszystkie składowe będą miały domyślne reguły dostępu. W przypadku klas (definiowanych poprzez `class`) jest to dostęp prywatny, natomiast dla typów strukturalnych[68] (słówko `struct`) - dostęp publiczny.
Trudno uwierzyć, ale w C++ jest to jedyna różnica pomiędzy klasami a strukturami! Słowa `class` i `struct` są więc niemal synonimami; jest to rzecz niespotykana w innych językach programowania, w których te dwie konstrukcje są zupełnie odrębne.

Dla skuteczniejszego rozwiania z powyższego opisu możliwej mgły niejasności, spójrzmy na ten oto przykładowy program i klasę:

```
// DegreesCalc - kalkulator temperatur

// typ wyliczeniowy określający skalę temperatur
```

[68] A także dla unii, chociaż jak wiemy, funkcjonują one inaczej niż struktury i klasy.

```
enum SCALE {SCL_CELSIUS = 'c', SCL_FAHRENHEIT = 'f', SCL_KELVIN = 'k'};

class CDegreesCalc
{
   private:
         // temperatura w stopniach Celsjusza
         double m_fStopnieC;
   public:
         // ustawienie i pobranie temperatury
         void UstawTemperature(double fTemperatura, SCALE Skala);
         double PobierzTemperature(SCALE Skala);
};

// ------------------------ funkcja main() -----------------------------

void main()
{
   // zapytujemy o skalę, w której będzie wprowadzona wartość
   char chSkala;
   std::cout << "Wybierz wejsciowa skale temperatur" << std::endl;
   std::cout << "(c - Celsjusza, f - Fahrenheita, k - Kelwina): ";
   std::cin >> chSkala;
   if (chSkala != 'c' && chSkala != 'f' && chSkala != 'k') return;

   // zapytujemy o rzeczoną temperaturę
   float fTemperatura;
   std::cout << "Podaj temperature: ";
   std::cin >> fTemperatura;

   // deklarujemy obiekt kalkulatora i przekazujemy doń temp.
   CDegreesCalc Kalkulator;
   Kalkulator.UstawTemperature (fTemperatura,
                                static_cast<SCALE>(chSkala));

   // pokazujemy wynik - czyli temperaturę we wszystkich skalach
   std::cout << std::endl;
   std::cout << "- stopnie Celsjusza: "
             << Kalkulator.PobierzTemperature(SCL_CELSIUS) << std::endl;
   std::cout << "- stopnie Fahrenheita: "
          << Kalkulator.PobierzTemperature(SCL_FAHRENHEIT) << std::endl;
   std::cout << "- kelwiny: "
             << Kalkulator.PobierzTemperature(SCL_KELVIN) << std::endl;

   // czekamy na dowolny klawisz
   getch();
}
```

Cała aplikacja jest prostym programem przeliczającym między trzema skalami temperatur:

Screen 36. Kalkulator przeliczający wartości temperatur

Jej pełny kod, z implementacją metod klasy `CDegreesCalc`, znaleźć można w programach przykładowych. Nas jednak bardziej interesuje forma definicji tejże klasy oraz podział jej składowych na prywatne oraz publiczne.
Widzimy więc wyraźnie, iż klasa posiada jedno prywatne pole - jest nim `m_fStopnieC`, w którym zapisywana jest temperatura w wewnętrznie używanej, wygodnej skali Celsjusza. Oprócz niego mamy jeszcze dwie publiczne metody - `UstawTemperature()` oraz `PobierzTemperature()`, dzięki którym uzyskujemy dostęp do naszego prywatnego pola. Jednocześnie oferują nam jednak dodatkową funkcjonalność, jaką jest dokonywanie przeliczania pomiędzy wartościami wyrażonymi w różnych miarach.

To bardzo częsta sytuacja, gdy prywatne pole klasy „obudowane" jest publicznymi metodami, zapewniającymi doń dostęp. Daje to wiele pożytecznych możliwości, jak choćby kontrola przypisywanej polu wartości czy tworzenie pól tylko do odczytu. Jednocześnie „prywatność" pola chroni je przed przypadkową, niepożądaną ingerencją z zewnątrz.
Takie zjawisko wyodrębniania pewnych fragmentów kodu nazywamy **hermetyzacją**.

Jak wiemy, prywatne składowe klasy nie są dostępne poza nią samą. Kiedy więc tworzymy nasz obiekt:

```
CDegreesCalc Kalkulator;
```

jesteśmy niejako „skazani" na korzystanie tylko z jego publicznych metod; próba odwołania się do prywatnego pola (poprzez `Kalkulator.m_fStopnieC`) skończy się bowiem błędem kompilacji.
Fakt ten wcale nas jednak nie ogranicza, lecz zabezpiecza przed niepowołanym dostępem do wewnętrznych informacji klasy, które z zasady powinny być do jej wyłącznej dyspozycji. Do komunikacji z otoczeniem istnieją za to dwie publiczne metody, i to z nich właśnie będziemy korzystać w funkcji `main()`.

Najpierw więc wywołujemy funkcję składową `UstawTemperature()`, podając jej wpisaną przez użytkownika wartość oraz wybraną skalę[69]:

```
Kalkulator.UstawTemperature (fTemperatura, static_cast<SCALE>(chSkala));
```

W tym momencie w ogóle nie interesują nas działania, które zostaną na tych danych podjęte - jest to wewnętrzna sprawa klasy `CDegreesCalc` (podobnie zresztą jak jej pole `m_fStopnieC`). Ważne jest, że w ich następstwie możemy użyć drugiej metody, `PobierzTemperature()`, do uzyskania podanej wcześniej wartości w wybranej przez siebie, nowej skali:

```
std::cout << "- stopnie Celsjusza: "
          << Kalkulator.PobierzTemperature(SCL_CELSIUS) << std::endl;
// itd.
```

Wszystkie kwestie dotyczące szczegółowych aspektów przeliczania owych wartości są zatem szczelnie poukrywane. Kod funkcji `main()` jest klarowny i wolny od niepotrzebnych detali, co nie zmienia faktu, iż w razie potrzeby możliwe jest zajęcie się nimi. Wystarczy przecież rzucić okiem implementacje metod klasy `CDegreesCalc`.

Zaprowadzanie porządku poprzez ograniczanie dostępu do pewnych elementów klasy to jedna z reguł, a jednocześnie zalet programowania obiektowego. Do jej praktycznej

[69] Znowu stosujemy tu technikę odpowiedniego dobrania wartości typu wyliczeniowego, przez co unikamy instrukcji `switch`.

realizacji służą w C++ poznane specyfikatory `private` oraz `public`. W miarę nabywania doświadczenia w pracy z klasami będziesz je coraz efektywniej stosował w swoim własnym kodzie.

Deklaracje pól

Pola są właściwą treścią każdego obiektu klasy, to one stanowią jego reprezentację w pamięci operacyjnej. Pod tym względem nie różnią się niczym od znanych ci już pól w strukturach i są po prostu zwykłymi zmiennymi, zgrupowanymi w jedną, kompleksową całość.

Jako miejsce na przechowywanie wszelkiego rodzaju danych, pola mają kluczowe znaczenie dla obiektów i dlatego powinny być chronione przez niepowołanym dostępem z zewnątrz. Przyjęło się więc, że w zasadzie wszystkie pola w klasach deklaruje się jako **prywatne**; ich nazwy zwykle poprzedza się też przedrostkiem `m_`, aby odróżnić je od zmiennych lokalnych:

```
class CFoo[70]
{
   private:
         int m_nJakasLiczba;
         std::string m_strJakisNapis;
```

Dostęp do danych zawartych w polach musi się zatem odbywać za pomocą dedykowanych metod. Rozwiązanie to ma wiele rozlicznych zalet: pozwala chociażby na tworzenie pól, które można jedynie odczytywać, daje sposobność wykrywania niedozwolonych wartości (np. indeksów przekraczających rozmiary tablic itp.) czy też podejmowania dodatkowych akcji podczas operacji przypisywania.
Rzeczone funkcje mogą wyglądać chociażby tak:

```
   public:
         int JakasLiczba()             { return m_nJakasLiczba;     }
         void JakasLiczba(int nLiczba) { m_nJakasLiczba = nLiczba;  }
         std::string JakisNapis()      { return m_strJakisNapis;    }
};
```

Nazwałem je tu identycznie jak odpowiadające im pola, pomijając jedynie przedrostki[71]. Niektórzy stosują nazwy w rodzaju `Pobierz...()`/`Ustaw...()` czy też z angielskiego - `Get...()`/`Set...()`. Leży to całkowicie w zakresie upodobań programisty.
Użycie naszych metod „dostępowych” może zaś przedstawiać się na przykład tak:

```
CFoo Foo;
Foo.JakasLiczba (10);           // przypisanie 10 do pola m_nJakasLiczba
std::cout << Foo.JakisNapis();  // wyświetlenie pola m_strJakisNapis
```

Zauważmy przy okazji, że pole `m_strJakisNapis` może być tutaj jedynie odczytane, gdyż nie przewidzieliśmy metody do nadania mu jakiejś wartości. Takie postępowanie jest często pożądane, ale zależy rzecz jasna od konkretnej sytuacji, a tu jest jedynie przykładem.

Wielkim mankamentem C++ jest brak wsparcia dla tzw. **właściwości** (ang. *properties*), czyli „nakładek” na pola klas, imitujących zmienne i pozwalających na użycie bardziej

[70] `foo` oraz `bar` to takie dziwne nazwy, stosowane przez programistów najczęściej w przykładowych kodach, dla bliżej nieokreślonych bytów, nie mających żadnego praktycznego sensu i służących jedynie w celach prezentacyjnych. Mają one tę zaletę, że nie można ich pomylić tak łatwo, jak np. litery `A`, `B`, `C`, `D` itp.
[71] Sprawia to, że funkcje odpowiadające temu samemu polu, a służące do zapisu i odczytu, są przeciążone.

naturalnej składni (choćby operatora =) niż dedykowane metody.
Wiele kompilatorów udostępnia więc tego rodzaju funkcjonalność we własnym zakresie - w Visual C++ jest to konstrukcja `__declspec(property(...))`, o której możesz przeczytać w MSDN. Nie dorównuje ona jednak podobnym mechanizmom znanym z Delphi.

Metody i ich prototypy

Metody czynią klasy. To dzięki swym funkcjom składowym pasywne zbiory danych, którymi są struktury, stają się aktywnymi obiektami.

Z praktycznego punktu widzenia metody niewiele różnią się od zwyczajnych funkcji - oczywiście poza faktem, iż są deklarowane zawsze wewnątrz jakiejś klasy:

```
class CFoo
{
   public:
         void Metoda();
         int InnaMetoda(int);
         // itp.
};
```

Deklaracje te mogą mieć formę **prototypów** funkcji, a stworzone w ten sposób metody wymagać jeszcze **implementacji**, czyli wpisania ich kodu. Czynnością tą zajmiemy się dokładnie w następnym paragrafie.
Warto jednak wiedzieć, że dopuszczalne jest także wprowadzanie kodu metod bezpośrednio **wewnątrz bloku** `class`. Robiliśmy tak zresztą w przypadku metod dostępowych do pól, a w podobnych sytuacjach rozwiązanie to sprawdza się bardzo dobrze. Nie należy aczkolwiek postępować w ten sposób z długimi metodami, zawierającymi skomplikowane algorytmy, gdyż może to spowodować znaczący wzrost rozmiaru wynikowego pliku EXE.

Kompilator traktuje bowiem takie funkcje jako **inline**, tzn. rozwijane w miejscu wywołania, i **wstawia** cały ich **kod** przy każdym odwołaniu się do nich. Dla krótkich, jednolinijkowych metod jest to dobre rozwiązanie, przyspieszające działanie programu. Dla dłuższych nie musi wcale takie być.
Dokładniejszych informacji na ten temat oraz o samych funkcjach inline tradycyjnie można znaleźć w MSDN.

To jeszcze nie koniec zabawy z metodami :) Niektóre z nich można mianowicie uczynić **stałymi**. Zabieg ten sprawia, że funkcja, na której go zaaplikujemy, nie może **modyfikować** żadnego z **pól klasy**[72], a tylko je co najwyżej odczytywać.
Po co komu takie udziwnienie? Teoretycznie jest to pewna wskazówka dla kompilatora, który być może uczyni nam w zamian łaskę poczynienia jakichś optymalizacji. Praktycznie jest to też pewien sposób na zabezpieczenie się przed omyłkowym zmodyfikowaniem obiektu w metodzie, która wcale nie miała czegoś takiego robić. Jednym słowem korzyści są piorunujące ;)
Uczynienie jakiejś metody stałą jest banalnie proste: wystarczy tylko dodać za listą jej parametrów magiczne słówko `const`, np.:

```
class CFoo
{
   private:
```

[72] No może nie całkiem żadnego; istnieje pewien drobny wyjątek od tej reguły, ale jest on na tyle drobny i na tyle sproadycznie stosowany, że nie wyjaśniam go bliżej i odsyłam tylko purystów do stosownego wyjaśnienia w MSDN.

```
        int m_nPole;
    public:
        int Pole() const  { return m_nPole; }
};
```

Funkcja `Pole()` (będąca *de facto* obudową dla zmiennej `m_nPole`) będzie tutaj słusznie metodą stałą.

Dla szczególnie zainteresowanych polecam lekturę uzupełniającą o stałych metodach, znajdującą się w miejscu wiadomym :)

Konstruktory i destruktory

Przebąkiwałem już parokrotnie o procesie tworzenia obiektów, podkreślają przy tym znaczenie tego procesu. Za chwilę wyjaśni się, dlatego jest to takie ważne…

Decydując się na zastosowanie technik obiektowych w konkretnym programie musimy mieć na uwadze fakt, iż oznacza to zdefiniowane przynajmniej kilku klas oraz instancji tychże. Istotą OOPu jest poza tym odpowiednia **komunikacja** między obiektami: wymiana danych, komunikatów, podejmowanie działań zmierzających do realizacji danego zdania, itp. Aby zapewnić odpowiedni przepływ informacji, krystalizuje się mniej lub bardziej rozbudowana **hierarchia obiektów**, kiedy to jeden obiekt **zawiera** w sobie drugi, czyli jest jego **właścicielem**. To dość naturalne: większość otaczających nas rzeczy można przecież rozłożyć na części, z których się składają (gorzej może być z powtórnym złożeniem ich w całość :D).
Konsekwencje tego stanu rzeczy dla procesu tworzenie (i niszczenia) obiektów są raczej oczywiste: kreacja obiektu zbiorczego musi pociągnąć za sobą stworzenie jego składników; podobnie jest też z jego destrukcją. Jasne, można te kwestie zostawić kompilatorowi, ale paradoksalnie czyni to kod trudniejszym do zrozumienia, pisania i konserwacji[73].

C++ oferuje nam na szczęście możliwość podjęcia odpowiednich działań zarówno podczas tworzenia obiektu, jak i jego niszczenia. Korzystamy z niej, wprowadzając do naszej klasy dwa specjalne rodzaje metod - są to tytułowe **konstruktory** oraz **destruktory**.

Konstruktor to specyficzna funkcja składowa klasy, wywoływana zawsze podczas tworzenia należącego doń obiektu.

Typowym zadaniem konstruktora jest zainicjowanie pól ich początkowymi wartościami, przydzielenie pamięci wykorzystywanej przez obiekt czy też uzyskanie jakichś kluczowych danych z zewnątrz.
Deklaracja konstruktora jest w C++ bardzo prosta. Metoda ta nie zwraca bowiem żadnej wartości (nawet `void`!), a jej nazwa odpowiada nazwie zawierającej ją klasy. Wygląda więc mniej więcej tak:

```
class CFoo
{
    private:
        // jakieś przykładowe pole...
        float m_fPewnePole;
    public:
        // no i przyszła pora na konstruktora ;-)
        CFoo()      { m_fPewnePole = 0.0; }
```

[73] Wbrew pozorom to racjonalna reguła: im więcej jest rzeczy, które kompilator robi „za plecami” programisty, tym bardziej zagmatwany jest kod - choćby nawet był krótszy.

```
};
```

Zazwyczaj też konstruktor nie przyjmuje żadnych parametrów, co nie znaczy jednak, że nie może tego czynić. Często są to na przykład startowe dane przypisywane do pól:

```
class CSomeObject
{
   private:
         // jakiś rodzaj współrzędnych
         float m_fX, m_fY;
   public:
         // konstruktory
         CSomeObject()                          { m_fX = m_fY = 0.0;    }
         CSomeObject(float fX, float fY)        { m_fX = fX; m_fY = fY; }
};
```

Posiadanie takiego parametryzowanego konstruktora ma pewien wpływ na sposób tworzenia obiektów, gdyż musimy wtedy podać dlań odpowiednie wartości. Dokładniej wyjaśnimy to w następnym paragrafie.
Warto też wiedzieć, że klasa może posiadać kilka konstruktorów - tak jak na powyższym przykładzie. Działają one wtedy podobnie jak funkcje przeciążane; decyzja, który z nich faktycznie zostanie wywołany, zależy więc od instrukcji tworzącej obiekt.

Z wiadomych względów konstruktory czynimy zawsze metodami publicznymi. Umieszczenie ich w sekcji `private` dałoby bowiem dość dziwny efekt: taka klasa nie mogłaby być normalnie instancjowana, tzn. niemożliwe byłoby utworzenie z niej obiektu w zwykły sposób.

OK, konstruktory mają zatem niebagatelną rolą, jaką jest powoływania do życia nowych obiektów. Doskonale jednak wiemy, że nic nie jest wieczne i nawet najdłużej działający program kiedyś będzie musiał być zakończony, a jego obiekty zniszczone. Tą niechlubną robotą zajmuje się kolejny, wyspecjalizawany rodzaj metod - **destruktory**.

Destruktor jest specjalną metodą, przywoływaną podczas niszczenia obiektu zawierającej ją klasy.

W naszych przykładowych klasach destruktor nie miałby wiele do zrobienia - zgoła nic, ponieważ żaden z prezentowanych obiektów nie wykonywał czynności, po których należałoby sprzątać. To się wszak niedługo zmieni, zatem poznanie destruktorów z pewnością nie będzie szkodliwe :)
Postać destruktora jest także niezwykle prosta i w dodatku zawsze identyczna. Funkcja ta nie bierze bowiem żadnych parametrów (bo i jakie miałaby brać?) i niczego nie zwraca. Jej nazwą jest zaś nazwa zawierającej klasy poprzedzona znakiem tyldy (`~`).

Nazewnictwo destruktorów to jedna z niewielu rzeczy, za które twórcom C++ należą się tęgie baty :D O co dokładnie chodzi?
Otóż teoretycznie znak tyldy uzyskujemy za pomocą klawisza Shift oraz tego znajdującego się w lewym górnym rogu alfanumerycznej części klawiatury. Problem polega na tym, że po pierwszym jego użyciu żądany znak nie pojawia się na ekranie. Dzieje się tak dlatego, iż dawniej za jego pomocą uzyskiwało się litery specyficzne dla pewnych języków, z kreseczkami - np. ś, é czy ó.
Fakt ten możnaby zignorować, jako że większość liter nie posiada swoich „kreseczkowych" odpowiedników, więc wciśnięcie ich klawiszy po znaku tyldy powoduje pojawienie się zarówno osławionego szlaczka, jak i samej litery. Do tej grupy nie należy jednak litera C, którą to przyjęło się pisać na początku nazw klas. Zamiast więc żądanej sekwencji `~C` uzyskujemy… `Ć`!
Jak sobie z tym radzić? Ja nawykłem do dwukrotnego przyciskania klawisza tyldy, a

następnie usuwania nadmiarowego znaku. Możliwe jest też użycie jakiejś „neutralnej" litery w miejsce C, a następnie skasowanie jej. Chyba najlepsze jest jednak wciskanie klawisza tyldy, a następnie spacji - wprawdzie to dwa przyciśnięcia, ale w ich wyniku otrzymujemy sam wężyk.

Klasa wyposażona w odpowiedni destruktor może zatem jawić się następująco:

```
class CBar
{
   public:
         // konstruktor i destruktor
         CBar()      { /* czynności startowe */ }  // konstruktor
         ~CBar()     { /* czynności kończące */ }  // destruktor
};
```

Jako że jego forma jest ściśle określona, **jedna klasa** może posiadać tylko **jeden destruktor**.

Coś jeszcze?

Pola, zwykłe metody oraz konstruktory i destruktory to zdecydowanie najczęściej spotykane i chyba najważniejsze elementy klas. Aczkolwiek nie jedyne; w dalszej części tego kursu poznamy jeszcze składowe statyczne, funkcje przeciążające operatory oraz tzw. deklaracje przyjaźni (naprawdę jest coś takiego! :D). Poznane tutaj składniki klasy będą jednak zawsze miały największe znaczenie.

Można jeszcze wspomnieć, że wewnątrz klasy (a także struktury i unii) możemy zdefiniować… kolejną klasę! Taką definicję nazywamy wtedy zagnieżdżoną. Technika ta nie jest stosowana zbyt często, więc zainteresowani poczytają o niej w MSDN :)
Podobnie zresztą jest z innymi typami, określanymi poprzez `enum` czy `typedef`.

Implementacja metod

Definicja klasy jest zazwyczaj tylko połową sukcesu i nie stanowie wcale końca jej określania. Dzieje się tak przynajmniej wtedy, gdy umieścimy w niej jakieś prototypy metod, bez podawania ich kodu.

Uzupełnieniem definicji klasy jest wówczas jej **implementacja**, a dokładniej owych prototypowanych funkcji składowych. Polega ona rzecz jasna na wprowadzeniu instrukcji składających się na kod tychże metod w jednym z modułów programu.
Operację tę rozpoczynamy od dołączenia do rzeczonego modułu pliku nagłówkowego z definicją naszej klasy, np.:

```
#include "klasa.h"
```

Potem możemy już zająć się każdą z niezaimplementowanych metod; postępujemy tutaj bardzo podobnie, jak w przypadku zwykłych, globalnych funkcji. Składnia metody wygląda bowiem następująco:

```
[typ_wartości/void] nazwa_klasy::nazwa_metody([parametry]) [const]
{
   instrukcje
}
```

Nowym elementem jest w niej *nazwa_klasy*, do której należy dana funkcja. Wpisanie jej jest konieczne: po pierwsze mówi ona kompilatorowi, że ma do czynienia z metodą klasy, a nie zwyczajną funkcją; po drugie zaś pozwala bezbłędnie zidentyfikować macierzystą klasę danej metody.

Między nazwą klasy a nazwą metody widoczny jest operator zasięgu `::`, z którym już raz mieliśmy przyjemność się spotkać. Teraz możemy oglądać go w nowej, chociaż zbliżonej roli.

Zaleca się, aby bloki metod tyczące się jednej klasy umieszczać w zwartej grupie, jeden pod drugim. Czyni to kod lepiej zorganizowanym.

Dwie jeszcze nowości można zauważyć w nagłówku metody. Zaznaczyłem mianowicie *`typ_zwracanej_wartości`* lub `void` jako jego nieobowiązkową część. Faktycznie może ona być **zbędna** - ale tylko w przypadku **konstruktora** tudzież **destruktora** klasy. Dla zwykłych funkcji składowych musi ona nadal występować.
Ostatnią różnicą jest ewentualny modyfikator `const`, który, jak pamiętamy, czyni metodę stałą. Jego obecność w tym miejscu powinna się pokrywać z występowaniem także w prototypie funkcji. Niezgodność w tej kwestii zostanie srodze ukarana przez kompilator :)

Oczywiście większością implementacji metody będzie blok jej *`instrukcji`*, tradycyjnie zawarty między nawiasami klamrowymi. Cóż ciekawego można o nim powiedzieć? Bynajmniej niewiele: nie różni się prawie wcale od analogicznych bloków globalnych funkcji. Dodatkowo jednak ma on dostęp do **wszystkich pól i metod** swojej klasy - tak, jakby były one jego zmiennymi albo funkcjami lokalnymi.

Wskaźnik `this`

Z poziomu metody mamy dostęp do jeszcze jednej, bardzo ważnej i przydatnej informacji. Chodzi tutaj o obiekt, na rzecz którego nasza metoda jest wywoływana; mówiąc ściśle, o odwołanie (**wskaźnik**) do niego.
Cóż to znaczy?… Przypomnijmy sobie zatem którąś z przykładowych klas, prezentowanych na poprzednich stronach. Gdybyśmy wywołali jakąś jej metodę, przypuśćmy że w ten sposób:

```
CFoo Foo;
Foo.JakasMetoda();
```

to wewnątrz bloku funkcji `CFoo::JakasMetoda()` moglibyśmy użyć omawianego wskaźnika, by zyskać pełen wgląd w obiekt `Foo`! Czasem mówi się więc, iż jest to dodatkowy, specjalny parametr metody - występuje przecież w jej wywołaniu.

Ów wyjątkowy wskaźnik, o którym traktuje powyższy opis, nazywa się `this` („to”). Używamy go zawsze wtedy, gdy potrzebujemy odwołać się do obiektu jako **całości**, a nie tylko do poszczególnych pól. Najczęściej oznacza to przekazanie go do jakiejś funkcji, zwykle konstruktora innego obiektu.
Jako że jest to wskaźnik, a nie obiekt *explicité*, korzystanie z niego różni się nieco od postępowania z „normalnymi” zmiennymi obiektowymi. Więcej na ten temat powiemy sobie w dalszej części tego rozdziału, zaś całkowicie wyjaśnimy w rozdziale 8, *Wskaźniki*.

Dla dociekliwych zawsze jednak istnieje MSDN :]

Praca z obiektami

Nawet dziesiątki wyśmienitych klas nie stanowią jeszcze gotowego programu, a jedynie pewien rodzaj reguł, wedle których będzie on realizowany. Wprowadzenie tych reguł w życie wymaga przeto stworzenia **obiektów** na podstawie zdefiniowanych klas.

W C++ mamy dwa główne sposoby „obchodzenia” się z obiektami; różnią się one pod wieloma względami, inne jest też zastosowanie każdego z nich. Naturalną i rozsądną koleją rzeczy będzie więc przyjrzenie się im obu :)

Zmienne obiektowe

Pierwszą strategię znamy już bardzo dobrze, używaliśmy jej bowiem niejednokrotnie nie tylko dla samych obiektów, lecz także dla wszystkich innych zmiennych.
W tym trybie korzystamy z klasy dokładnie tak samo, jak ze wszystkich innych typów w C++ - czy to wbudowanych, czy też definiowanych przez nas samych (jak `enum`'y, struktury itd.).

Deklarowanie zmiennych i tworzenie obiektów

Zaczynamy oczywiście od deklaracji zmiennej, niebędącej dla nas żadną niespodzianką:

```
CFoo Obiekt;
```

Powyższa linijka kodu wykonuje jednak znacznie więcej czynności, niż jest to widoczne na pierwszy czy nawet drugi rzut oka. Ona mianowicie:

- wprowadza nam nową zmienną `Obiekt` typu `CFoo`. Nie jest to rzecz jasna żadna nowość, ale dla porządku warto o tym przypomnieć.
- tworzy w pamięci operacyjnej obszar, w którym będą przechowywane **pola obiektu**. To także nie jest zaskoczeniem: pola, jako bądź co bądź zmienne, muszą rezydować gdzieś w pamięci, więc robią to w identyczny sposób jak pola struktur.
- wywołuje konstruktor klasy `CFoo` (czyli procedurę `CFoo::CFoo()`), by dokończył aktu kreacji obiektu. Po jego zakończeniu możemy uznać nasz obiekt za ostatecznie stworzony i gotowy do użycia.

Te trzy etapy są niezbędne, abyśmy mogli bez problemu korzystać z stworzonego obiektu. W tym przypadku są one jednak realizowane całkowicie automatycznie i nie wymagają od nas żadnej uwagi. Przekonamy się później, że nie zawsze tak jest i, co ciekawe, wcale nie będziemy tym zmartwieni :D

Muszę jeszcze wspomnieć o pewnym drobnym wymaganiu, stawianym nam przez kompilator, któremu chcemy podać wiersz kodu umieszczony na początku paragrafu. Otóż klasa `CFoo` musi tutaj posiadać **bezparametrowy konstruktor**, albo też nie mieć wcale procedury tego rodzaju (wtedy etap z jej wywoływaniem zostanie po prostu pominięty).
W innym przypadku potrzebne jest jeszcze przekazanie odpowiednich parametrów konstruktorowi, który takowych wymaga. Konieczność tą realizujemy podobną metodą, co wywołanie zwyczajnej funkcji:

```
CFoo Foo(10, "jakiś tekst");     // itp.
```

Czy nie przypomina nam to czegoś?… Ależ oczywiście - identycznie postępowaliśmy z łańcuchami znaków (czyli obiektami klasy `std::string`), tworząc je chociażby tak:

```
#include <string>
std::string strBuffer("Jakie te obiekty są proste! ;-)");
```

Widzimy więc, że znany nam i lubiany typ `std::string` wyjątkowo podpada pod zasady programowania obiektowego :)

Żonglerka obiektami

Zadeklarowane przed chwilą zmienne obiektowe są w istocie takimi samymi zmiennymi, jak wszystkie inne w programach C++. Możliwe jest zatem przeprowadzanie nań operacji, którym podlegają na przykład liczby całkowite, napisy czy tablice.

Nie mam tu wcale na myśli jakichś złożonych manipulacji, wymagających skomplikowanych algorytmów, lecz całkiem zwyczajnych i codziennych, jak przypisanie czy przekazywanie do funkcji.
Czy można powiedzieć cokolwiek ciekawego o tak trywialnych czynnościach? Okazuje się, że tak. Zwrócimy wprawdzie uwagę na dość oczywiste fakty z nimi związane, lecz znajomość owych „banałów" okaże się później niezwykle przydatna. Przy okazji będzie to dobra okazja to powtórzenia nabytej wiedzy, a tego przecież nigdy dość :D

Na użytek dalszych wyjaśnień zdefiniujemy sobie taką oto klasę lampy:

```
class CLamp
{
   private:
         COLOR m_Kolor;          // kolor lampy
         bool m_bWlaczona;       // czy lampa świeci się?
   public:
         // konstruktory
         CLamp()                 { m_Kolor = COLOR_WHITE; }
         CLamp(COLOR Kolor)      { m_Kolor = Kolor; }

         //-------------------------------------------------------------

         // metody
         void Wlacz()            { m_bWlaczona = true; }
         void Wylacz()           { m_bWlaczona = false; }

         //-------------------------------------------------------------

         // metody dostępowe do pól
         COLOR Kolor() const     { return m_Kolor; }
         bool Wlaczona() const   { return m_bWlaczona; }
};
```

Klasa ta jest znakomitą syntezą wszystkich wiadomości przekazanych w tym podrozdziale. Jeżeli więc nie rozumiesz do końca znaczenia któregoś z jej elementów, powinieneś powrócić do poświęconemu mu miejsca w tekście.

Natychmiast też zadeklarujemy i stworzymy dwa obiekty należące do naszej klasy:

```
CLamp Lampa1(COLOR_RED), Lampa2(COLOR_GREEN);
```

Tym sposobem mamy więc lampy, sztuk dwie, w kolorze czerwonym oraz zielonym. Moglibyśmy użyć ich metod, aby je obie włączyć; zrobimy jednak coś dziwniejszego - **przypiszemy** jedną lampę do drugiej:

```
Lampa1 = Lampa2;
```

„A co to za dziwadło?", słusznie pomyślisz. Taka operacja jest jednak całkowicie poprawna i daje dość ciekawe rezultaty. By ją dobrze zrozumieć musimy pamiętać, że `Lampa1` oraz `Lampa2` są to przede wszystkim **zmienne**, zmienne które przechowują pewne **wartości**. Fakt, że tymi wartościami są obiekty, które w dodatku interpretujemy w sposób prawie realny, nie ma tutaj większego znaczenia.
Pomyślmy zatem, jaki efekt spowodowałby ten kod, gdybyśmy zamiast klasy `CLamp` użyli jakiegoś zwykłego, skalaranego typu?…

```
int nLiczba1 = 10, nLiczba2 = 20;
nLiczba1 = nLiczba2;
```

Dawna wartośc zmiennej, do której nastąpiło przypisanie, zostałaby zapomniana i obie zmienne zawierałyby tę samą liczbę.
Dla obiektów rzecz ma się identycznie: po wykonaniu przypisania zarówno `Lampa1`, jak i `Lampa2` reprezentować będą obiekty zielonych lamp. Czerwona lampa, pierwotnie zawarta w zmiennej `Lampa1`, zostanie **zniszczona**[74], a w jej miejsce pojawi się **kopia** zawartości zmiennej `Lampa2`.

Nie bez powodu zaakcentowałem wyżej słowo „kopia”. Obydwa obiekty są bowiem od siebie całkowicie **niezależne**. Jeżeli włączylibyśmy jeden z nich:

```
Lampa1.Wlacz();
```

drugi nie zmieniłby się wcale i nie obdarzył nas swym własnym światłem.

Możemy więc podsumować nasz wywód krótką uwagą na temat zmiennych obiektowych:

Zmienne obiektowe przechowuje obiekty w ten sam sposób, w jaki czynią to zwykłe zmienne ze swoimi wartościami. Identycznie odbywa się też przypisywanie[75] takich zmiennych - tworzone są wtedy odpowiednie **kopie** obiektów.

Wspominałem, że wszystko to może wydawać się naturalne, oczywiste i niepodważalne. Konieczne było jednak dokładne wyjaśnienie w tym miejscu tych z pozoru prostych zjawisk, gdyż drugi sposób postępowania z obiektami (który poznamy za moment) wprowadza w tej materii istotne zmiany.

Dostęp do składników

Kontrolowanie obiektu jako całości ma rozliczne zastosowania, ale jednak znacznie częściej będziemy używać tylko jego pojedynczych składników, czyli pól lub metod.

Doskonale wiemy już, jak się to robi: z pomocą przychodzi nam zawsze **operator wyłuskania** - kropka (`.`). Stawiamy więc go po nazwie obiektu, by potem wpisać nazwę wybranego elementu, do którego chcemy się odwołać.

Pamiętajmy, że posiadamy wtedy dostęp jedynie do składowych **publicznych** klasy, do której należy obiekt.

Dalsze postępowanie zależy już od tego, czy naszą uwagę zwróciliśmy na pole, czy na metodę. W tym pierwszym, rzadszym przypadku nie odczujemy żadnej różnicy w stosunku do pól w strukturach - i nic dziwnego, gdyż nie ma tu rzeczywiście najmniejszej rozbieżności :) Wywołanie metody jest natomiast łudząco zbliżone do uruchomienia zwyczajnej funkcji - tyle że w grę wchodzą tutaj nie tylko jej parametry, ale także obiekt, na rzecz którego daną metodę wywołujemy.

Jak wiemy, jest on potem dostępny wewnątrz metody poprzez wskaźnik `this`.

Niszczenie obiektów

Każdy stworzony obiekt musi prędzej czy poźniej zostać zniszczony, aby móc odzyskać zajmowaną przez niego pamięć i spokojnie zakończyć program. Dotyczy to także zmiennych obiektowych, lecz dzieje się to trochę jakby za plecami programisty.

[74] W pełnym znaczeniu tego słowa - z wywołaniem destruktora i późniejszym zwolnieniem pamięci.
[75] To samo można zresztą powiedzieć o wszystkich operacjach podobnych do przypisania, tj. inicjalizacji oraz przekazywaniu do funkcji.

Zauważmy bowiem, iż w żadnym z naszych dotychczasowych programów, wykorzystujących techniki obiektowe, nie pojawiły się instrukcje, które jawnie odpowiadałyby za niszczenie stworzonych obiektów. Nie oznacza to bynajmniej, że zalegają one w pamięci operacyjnej[76], zajmując ją niepotrzebnie. Po prostu kompilator sam dba o to, by ich destrukcja nastąpiła w stosownej chwili.

A zatem kiedy jest ona faktycznie dokonywana? Nietrudno jest obmyślić odpowiedź na to pytanie, jeżeli przypomnimy sobie pojęcie **zasięgu** zmiennej. Powiedzieliśmy sobie ongiś, iż jest to taki obszar kodu programu, w którym dana zmienna jest **dostępna**. Dostępna - to znaczy zadeklarowana, z przydzieloną dla siebie pamięcią, a w przypadku zmiennej obiektowej - posiadająca również obiekt **stworzony** poprzez konstruktor klasy.
Moment **opuszczenia zasięgu** zmiennej przez punkt wykonania programu jest więc kresem jej istnienia. Jeśli nieszczęsna zmienna była obiektową, do akcji wkracza destruktor klasy (jeżeli został określony), sprzątając ewentualny bałagan po obiekcie i **niszcząc** go. Dalej następuje już tylko zwolnienie pamięci zajmowanej przez zmienną i jej kariera kończy się w niebycie :)

Zapamiętajmy więc, że:

Wyjście programu **poza zasięg** zmiennej obiektowej **niszczy** zawarty w niej obiekt.

Podsumowanie

Prezentowane tu własności zmiennych obiektowych być może wyglądają na nieznane i niespotkane wcześniej. Naprawdę jednak nie są niczym szczególnym, gdyż spotykaliśmy się z nimi od samego początku nauki programowania - w większości (z wyłączeniem wyłuskiwania składników) dotyczą one bowiem **wszystkich** zmiennych!
Teraz wszakże omówiliśmy je sobie nieco dokładniej, koncentrując się przede wszystkim na „życiu" obiektów - chwilach ich tworzenia i niszczenia oraz operacjach na nich. Mając ugruntowaną tę więdzę, będzie nam łatwiej zmierzyć się z drugim sposobem stosowania obiektów, który jest przedstawiony w następnym paragrafie.

Wskaźniki na obiekty

Przyznam szczerze: miałem pewne wątpliwości, czy słuszne jest zajmowanie się wskaźnikami na obiekty już w tej chwili, bez dogłebnego przedstawienia samych wskaźników. Tę naruszoną przeze mnie kolejność zachowałaby pewnie większość autorów kursów czy książek o C++.
Ja jednak postawiłem sobie za cel nauczenie czytelnika *programowania w języku C++* (i to w konkretnym celu!), nie zaś samego *języka C++*. Narzuca to nieco inny porządek treści, skoncentrowany w pierwszej kolejności na najpotrzebniejszych zagadnieniach praktycznych, a dopiero potem na pozostałych możliwościach języka. Do tych „kwestii pierwszej potrzeby" niewątpliwie należy zaliczyć ideę programowania obiektowego, wskaźniki spychając tym samym na nieco dalszy plan.
Jednocześnie jednak nie mogę przy okazji OOPu pominąć milczeniem tematu wskaźników na obiekty, które są praktycznie niezbędne do poprawnego konstruowania aplikacji z wykorzystaniem klas. Dlatego też pojawia się on właśnie teraz; mimo wszystko ufam, że zrozumienie go nie będzie dla ciebie wielkim kłopotem.

Po tak „zachęcającym" wstępie nie będę zdziwiony, jeżeli w tej chwili duża część czytelników zakończy lekturę ;-) Skrycie wierzę jednak, że ambitnym kandydatom na programistów gier żadne wskaźniki nie będą straszne, a już na pewno nie przelękną się ich obiektowych odmian. Nie bedziemy zatem tracić więcej czasu oraz miejsca i natychmiast przystąpimy do dzieła.

[76] Zjawisko to nazywamy wyciekiem pamięci i jest ono wysoce niepożądane, zaś interesować nas będzie bardziej w rozdziale traktującym o wskaźnikach.

Deklarowanie wskaźników i tworzenie obiektów

Od czegóż to mielibyśmy zacząć, jeżeli nie od jakiejś zmiennej? W końcu bez zmiennych nie ma obiektów, a bez obiektów nie ma programowania (obiektowego :D). Zadeklarujmy więc na początek taką oto dziwną zmienną:

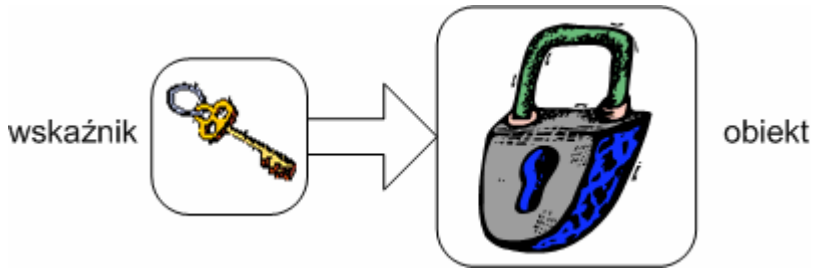
```
CFoo* pFoo;
```

Wszystko byłoby tu znajome, gdyby nie ta gwiazdka przy nazwie klasy `CFoo`. To właśnie ona sprawia, że `pFoo` nie jest zmienną obiektową, ale właśnie **wskaźnikiem na obiekt**, w tym przypadku obiekt klasy `CFoo`.

To ważne stwierdzenie - `pFoo` nie jest tutaj obiektem, on może co najwyżej na taki obiekt **wskazywać**. Innymi słowy, może być jedynie **odwołaniem** do obiektu, połączeniem z nim - ale zmienna ta nie będzie nigdy sama przechowywać żadnych danych, należących do owego obiektu. Będzie raczej czymś w rodzaju pozycji w spisie treści, odnoszącej się do rozdziału w książce.
Niniejsza linijka kodu **nie tworzy** więc żadnego obiektu, a jedynie przygotowuje nań miejsce w programie. Właściwa kreacja musi nastąpić później i wygląda nieco inaczej niż to, do czego przywykliśmy:

```
pFoo = new CFoo;
```

Słówko `new` („nowy”, niektórzy każą je zwać operatorem) służy właśnie do utworzenia obiektu. Wykonuje ono prawie wszystkie czynności potrzebne do realizacji tego procesu, a więc przydziela odpowiednią ilość pamięci dla naszego obiektu i wywołuje konstruktor jego klasy.
Czym zatem zasługuje sobie na odrębność? Podstawową różnicą jest to, że tworzony obiekt jest umieszczany w **dowolnym miejscu pamięci**, a nie w którejś z naszych zmiennych (a już na pewno nie w `pFoo`!). Nie oznacza to jednakże, iż nie mamy o nim żadnych informacji i nie możemy z niego normalnie korzystać. Otóż `pFoo` staje się tutaj **łącznikiem** z naszym odległym tworem; za **pośrednictwem** tego wskaźnika mamy bowiem pełną swobodę dostępu do stworzonego obiektu. Jak się wkrótce przekonasz, możliwe jest przy jego pomocy odwoływanie się do składników obiektu (pól i metod) w niemal taki sam sposób, jak w przypadku zmiennych obiektowych.



Schemat 18. Wskaźnik na obiekt jest pewnego rodzaju kluczem do niego

Jeden dla wszystkich, wszystkie do jednego

Ogromne i ważne różnice ujawniają się dopiero podczas manipulowania kilkoma takimi wskaźnikami. Mam tu na myśli przede wszystkim instrukcje przypisania, rozważane już dokładnie dla zmiennych obiektowych. Teraz podobne eksperymenta będziemy dokonywali na wskaźnikach; zobaczymy, dokąd nas one zaprowadzą…

Do naszych celów po raz kolejny spożytkujemy zdefiniowaną w poprzednim paragrafie klasę `CLamp`. Zaczniemy jednak od zadeklarowania wskaźnika na obiekt tej klasy z jednoczesnym stworzeniem obiektu lampy:

```
CLamp* pLampa1 = new CLamp;
```

Przypominam, iż w ten sposób powołaliśmy do życia obiekt, który został umieszczony **gdzieś** w pamięci, a wskaźnik `pLampa1` jest tylko odwołaniem do niego.
Dalszej części nietrudno się domyśleć. Wprowadzamy sobie zatem drugi wskaźnik i przypisujemy doń ten pierwszy, o tak:

```
CLamp* pLampa2 = pLampa1;
```

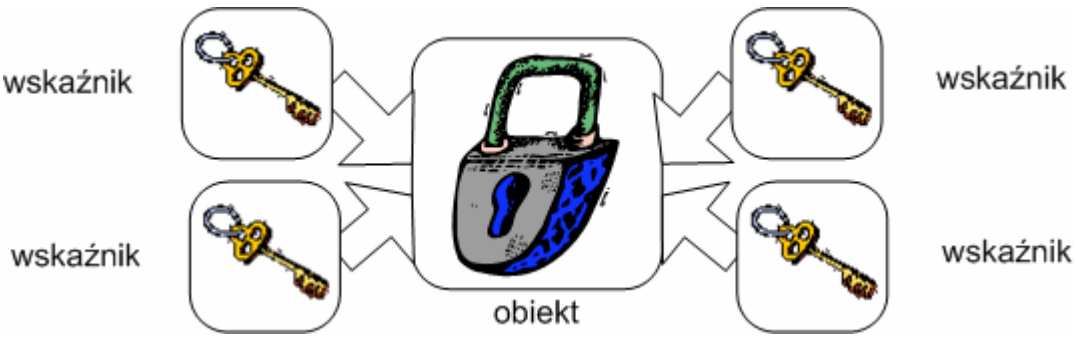
Mamy teraz dwa **takie same wskaźniki**… Czy to znaczy, iż posiadamy także parę identycznych obiektów?

Otóż nie! Nasza lampa nadal egzystuje samotnie, bowiem skopiowaliśmy jedynie samo **odwołanie** do niej. Obecnie użycie zarówno wskaźnika `pLampa1`, jak i `pLampa2` będzie uzyskaniem dostępu do **jednego i tego samego obiektu**.
To znacząca modyfikacja w stosunku do zmiennych obiektowych. Tam każda reprezentowała i przechowywała swój własny obiekt, a instrukcje przypisywania między nimi powodowały wykonywanie kopii owych obiektów.
Tutaj natomiast mamy tylko **jeden obiekt**, za to **wiele dróg dostępu** do niego, czyli wskaźników. Przypisywanie między nimi dubluje jedynie te drogi, zaś sam obiekt pozostaje niewzruszony.

Podsumowując:

Wskaźnik na obiekt jest jedynie **odwołaniem** do niego. Wykonanie przypisania do wskaźnika może więc co najwyżej **skopiować owo odwołanie**, pozostawiając docelowy obiekt całkowicie **niezmienionym**.

Mówiąc obrazowo, uzyskiwanie dodatkowego wskaźnika do obiektu jest jak wyrobienie sobie dodatkowego klucza do tego samego zamka. Choćbyśmy mieli ich cały brelok, wszystkie będą otwierały tylko jedne i te same drzwi.



Schemat 19. Możemy mieć wiele wskaźników do tego samego obiektu

Dostęp do składników

Cały czas napomykam, że wskaźnik jest pewnego rodzaju łączem do obiektu. Wypadałoby więc wresznie połączyć się z tym obiektem, czyli uzyskać dostęp do jego składników.

Operacja ta nie jest zbytnio skomplikowana, gdyż by ją wykonać posłużymy się znaną już koncepcją **operatora wyłuskania**. W przypadku wskaźników nie jest nim jednak kropka, ale strzałka (->). Otrzymujemy ją, wpisując kolejno dwa znaki: myślnika oraz symbolu większości.
Aby zatem włączyć naszą lampę, wystarczy wywołać jej odpowiednią metodę przy pomocy któregoś z dwóch wskaźników oraz poznanego właśnie operatora:

```
pLampa1->Wlacz();
```

Możemy także sprawdzić, czy drugi wskaźnik istotnie odwołuje się do tego samego obiektu co pierwszy. Wystarczy wywołać za jego pomocą metodę `Wlaczona()`:

```
pLampa2->Wlaczona();
```

Nie będzie niespodzianką fakt, iż zwróci ona wartość `true`.

Zbierzmy więc w jednym miejscu informacje na temat obu operatorów wyłuskania:

Operator kropki (`.`) pozwala uzyskać dostęp do składników obiektu zawartego w **zmiennej obiektowej**.

Operator strzałki (`->`) wykonuje analogiczną operację dla **wskaźnika na obiekt**.

Jak najlepiej zapamiętać i rozróżniać te dwa operatory? Proponuję prosty sposób:

- pamiętamy, że zmienna obiektowa przechowuje obiekt jako swoją wartość. Mamy go więc dosłownie „na wyciągnięcie ręki" i nie potrzebujemy zbytnio się wysilać, aby uzyskać dostęp do jego składników. Służący temu celowi operator może więc być bardzo mały, tak mały jak… punkt :)
- kiedy zaś używamy wskaźnika na obiekt, wtedy nasz byt jest daleko stąd. Potrzebujemy wówczas odpowiednio dłuższego, dwuznakowego operatora, który dodatkowo wskaże nam (strzałka!) właściwą drogę do poszukiwanego obiektu.

Takie wyjaśnienie powinno być w miarę pomocne w przyswojeniu sobie znaczenia oraz zastosowania obu operatorów.

Niszczenie obiektów

Wszelkie obiekty kiedyś należy zniszczyć; czynność ta, oprócz wyrabiania dobrego nawyku sprzątania po sobie, zwalnia pamięć operacyjną, które te obiekty zajmowały. Po zniszczeniu wszystkich możliwe jest bezpieczne zakończenie programu.

Podobnie jak tworzenie, tak i niszczenie obiektów dostępnych poprzez wskaźniki nie jest wykonywane automatycznie. Wymagana jest do tego odrębna instrukcja - na szczęście nie wygląda ona na wielce skomplikowaną i przedstawia się następująco:

```
delete pFoo;   // pFoo musi tu być wskaźnikiem na istniejący obiekt
```

`delete` („usuń", podobnie jak `new` jest uważane za operator) dokonuje wszystkich niezbędnych czynności potrzebnych do zniszczenia obiektu reprezentowanego przez wskaźnik. Wywołuje więc jego destruktor, a następnie zwalnia pamięć zajętą przez obiekt, który kończy wtedy definitywnie swoje istnienie.

To tyle jeśli chodzi o życiorys obiektu. Co się jednak dzieje z samym wskaźnikiem? Otóż nadal wskazuje on na **miejsce w pamięci**, w którym jeszcze niedawno egzystował nasz obiekt. Teraz jednak już go tam nie ma; wszelkie próby odwołania się do tego obszaru skończą się więc błędem, zwanym **naruszeniem zasad dostępu** (ang. *access violation*). Pamiętajmy zatem, iż:

Nie należy próbować uzyskać dostępu do zniszczonego (lub niestworzonego) obiektu poprzez wskaźnik na niego. Spowoduje to bowiem błąd wykonania programu i jego awaryjne zakończenie.

Musimy być także świadomi, że w momencie usuwania obiektu traci ważność nie tylko ten wskaźnik, którego użyliśmy do dokonania aktu zniszczenia, ale też **wszystkie inne**

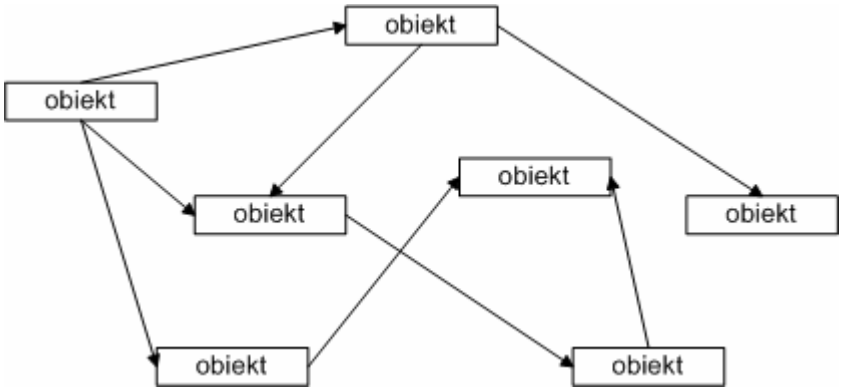
wskaźniki odnoszące się do tego obiektu! To zresztą naturalne, skoro co do jednego wskazują one na tą samą, nieaktualną już lokację w pamięci.

Stosowanie wskaźników na obiekty

Wczytując się w powyższy opis i spoglądając nań krytycznym okiem można uznać, że stosowanie wskaźników na obiekty jest tylko niepotrzebnym zawracaniam sobie głowy i utrudnianiem życia. Nie dość, że trzeba samemu dbać o tworzenie i niszczenie obiektów, to jeszcze nasz program może się niechybnie „wysypać", jeśli spróbujemy odwołać się do nieistniejącego obiektu. I gdzie są te obiecane korzyści?…
Taka ocena jest naturalnie mocno niesprawiedliwa, a moim zadaniem jest przekonanie cię, iż wskaźniki są nie tylko przydatne w programowaniu obiektowym, ale wydają się wręcz **niezbędne**.

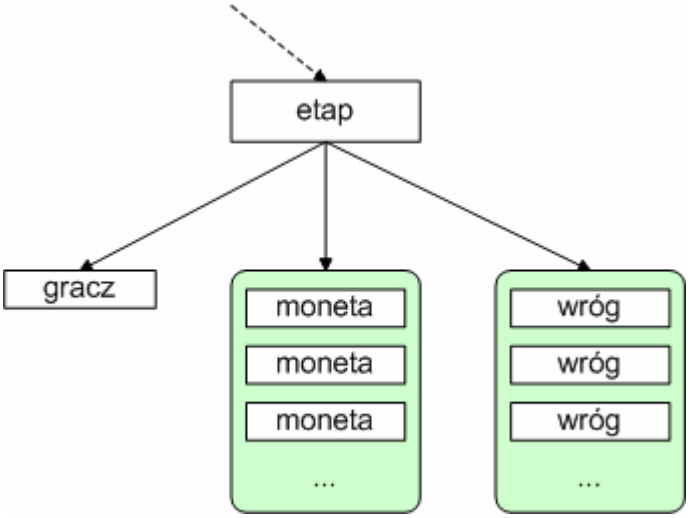
Przypomnijmy sobie najpierw, cóż ciekawego powiedzieliśmy o obiektach na samych początku rozdziału. Mianowicie wyjaśniliśmy sobie, że są to drobne cegiełki, z których programista buduje swoją aplikację.
To całkiem dobre porównanie, gdyż kryje w sobie jeszcze jeden ukryty sens: niewiele można zrobić z zestawem cegieł, jeżeli nie będziemy dysponowali jakimś **spoiwem**, łączącym je w calość. Rolę łączników spełniają właśnie wskaźniki.

Każdy obiekt, aby być użytecznym, powinien być jakoś połączony z innym obiektem. To w zasadzie dosyć oczywista prawda, jednak na początku można sobie nie całkiem zdawać z niej sprawę.
Takie relacje najprościej realizować za pomocą wskaźników. Sposób, w jaki łączą one obiekty, jest bardzo prosty: otóż jeden z nich powinien posiadać **pole, będące wskaźnikiem** na drugi obiekt. Ów drugi koniec łącza może, jak wiemy, istnieć w dowolnym miejscu pamięci, co więcej - możliwe jest, by „dochodził" do niego więcej niż jeden wskaźnik! W ten sposób obiekty mogą brać udział w dowolnej liczbie wzajemnych relacji.



Schemat 20. Działanie aplikacji opiera się na zależnościach między obiektami

Tak to wygląda w teorii, ale ponieważ jeden przykład wart jest tysiąca słów, najlepiej będzie, jeżeli przyjrzysz się takowemu przykładowi. Przypuśćmy więc, że jesteśmy w trakcie pisania gry podobnej do sławnego Lode Runnera: należy w niej zebrać wszystkie przedmioty znajdujące się na planszy (zazwyczaj są to monety albo inne bogactwa), aby awansować do kolejnego etapu. Jakie obiekty i jakie zależności należałoby w tym przypadku stworzyć?
Najlepiej zacząć od tego największego i najważniejszego, grupującego wszystkie inne - na przykład samego etapu. Podrzędnym w stosunku do niego będzie obiekt gracza oraz, rzecz jasna, pewna ilość obiektów monet (zapewne umieszczonych w tablicy albo innym tego rodzaju pojemniku). Do tego dodamy pewnie jeszcze kilku wrogów; ostatecznie nasz prosty model przedstawiać się będzie następująco:



Schemat 21. Fragment przykładowego diagramu powiązań obiektów w grze

Dzięki temu, że obiekt etapu posiadą dostęp (naturalnie poprzez wskaźnik) do obiektów gracza czy też wrogów, może chociażby uaktualniać ich pozycję na ekranie w odpowiedzi na wciskanie klawiszy na klawiaturze lub upływ czasu. Odpowiednie rozkazy będzie zapewne otrzymywał „z góry", tj. od obiektu nadrzędnego wobec niego - najprawdopodobniej jest to główny obiekt gry.

W podobny sposób, o wiele naturalniejszy niż w programowaniu strukturalnym, projektujemy model obiektowy każdego w zasadzie programu. Nie musimy już rozdzielać swoich koncepcji na dane i kod, wystarczy że stworzymy odpowiednie klasy oraz obiekty i zapewnimy powiązania między nimi. Rzecz jasna, z wykorzystaniem wskaźników na obiekty :)

Podsumowanie

Kończący się rozdział był nieco krótszy niż parę poprzednich. Podejrzewam jednak, że przebrnięcie przez niego zajęło ci może nawet więcej czasu i było o wiele trudniejsze. Wszystko dlatego że poznawaliśmy tutaj zupełnie nową koncepcję programowania, która wprawdzie ideowo jest o wiele bliższa człowiekowi niż techniki strukturalne, ale w zamian wymaga od razu przyswojenia sobie sporej porcji nowych wiadomości i pojęć. Nie martw się zatem, jeśli nie były one dla ciebie całkiem jasne; zawsze przecież możesz wrócić do trudniejszych fragmentów tekstu w przyszłości (ponowne przeczytanie całego rozdziału jest naturalnie również dopuszczalne :D).

Nasze spotkanie z programowaniem obiektowym będziemy zresztą kontynuowali w następnym rozdziale, w którym to ostatecznie wyjaśni się, dlaczego jest ono takie wspaniałe ;)

Pytania i zadania

Nowopoznane, arcyważne zagadnienie wymaga oczywiście odpowiedniego powtórzenia. Nie krępuj się więc i odpowiedz na poniższe pytania :)

Pytania

1. Czym są obiekty i jaka jest ich rola w programowaniu z użyciem technik OOP?
2. Jakie etapy obejmuje wprowadzenie do programu nowej klasy?

3. Jakie składniki możemy umieścić w definicji klasy?
4. (**Trudne**) Które składowe klasa posiada zawsze, niezależnie od tego czy je zdefiniujemy, czy nie?
5. W jaki sposób możemy z wnętrza metody uzyskać dostęp do obiektu, na rzecz którego została ona wywołana?
6. Czym różni się użycie wskaźnika na obiekt od zmiennej obiektowej?
7. Jak odrębne obiekty w programie mogą „wiedzieć” o sobie nawzajem i przekazywać między sobą informacje?

Ćwiczenia

1. Zdefiniuj prostą klasę reprezentującą książkę.
2. Napisz program podobny do przykładu `DegreesCalc`, ale przeliczający między jednostkami informacji (bajtami, kilobajtami itd.).