7

PROGRAMOWANIE OBIEKTOWE

Gdyby murarze budowali domy tak,
jak programiści piszą programy,
to jeden dzięcioł zniszczyłby całą cywilizację.
ze zbioru prawd o oprogramowaniu

Witam cię serdecznie, drogi Czytelniku! Powitanie to jest tutaj jak najbardziej wskazane. Twoja obecność wskazuje bowiem, że nadzwyczaj szybko wydostałeś się spod sterty nowych wiadomości, którymi obarczyłem cię w poprzednim rozdziale :) A nie było to wcale takie proste, zważywszy że poznałeś tam zupełnie nową technikę programowania, opierającą się na całkiem innych zasadach niż te dotychczas ci znane.

Mimo to mogłeś uczuć pewien niedosyt. Owszem, idea OOPu była tam przedstawiona jako w miarę naturalna, a nawet intuicyjna (w każdym razie bardziej niż programowanie strukturalne). Potrzeba jednak sporej dozy optymizmu, aby uznać ją na tym etapie za coś rewolucyjnego, co faktycznie zmienia sposób myślenia o programowaniu (a jednocześnie znacznie je ułatwia).
By w pełni przekonać się do tej koncepcji, trzeba o niej wiedzieć nieco więcej; kluczowe informacje na ten temat są zawarte w tym oto rozdziale. Sądzę więc, że choćby z tego powodu będzie on dla ciebie bardzo interesujący :D

Zajmiemy się w nim dwoma niezwykle ważnymi zagadnieniami programowania obiektowego: dziedziczeniem oraz metodami wirtualnymi. Na nich właśnie opiera się cała jego potęga, pozwalająca tworzyć efektowne i efektywne programy.
Zobaczymy zresztą, jak owo tworzenie wygląda w rzeczywistości. Końcową część rozdziału poświęciłem bowiem na zestaw rad i wskazówek, które, jak sądzę, okażą się pomocne w projektowaniu aplikacji opartych na modelu OOP.

Kontynuujmy zatem poznawanie wspaniałego świata programowania obiektowego :)

Dziedziczenie

Drugim powodem, dla którego techniki obiektowe zyskały taką popularność[77], jest znaczący postęp w kwestii **ponownego wykorzystywania** raz napisanego kodu oraz **rozszerzania i dostosywania** go do własnych potrzeb.
Cecha ta leży u samych podstaw OOPu: program konstruowany jako zbiór współdziałających obiektów nie jest już bowiem monolitem, ścisłym połączeniem danych i wykonywanych nań operacji. „Rozdrobniona” struktura zapewnia mu zatem **modularność**: nie jest trudno dodać do gotowej aplikacji nową funkcję czy też

[77] Pierwszym jest wspominana nie raz „naturalność” programowania, bez konieczności podziału na dane i kod.

wyodrębnić z niej jeden podsystem i użyć go w kolejnej produkcji. Ułatwia to i przyspiesza realizację kolejnych projektów.

Wszystko zależy jednak od umiejętności i doświadczenia programisty. Nawet stosując techniki obiektowe można stworzyć program, którego elementy będą ze sobą tak ściśle zespolone, że próba ich użycia w następnej aplikacji będzie przypominała wciskanie słonia do szklanej butelki.

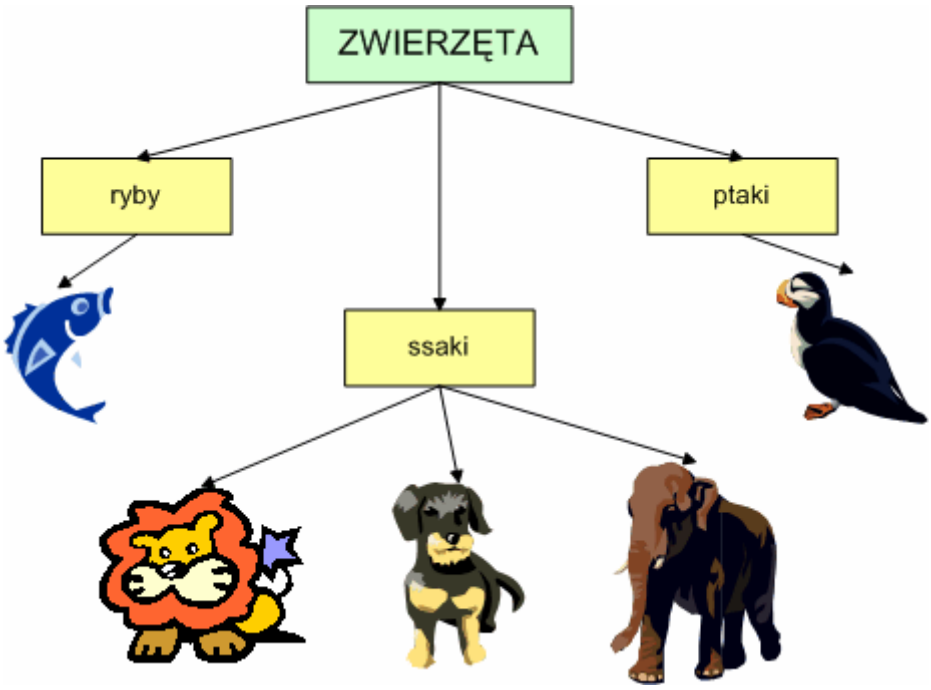
Istnieje jeszcze jedna przyczyna, dla której kod oparty na programowaniu obiektowym łatwiej poddaje się „recyklingowi", mającemu przygotować go do ponownego użycia. Jest nim właśnie tytułowy mechanizm dziedziczenia.
Korzyści płynące z jego stosowania nie ograniczają się jednakże tylko do wtórnego „przerobu" już istniejącego kodu. Przeciwnie, jest to fundamentalny aspekt OOPu niezmiernie ułatwiający i uprzyjemniający projektowanie każdej w zasadzie aplikacji. W połączeniu z technologią funkcji wirtualnych oraz polimorfizmu daje on niezwykle szerokie możliwości, o których szczegółowo traktuje praktycznie cały niniejszy rozdział.

Rozpoczniemy zatem od dokładnego opisu tego bardzo pożytecznego mechanizmu programistycznego.

O powstawaniu klas drogą doboru naturalnego

Człowiek jest taką dziwną istotą, która bardzo lubi posiadać uporządkowany i usystematyzowany obraz świata. Wprowadzanie porządku i pewnej hierarchii co do postrzeganych zjawisk i przedmiotów jest dla nas niemal naturalną potrzebą.

Chyba najlepiej przejawia się to w klasyfikacji biologicznej. Widząc na przykład psa wiemy przecież, że nie tylko należy on do gatunku zwanego psem domowym, lecz także do gromady znanej jako ssaki (wraz z końmi, słoniami, lwami, małpami, ludźmi i całą resztą tej menażerii). Te z kolei, razem z gadami, ptakami czy rybami należą do kolejnej, znacznie większej grupy organizmów zwanych po prostu zwierzętami.
Nasz pies jest zatem **jednocześnie** psem domowym, ssakiem i zwierzęciem:



Schemat 22. Klasyfikacja zwierząt jako przykład hierarchii typów obiektów

Gdyby był obiektem w programie, wtedy musiałby należeć aż do trzech klas naraz[78]! Byłoby to oczywiście niemożliwe, jeżeli wszystkie miałyby być wobec siebie równorzędne. Tutaj jednak tak nie jest: występuje między nimi hierarchia, jedna klasa pochodzi od drugiej. Zjawisko to nazywamy właśnie **dziedziczeniem**.

Dziedziczenie (ang. *inheritance*) to tworzenie nowej klasy na podstawie jednej lub kilku istniejących wcześniej klas bazowych.

Wszystkie klasy, które powstają w ten sposób (nazywamy je **pochodnymi**), posiadają pewne elementy wspólne. Części te są **dziedziczone** z klas bazowych, gdyż tam właśnie zostały zdefiniowane.
Ich zbiór może jednak zostać **poszerzony** o pola i metody specyficzne dla klas pochodnych. Będą one wtedy współistnieć z „dorobkiem" pochodzącym od klas bazowych, ale mogą oferować dodatkową funkcjonalność.

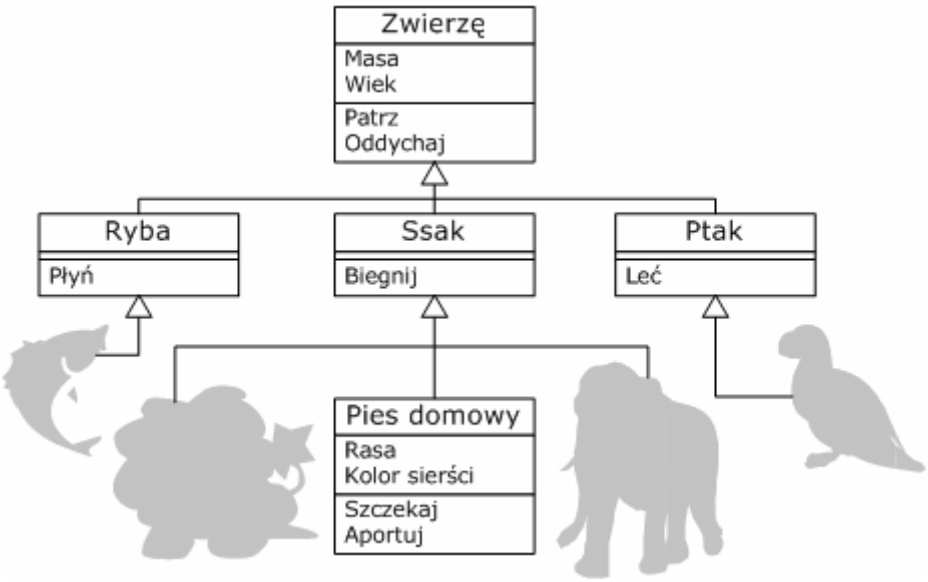
Tak w teorii wygląda system dziedziczenia w programowaniu obiektowym. Najlepiej będzie, jeżeli teraz przyjrzymy się, jak w praktyce może wyglądać jego zastosowanie.

Od prostoty do komplikacji, czyli ewolucja

Powróćmy więc do naszego przykładu ze zwierzętami. Chcąc stworzyć programowy odpowiednik zaproponowanej hierarchii, musielibyśmy zdefiniować najpierw odpowiednie **klasy bazowe**. Następnie **odziedziczylibyśmy** ich pola i metody w **klasach pochodnych** i dodali nowe, właściwe tylko im. Powstałe klasy same mogłyby być potem bazami dla kolejnych, jeszcze bardziej wyspecjalizowanych typów.
Idąc dalej tą drogą dotarlibyśmy wreszcie do takich klas, z których sensowne byłoby już tworzenie normalnych obiektów.

Pojęcie klas bazowych i klas pochodnych jest zatem **względne**: dana klasa może wprawdzie pochodzić od innych, ale jednocześnie być bazą dla kolejnych klas. W ten sposób ustala się wielopoziomowa hierarchia, podobna zwykle do drzewka.

Ilustracją tego procesu może być poniższy diagram:



Schemat 23. Hierarchia klas zwierząt

[78] A raczej do siedmiu lub ośmiu, gdyż dla prostoty pominąłem tu większość poziomów systematyki.

Wszystkie przedstawione na nim klasy wywodzą się z jednej, nadrzędnej wobec wszystkich: jest nią naturalnie klasa *Zwierzę*. Dziedziczy z niej każda z pozostałych klas - **bezpośrednio**, jak *Ryba*, *Ssak* oraz *Ptak*, lub **pośrednio** - jak *Pies domowy*.
Tak oto tworzy się kilkupoziomowa klasyfikacja oparta na mechanizmie dziedziczenia.

Z klasy bazowej do pochodnej, czyli dziedzictwo przodków

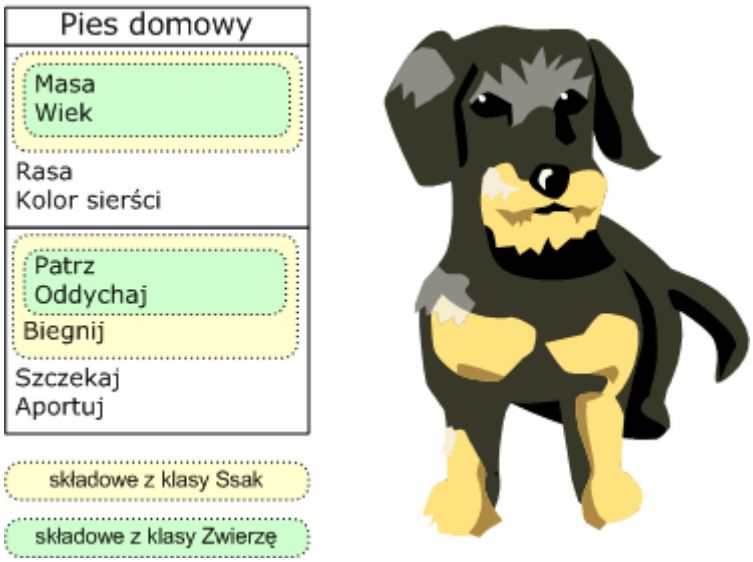
O podstawowej konsekwencji takiego rozwiązania zdążyłem już wcześniej wspomnieć. Jest nią mianowicie **przekazywanie** pól oraz metod pochodzących z klasy bazowej do wszystkich klas pochodnych, które się z niej wywodzą. Zatem:

Klasa pochodna zawiera pola i metody **odziedziczone** po klasach bazowych. Może także posiadać dodatkowe, unikalne dla siebie składowe - nie jest to jednak obowiązkiem.

Prześledźmy teraz sposób, w jaki odbywa się odziedziczanie składowych na przykładzie naszej prostej hierarchii klas zwierząt.

U jej podstawy leży „najbardziej bazowa" klasa *Zwierzę*. Zawiera ona dwa pola, określające masę i wiek zwierzęcia, oraz metody odpowiadające za takie czynności jak widzenie i oddychanie. Składowe te mogły zostać umieszczone tutaj, gdyż dotyczą one wszystkich interesujących nas zwierząt i będą **miały sens** w każdej z klas pochodnych.
Tymi klasami, bezpośrednio dziedziczącymi od klasy *Zwierzę*, są *Ryba*, *Ssak* oraz *Ptak*. Każda z nich niejako „z miejsca" **otrzymuje** zestaw pól i metod, którymi legitymowało się bazowe *Zwierzę*. Klasy te wprowadzają jednak także dodatkowe, własne metody: i tak *Ryba* może pływać, *Ssak* biegać[79], zaś *Ptak* latać. Nie ma w tym nic dziwnego, nieprawdaż? :)
Wreszcie, z klasy *Ssak* dziedziczy najbardziej interesująca nas klasa, czyli *Pies domowy*. Przejmuje ona wszystkie pola i metody z klasy *Ssak*, a więc pośrednio także z klasy *Zwierzę*. Uzupełnia je przy tym o kolejne składowe, właściwe tylko sobie.

Ostatecznie więc klasa *Pies domowy* zawiera znacznie więcej pól i metod niż mogłoby się z początku wydawać:



Schemat 24. Składowe klasy *Pies domowy*

[79] Delfiny muszą mi wybaczyć nieuwzględnienie ich w tym przykładzie :D

Wykazuje poza tym pewną budowę wewnętrzną: niektóre jej pola i metody możemy bowiem określić jako własne i unikalne, zaś inne są odziedziczone po klasie bazowej i mogą być wspólne dla wielu klas. Nie sprawia to jednak żadnej różnicy w korzystaniu z nich: funkcjonują one identycznie, jakby były zawarte bezpośrednio wewnątrz klasy.

Obiekt o kilku klasach, czyli zmienność gatunkowa

Oczywiście klas nie definiuje się dla samej przyjemności ich definiowania, lecz dla tworzenia z nich obiektów. Jeżeli więc posiadalibyśmy przedstawioną wyżej hierarchię w jakimś prawdziwym programie, to z pewnością pojawiłyby się w nim także instancje zaprezentowanych klas, czyli odpowiednie obiekty.

W ten sposób wracamy do problemu postawionego na samym początku: jak obiekt może należeć do kilku klas naraz? Różnica polega wszak na tym, że mamy już jego gotowe rozwiązanie :) Otóż nasz obiekt psa należałby **przede wszystkim** do klasy *Pies domowy*; to właśnie tej nazwy użylibyśmy, by zadeklarować reprezentującą go zmienną czy też pokazujący nań wskaźnik. Jednocześnie jednak byłby on typu *Ssak* oraz typu *Zwierzę*, i mógłby występować w tych miejscach programu, w których byłby wymagany jeden z owych typów.
Fakt ten jest przyczyną istnienia w programowaniu obiektowym zjawiska zwanego polimorfizmem. Poznamy je dokładnie jeszcze w tym rozdziale.

Dziedziczenie w C++

Pozyskawszy ogólne informacje o dziedziczeniu jako takim, możemy zobaczyć, jak idea ta została przełożona na nasz nieoceniony język C++ :) Dowiemy się więc, w jaki sposób definiujemy nowe klasy w oparciu o już istniejące oraz jakie dodatkowe efekty są z tym związane.

Podstawy

Mechanizm dziedziczenia jest w C++ bardzo rozbudowany, o wiele bardziej niż w większości pozostalych języków zorientowanych obiektowo[80]. Udostępnia on kilka szczególnych możliwości, które być może nie są zawsze niezbędne, ale pozwalają na dużą swobodę w definiowaniu hierarchii klas. Poznanie ich wszystkich nie jest konieczne, aby sprawnie korzystać z dobrodziejstw programowania obiektowego, jednak wiemy doskonale, że wiedza jeszcze nikomu nie zaszkodziła :D

Zaczniemy oczywiście od najbardziej elementarnych zasad dziedziczenia klas oraz przyjrzymy się przykładom ilustrującym ich wykorzystanie.

Definicja klasy bazowej i specyfikator `protected`

Jak pamiętamy, definicja klasy składa się przede wszystkim z listy deklaracji jej pól oraz metod, podzielonych na kilka części wedle specyfikatorów praw dostępu. Najczęściej każdy z tych specyfikatorów występuje co najwyżej w jednym egzemplarzu, przez co składnia definicji klasy wygląda następująco:

```
class nazwa_klasy
{
    [private:]
          [deklaracje_prywatne]
    [protected:]
          [deklaracje_chronione]
    [public:]
```

[80] Dorównują mu chyba tylko rozwiązania znane z Javy.

```
        [deklaracje_publiczne]
};
```

Nieprzypadkowo pojawił się tu nowy specyfikator, `protected`. Jego wprowadzenie związane jest ściśle z pojęciem dziedziczenia. Pojęcie to wpływa zresztą na dwa pozostałe rodzaje praw dostępu do składowych klasy.
Zbierzmy więc je wszystkie w jednym miejscu, wyjaśniając definitywnie znaczenie każdej z etykiet:

- `private`: poprzedza deklaracje składowych, które mają być dostępne **jedynie** dla metod definiowanej klasy. Oznacza to, iż nie można się do nich dostać, używając obiektu lub wskaźnika na niego oraz operatorów wyłuskania . lub ->.
  Ta wyłączność znaczy również, że prywatne składowe **nie są dziedziczone** i nie ma do nich dostępu w klasach pochodnych, gdyż nie wchodzą w ich skład.
- specyfikator `protected` („chronione") także nie pozwala, by użytkownicy obiektów naszej klasy „grzebali" w opatrzonych nimi polach i metodach. Jak sama nazwa wskazuje, są one **chronione** przed takim dostępem z zewnątrz.
  Jednak w przeciwieństwie do deklaracji `private`, składowe zaznaczone przez `protected` **są dziedziczone** i występują w klasach pochodnych, będąc dostępnymi dla ich własnych metod.

Pamiętajmy zatem, że zarówno `private`, jak i `protected` **nie pozwala**, aby oznaczone nimi składowe klasy były dostępne **na zewnątrz**. Ten drugi specyfikator zezwala jednak na dziedziczenie pól i metod.

- `public` jest najbardziej liberalnym specyfikatorem. Nie tylko pozwala na odziedziczanie swych składowych, ale także na udostępnianie ich szerokiej rzeszy obiektów poprzez operatory wyłuskania.

Powyższe opisy brzmią może nieco sucho i niestrawnie, dlatego przyjrzymy się jakiemuś przykładowi, który będzie bardziej przemawiał do wyobraźni. Mamy więc taką oto klasę prostokąta:

```
class CRectangle
{
   private:
         // wymiary prostokąta
         float m_fSzerokosc, m_fWysokosc;
   protected:
         // pozycja na ekranie
         float m_fX, m_fY;
   public:
         // konstruktor
         CRectangle()      { m_fX = m_fY = 0.0;
                             m_fSzerokosc = m_fWysokosc = 10.0; }

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

         // metody
         float Pole() const   { return m_fSzerokosc * m_fWysokosc; }
         float Obwod() const  { return 2 * (m_fSzerokosc+m_fWysokosc); }
};
```

Opisują go cztery liczby, wyznaczające jego pozycję oraz wymiary. Współrzędne X oraz Y uczyniłem tutaj polami chronionymi, zaś szerokość oraz wysokość - prywatnymi.
Dlaczego właśnie tak?…
Otóż powyższa klasa będzie również bazą dla następnej. Pamiętamy z geometrii, że szczególnym rodzajem prostokąta jest kwadrat. Ma on wszystkie boki o tej samej długości, zatem nielogiczne jest stosowań do nich pojęcia szerokości i wysokości.

Wielkość kwadratu określa bowiem tylko jedna liczba, więc defincja odpowiadającej mu klasy może wyglądać następująco:

```
class CSquare : public CRectangle      // dziedziczenie z CRectangle
{
   private:
         // zamiast szerokości i wysokości mamy tylko długość boku
         float m_fDlugoscBoku;

   // pola m_fX i m_fY są dziedziczone z klasy bazowej, więc nie ma
   // potrzeby ich powtórnego deklarowania

   public:
         // konstruktor
         CSquare { m_fDlugoscBoku = 10.0; }

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

         // nowe metody
         float Pole() const  { return m_fDlugoscBoku * m_fDlugoscBoku; }
         float Obwod() const { return 4 * m_fDlugoscBoku; }
};
```

Dziedziczy ona z `CRectangle`, co zostało zaznaczone w pierwszej linijce, ale postać tej frazy chwilowo nas nie interesuje :) Skoncentrujmy się raczej na konsekwencjach owego dziedziczenia.
Porozmawiajmy najpierw o nieobecnych. Pola `m_fSzerokosc` oraz `m_fWysokosc` były w klasie bazowej oznaczone jako prywatne, zatem ich zasięg ogranicza się jedynie do tej klasy. W pochodnej `CSquare` nie ma już po nich śladu; zamiast tego pojawia się bardziej naturalne pole `m_fDlugoscBoku` z sensowną dla kwadratu wielkością.
Związane są z nią także dwie nowe-stare metody, zastępujące te z `CRectangle`. Do obliczania pola i obwodu wykorzystujemy bowiem samą długość boku kwadratu, nie zaś „jego" szerokośc i wysokość, których w klasie w ogóle nie ma.

W definicji `CSquare` nie ma także deklaracji `m_fX` oraz `m_fY`. Nie znaczy to jednak, że klasa tych pól nie posiada, gdyż zostały one po prostu **odziedziczone** z bazowej `CRectangle`. Stało się tak oczywiście za sprawą specyfikatora `protected`.
Co więc powinniśmy o nim pamiętać? Otóż:

Należy używać specyfikatora `protected`, kiedy chcemy uchronić składowe przed dostępem z zewnątrz, ale jednocześnie mieć je do dyspozycji w klasach pochodnych.

Definicja klasy pochodnej

Dopiero posiadając zdefiniowaną klasę bazową możemy przystąpić do określania dziedziczącej z niej klasy pochodnej. Jest to konieczne, bo w przeciwnym wypadku kazalibyśmy kompilatorowi korzystać z czegoś, o czym nie miałby wystarczających informacji.

Składnię definicji klasy pochodnej możemy poglądowo przedstawić w ten sposób:

```
class nazwa_klasy [: [specyfikatory] [nazwa_klasy_bazowej] [, ...]]
{
   deklaracje_składowych
};
```

Ponieważ z sekwencją *deklaracji_składowych* spotkaliśmy się już nie raz i nie dwa razy, skupimy się jedynie na pierwszej linijce podanego schematu.

To w niej właśnie podajemy klasy bazowe, z których chcemy dziedziczyć. Czynimy to, wpisując dwukropek po nazwie definiowanej właśnie klasy i podając dalej listę jej klas bazowych, oddzielonych przecinkami. Zwykle nie będzie ona zbyt długa, gdyż w większości przypadków wystarczające jest pojedyncze dziedziczenie, zakładające tylko jedną klasę bazową.

Istotne są natomiast kolejne *specyfikatory*, które opcjonalnie możemy umieścić przed każdą *nazwą_klasy_bazowej*. Wpływają one na proces dziedziczenia, a dokładniej na **prawa dostępu**, na jakich klasa pochodna otrzymuje składowe klasy bazowej.
Kiedy zaś mowa o tychże prawach, natychmiast przypominamy sobie o słówkach `private`, `protected` i `public`, nieprawdaż? ;) Rzeczywiście, *specyfikatory* dziedziczenia występują zasadniczo w liczbie trzech sztuk i są identyczne z tymi występującymi wewnątrz bloku klasy. O ile jednak tamte pojawiają się w prawie każdej sytuacji i klasie, o tyle tutaj specyfikator `public` ma niemal całkowity monopol, a użycie pozostałych dwóch należy do niezmiernie rzadkich wyjątków.
Dlaczego tak jest? Otóż w 99.9% przypadków nie ma najmniejszej potrzeby zmiany praw dostępu do składowych odziedziczonych po klasie bazowej. Jeżeli więc któreś z nich zostały tam zadeklarowane jako `protected`, a inne jako `public`, to prawie zawsze życzymy sobie, aby w klasie pochodnej zachowały te same prawa. Zastosowanie dziedziczenia `public` czyni zadość tym żądaniom, dlatego właśnie jest ono tak często stosowane.

O pozostałych dwóch specyfikatorach możesz przeczytać w MSDN. Generalnie ich działanie nie jest specjalnie skomplikowane, gdyż nadają składowym klasy bazowej prawa dostępu właściwe swoim „etykietowym" odpowiednikom. Tak więc dziedziczenie `protected` czyni wszystkie składowe klasy bazowej chronionymi w klasie pochodnej, zaś `private` sprowadza je do dostępu prywatnego.

Formalnie rzecz ujmując, stosowanie specyfikatorów dziedziczenia jest nieobowiązkowe. W praktyce jednak trudno korzystać z tego faktu, ponieważ pominięcie ich jest równoznacznie z zastosowaniem specyfikatora `private`[81] - nie zaś naturalnego `public`! Niestety, ale tak właśnie jest i trzeba się z tym pogodzić.

Nie zapominaj więc o specyfikatorze `public`, gdyż jego brak przed nazwą klasy bazowej jest niemal na pewno błędem.

Dziedziczenie pojedyncze

Najprostszą i jednocześnie najczęściej występującą w dziedziczeniu sytuacją jest ta, w której mamy do czynienia tylko z **jedną klasa bazową**. Wszystkie dotychczas pokazane przykłady reprezentowały to zagadnienie; nazywamy je **dziedziczeniem pojedynczym** lub **jednokrotnym** (ang. *single inheritance*).

Proste przypadki

Najprostsze sytuacje, w których mamy do czynienia z tym rodzajem dziedziczenia, są często spotykane w programach. Polegają one na tym, iż jedna klasa jest tworzona na podstawie drugiej poprzez zwyczajne rozszerzenie zbioru pól i metod.
Ilustracją będzie tu kolejny przykład geometryczny :)

```
class CEllipse       // elipsa, klasa bazowa
{
```

[81] Zakładając, że mówimy o klasach deklaroanych poprzez słowo `class`. W przypadku struktur (słowo `struct`), które są w C++ niemal tożsame z klasami, to `public` jest domyślnym specyfikatorem - zarówno dziedziczenia, jak i dostępu do składowych.

```
    private:
          // większy i mniejszy promień elipsy
          float m_fWiekszyPromien;
          float m_fMniejszyPromien;
    protected:
          // współrzędne na ekranie
          float m_fX, m_fY;
    public:
          // konstruktor
          CEllipse()  { m_fX = m_fY = 0.0;
                        m_fWiekszyPromien = m_fMniejszyPromien = 10.0; }

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

          // metody
          float Pole() const
                { return PI * m_fWiekszyPromien * m_fMniejszyPromien; }
};

class CCircle : public CEllipse        // koło, klasa pochodna
{
    private:
          // promień koła
          float m_fPromien;
    public:
          // konstruktor
          CCircle()   ( m_fPromien = 10.0; }

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

          // metody
          float Pole() const   { return PI * m_fPromien * m_fPromien; }
          float Obwod() const  { return 2 * PI * m_fPromien; }
};
```

Jest on podobny do wariantu z prostokątem i kwadratem. Tutaj klasa `CCircle` jest pochodną od `CEllipse`, zatem dziedziczy wszystkie jej składowe, które nie są prywatne. Uzupełnia ponadto ich zbiór o dodatkową metodę `Obwod()`, obliczającą długość okręgu okalającego nasze koło.

Sztafeta pokoleń

Hierarchia klas nierzadko nie kończy się na jednej klasie pochodnej, lecz sięga nawet bardziej wgłąb. Nowo stworzona klasa może być bowiem bazową dla kolejnych, te zaś - dla następnych, itd.

Na samym początku spotkaliśmy się zresztą z takim przypadkiem, gdzie klasami były rodzaje zwierząt. Spróbujemy teraz przełożyć tamten układ na język C++.
Zaczynamy oczywiście od klasy, z której wszystkie inne biorą swój początek - `CAnimal`:

```
class CAnimal        // Zwierzę
{
    protected:
          // pola klasy
          float m_fMasa;
          unsigned m_uWiek;
    public:
          // konstruktor
          CAnimal()   { m_uWiek = 0; }
```

```
        //-------------------------------------------------------------

        // metody
        void Patrz();
        void Oddychaj();

        // metody dostępowe do pól
        float Masa() const      { return m_fMasa; }
        void Masa(float fMasa)  { m_fMasa = fMasa; }
        unsigned Wiek() const   { return m_uWiek; }
};
```

Jej postać nie jest chyba niespodzianką: mamy tutaj wszystkie ustalone wcześniej, publiczne metody oraz pola, które oznaczyliśmy jako `protected`. Zrobiliśmy tak, bo chcemy, by były one przekazywane do klas pochodnych od `CAnimal`.

A skoro już wspomnialiśmy o klasach pochodnych, pomyślmy o ich definicjach. Zważywszy, że każda z nich wprowadza tylko jedną nową metodę, powinny one być raczej proste - i istotnie takie są:

```
class CFish : public CAnimal     // Ryba
{
   public:
        void Plyn();
};

class CMammal : public CAnimal   // Ssak
{
   public:
        void Biegnij();
};

class CBird : public CAnimal     // Ptak
{
   public:
        void Lec();
};
```

Nie zapominamy rzecz jasna, że oprócz widocznych powyżej deklaracji zawierają one także wszystkie składowe wzięte od klasy `CAnimal`. Powtarzam to tak często, że chyba nie masz już co do tego żadnych wątpliwości :D

Ostatnią klasą z naszego drzewa gatunkowego był, jak pamiętamy, *Pies domowy*. Definicja jego klasy także jest dosyć prosta:

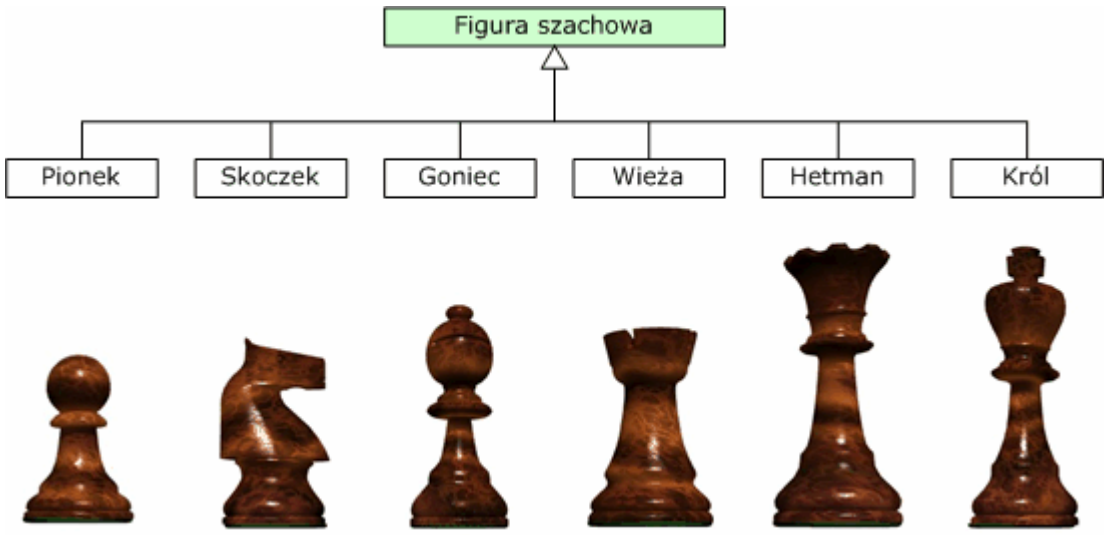
```
class CHomeDog : public CMammal  // Pies domowy
{
   protected:
        // nowe pola
        RACE m_Rasa;
        COLOR m_KolorSiersci;
   public:
        // metody
        void Aportuj();
        void Szczekaj();

        // metody dostępowe do pól
        RACE Rasa() const            { return m_Rasa; }
        COLOR KolorSiersci() const   { return m_KolorSiersci; }
};
```

Jak zwykle typy `RACE` i `COLOR` są mocno umowne. Ten pierwszy byłby zapewne odpowiednim `enum`'em.

Wiemy jednakże, iż kryje się za nią całe bogactwo pól i metod odziedziczonych po klasach bazowych. Dotyczy to zarówno **bezpośredniego** przodka klasy `CHomeDog`, czyli `CMammal`, jak i jej **pośredniej** bazy - `CAnimal`. Jedyną znacząca tutaj różnicą pomiędzy tymi dwoma klasami jest fakt, że pierwsza występuje w definicji `CHomeDog`, zaś druga nie.

Płaskie hierarchie

Oprócz rozbudowanych, wielopoziomowych relacji typu baza-pochodna w powszechnym zastosowaniu są też takie modele, w których z jednej klasy bazowej dziedziczy wiele klas pochodnych. Jest to tzw. **płaska hierarchia** i wygląda np. w ten sposób:



Schemat 25. Płaska hierarchia klas figur szachowych (ilustracje pochodzą z serwisu David Howell Chess)

Po przełożeniu jej na język C++ otrzymalibyśmy coś w tym rodzaju:

```
// klasa bazowa
class CChessPiece { /* definicja */ };              // Figura szachowa

// klasy pochodne
class CPawn : public CChessPiece { /* ... */ };      // Pionek
class CKnight : public CChessPiece { /* ... */ };    // Skoczek[82]
class CBishop : public CChessPiece { /* ... */ };    // Goniec
class CRook : public CChessPiece { /* ... */ };      // Wieża
class CQueen : public CChessPiece { /* ... */ };     // Hetman
class CKing : public CChessPiece { /* ... */ };      // Król
```

Oprócz logicznego uporządkowania rozwiązanie to ma też inne zalety. Jeśli bowiem zadeklarowalibyśmy wskaźnik na obiekt klasy `CChessPiece`, to poprzez niego moglibyśmy odwoływać się do obiektów krórejkolwiek z klas pochodnych. Jest to jedna z licznych pozytywnych konsekwencji polimorfizmu, które zresztą poznamy wkrótce. W tym przypadku oznaczałaby ona, że za obsługę każdej z **sześciu** figur szachowych odpowiadałby najprawdopodobniej **jeden** i ten sam kod.

[82] Nazwy klas nie są tłumaczeniami z języka polskiego, lecz po prostu angielskimi nazwami figur szachowych.

Można zauważyć, ze bazowa klasa `CChessPiece` nie będzie tutaj służyć do tworzenia obiektów, lecz tylko do wyprowadzania z niej kolejnych klas. Sprawia to, że byłaby ona dobrym kandydatem na tzw. klasę abstrakcyjną. O tym zagadnieniu będziemy mówić przy okazji metod wirtualnych.

Podsumowanie

Myślę, że po takiej ilości przykładów oraz opisów koncepcja tworzenia klas pochodnych poprzez dziedziczenie powinna być ci już doskonale znana :) Nie należy ona wszakże do trudnych; ważne jest jednak, by poznać związane z nią niuanse w języku C++.

O dziedziczeniu pojedynczym można także poczytać nieco w MSDN.

Dziedziczenie wielokrotne

Skoro możliwe jest dziedziczenie z wykorzystaniem jednej klasy bazowej, to raczej naturalne jest rozszerzenie tego zjawiska także na przypadki, w której z **kilku klas bazowych** tworzymy jedną klasę pochodną. Mówimy wtedy o **dziedziczeniu wielokrotnym** (ang. *multiple inheritance*).

C++ jest jednym z niewielu języków, które udostępniają taką możliwość. Nie świadczy to jednak o jego niebotycznej wyższości nad nimi. Tak naprawdę technika dziedziczenia wielokrotnego nie daje żadnych nadzwyczajnych korzyści, a jej użycie jest przy tym dość skomplikowane. Decydując się na jej wykorzystanie należy więc posiadać całkiem spore doświadczenie w programowaniu.
Jakkolwiek zatem dziedziczenie wielokrotne bywa czasem przydatnym narzędziem, stosowanie go (przynajmniej powszechne) w tworzonych aplikacjach **nie jest zalecane**. Jeżeli pojawia się taka konieczność, należy wtedy najprawdopodobniej zweryfikować swój projekt; w większości sytuacji te same, a nawet lepsze efekty można osiągnąć nie korzystając z tego wielce wątpliwego rozwiązania.

Dla szczególnie zainteresowanych i odważnych istnieje oczywiście opis w MSDN.

Pułapki dziedziczenia

Chociaż idea dziedziczenia jest teoretycznie całkiem prosta do zrozumienia, jej praktyczne zastosowanie może niekiedy nastręczać pewnych problemów. Są one zazwyczaj specyficzne dla konkretnego języka programowania, jako że występują w tym względzie pewne różnice między nimi.

W tym paragrafie zajmiemy się takimi właśnie drobnymi niuansami, które są związane z dziedziczeniem klas w języku C++. Sekcja ta ma raczej charakter formalnego uzupełnienia, dlatego początkujący programiści mogą ją ze spokojem pominąć - szczególnie podczas pierwszego kontaktu z tekstem.

Co nie jest dziedziczone?

Wydawałoby się, że klasa pochodna powinna przejmować wszystkie składowe pochodzące z klasy bazowej - oczywiście z wyjątkiem tych oznaczonych jako `private`. Tak jednak nie jest, gdyż w trzech przypadkach nie miałoby to sensu. Owe trzy „nieprzechodnie” składniki klas to:

- **konstruktory**. Zadaniem konstruktora jest zazwyczaj inicjalizacja pól klasy na ich początkowe wartości, stworzenie wewnętrznych obiektów czy też alokacja dodatkowej pamięci. Czynności te prawie zawsze wymagają zatem dostępu do prywatnych pól klasy. Jeżeli więc konstruktor z klasy bazowej zostałby „wrzucony” do klasy pochodnej, to utraciłby z nimi niezbędne połączenie - wszak „zostałyby” one w klasie bazowej! Z tego też powodu konstruktory nie są dziedziczone.

- **destruktory**. Sprawa wygląda tu podobnie jak punkt wyżej. Działanie destruktorów najczęściej także opiera się na polach prywatnych, a skoro one nie są dziedziczone, zatem destruktor też nie powinien przechodzić do klas pochodnych.

Dość ciekawym uzasadnieniem niedziedziczenia konstruktorów i destruktorów są także same ich nazwy, odpowiadające klasie, w której zostały zadeklarowane. Gdyby zatem przekazać je klasom pochodnych, wtedy zasada ich nazewnictwa zostałaby złamana. Chociaż trudno odmówić temu podejściu pomysłowości, nie ma żadnego powodu, by uznać je za błędne.

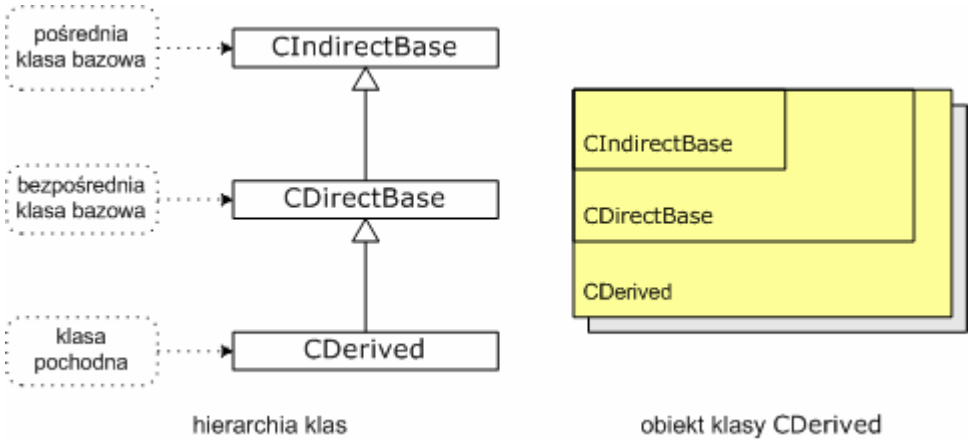
- **przeciążony operator przypisania (=)**. Zagadnienie przeciążania operatorów omówimy dokładnie w jednym z przyszłych rozdziałów. Na razie zapamiętaj, że składowa ta odpowiada za sposób, w jaki obiekt jest kopiowany z jednej zmiennej do drugiej. Taki transfer zazwyczaj również wymaga dostępu do pól prywatnych klasy, co od razu wyklucza dziedziczenie.

Ze względu na specjalne znaczenie konstruktorów i destruktorów, ich funkcjonowanie w warunkach dziedziczenia jest dość specyficzne. Nieco dalej zostało ono bliżej opisane.

Obiekty kompozytowe

Sposób, w jaki C++ realizuje pomysł dziedziczenia, jest sam w sobie dosyć interesujący. Większość koderów uczących się tego języka z początku całkiem logicznie przypusza, że kompilator zwyczajnie pobiera deklaracje z klasy bazowej i wstawia je do pochodnej, ewentualne powtórzenia rozwiązując na korzyść tej drugiej.
Swego czasu też tak myślałem i, niestety, myliłem się: faktyczna prawda jest bowiem nieco bardziej zakręcona :)

Otóż wewnętrznie używana przez kompilator definicja klasy pochodnej jest **identyczna** z tą, którą wpisujemy do kodu; nie zawiera żadnych pól i metod pochodzących z klas bazowych! Jakim więc cudem są one dostępne?
Odpowiedź jest raczej zaskakująca: podczas tworzenia obiektu klasy pochodnej dokonywana jest także kreacja obiektu klasy bazowej, który staje się jego **częścią**. Zatem nasz obiekt pochodny to tak naprawdę obiekt bazowy plus dodatkowe pola, zdefiniowane w jego własnej klasie. Przy bardziej rozbudowanej hierarchii klas zaczyna on przypominać cebulę:



Schemat 26. Obiekt klasy pochodnej zawiera w sobie obiekty klas bazowych

Praktyczne konsekwencje tego stanu rzeczy są związane chociażby z konstruowaniem i niszczeniem tych wewnętrznych obiektów.

W C++ obowiązuje zasada, iż najpierw wywoływany jest konstruktor „najbardziej bazowej" klasy danego obiektu, a potem te stojące kolejno niżej w hierarchii. Ponieważ klasa może posiadać więcej niż jeden konstruktor, kompilator musiałby podjąć decyzję, który z nich powinien zostać użyty. Nie robi tego jednak, lecz oczekuje, że zawsze[83] będzie obecny domyślny konstruktor bezparametrowy.

Dlatego też każda klasa, z której będą dziedziczyły inne, powinna posiadać taki właśnie bezparametrowy (domyślny) konstruktor.

Podobny problem nie istnieje dla destruktorów, gdyż one nigdy nie posiadają parametrów. Podczas niszczenia obiektu są one wywoływane w kolejności od tego z klasy pochodnej do tych z klas bazowych.

***

Kończący się podrozdział opisywał mechanizm dziedziczenia - jedną z podstaw techniki programowania zorientowanego obiektowego. Mogłeś więc dowiedzieć się, w jaki sposób tworzyć nowe klasy na podstawie już istniejących i projektować ich hierarchie, obrazujące naturalne związki typu „ogół-szczegół".

W następnej kolejności poznamy zalety metod wirtualnych oraz porozmawiamy sobie o największym osiągnięciu OOPu, czyli polimorfizmie. Będzie więc bardzo ciekawie :D

Metody wirtualne i polimorfizm

Dziedziczenie jest oczywiście niezwykle ważnym, a wręcz niezbędnym skadnikiem programowania obiektowego. Stanowi jednak tylko podstawę dla dwóch kolejnych technik, mających dużo większe znaczenie i pozwalających na o wiele efektywniejsze pisanie kodu. Mam tu na myśli tytułowe metody wirtualne oraz częściowo bazujący na nich polimorfizm. Wszystkie te dziwne terminy zostaną wkrótce wyjaśnione, zatem nie wpadajmy zbyt pochopnie w panikę ;)

Wirtualne funkcje składowe

Idea dziedziczenia w znanej nam dotąd postaci jest nastawiona przede wszystkim na uzupełnianie definicji klas bazowych o kolejne składowe w klasach pochodnych. Tylko czasami zastępowaliśmy już istniejące metody ich **nowymi wersjami**, właściwymi dla tworzonych klas.
Takie sytuacje są jednak w praktyce dosyć częste - albo raczej korzystne jest **prowokowanie** takich sytuacji, gdyż niejednokrotnie dają one świetne rezultaty i niespotykane wcześniej możliwości przy niewielkim nakładzie pracy. Oczywiście dzieje się tak tylko wtedy, gdy mamy odpowiednie podejście do sprawy…

To samo, ale inaczej

Raz jeszcze zajmijmy się naszą hierarchią klas zwierząt. Tym razem skierujemy uwagę na metodę *Oddychaj* z klasy *Zwierzę*.

Jej obecność u szczytu diagramu, w klasie, z której początek biorą wszystkie inne, jest z pewnością uzasadniona. Każde zwierzę, niezależnie od gatunku, musi przecież pobierać z otoczenia tlen niezbędny do życia, a proces ten nazywamy potocznie właśnie oddychaniem. Jest to bezdyskusyjne.

[83] Konieczność tę można obejść stosując tzw. listy inicjalizacyjne, o których dowiesz się za jakiś czas.

Mniej oczywisty jest natomiast fakt, że „techniczny” przebieg tej czynności może się zasadniczo różnić u poszczególnych zwierząt. Te żyjące na lądzie używają do tego narządów zwanych płucami, zaś zwierzęta wodne - chociażby ryby - mają w tym celu wykształcone skrzela, funkcjonujące na zupełnie innej zasadzie.

Spostrzeżenia te nietrudno przełożyć na bliższy nam sposób myślenia, związany bezpośrednio z programowaniem. Oto więc klasy wywodzące się do *Zwierzęcia* powinny w inny sposób implementować metodę *Oddychaj*; jej treść musi być odmienna przynajmniej dla *Ryb*y, a i *Ssak* oraz *Gad* mają przecież własne patenty na proces oddychania.
Rzeczona metoda podpada zatem pod **redefinicję** w każdej z klas dziedziczących od klasy *Zwierzę*:

Schemat 27. Przedefiniowanie metody z klasy bazowej w klasach pochodnych

Deklaracja metody wirtualnej

Teoretycznie klasa *Zwierzę* mogłaby być całkowicie „nieświadoma” tego, że jedna z jej metod jest definiowana w inny sposób w klasie pochodnej. Lepiej jednak, abyśmy przewidzieli taką konieczność i poczynili odpowiedni krok. Jest nim uczynienie funkcji *Oddychaj* metodą wirtualną w klasie *Zwierzę*.

Metoda wirtualna jest przygotowana na zastąpienie siebie przez nową wersję, zdefiniowaną w klasie pochodnej.

Aby daną funkcję składową zadeklarować jako wirtualną, należy poprzedzić jej prototyp słowem kluczowym `virtual`:

```
#include <iostream>

class CAnimal
{
   // (pomijamy pozostałe składowe klasy)

   public:
         virtual void Oddychaj()
               { std::cout << "Oddycham..." << std::endl; }
};
```

W ten sposób przygotowujemy ją na ewentualne ustąpienie miejsca bardziej wyspecjalizowanym wersjom, podanym w klasach pochodnych. Skorzystanie z mechanizmu metod wirtualnych jest tutaj lepszym rozwiązaniem niż zignorowanie go, gdyż uaktywnia to możliwości polimorfizmu związane z obiektami. Zapoznamy się z nimi w dalszej części tekstu.

Przedefiniowanie metody wirtualnej

Celem wprowadzenia funkcji wirtualnej `Oddychaj()` do klasy `CAnimal` było, jak to zaznaczyliśmy na początku, jej późniejsze **przedefiniowanie** (ang. *override*) w klasach pochodnych. Operacji tej dokonujemy prostą drogą, bowiem zwyczajnie definiujemy nową wersję metody w owych klasach:

```
class CFish : public CAnimal
{
   public:
         void Oddychaj()   // redefinicja metody wirtualnej
               { std::cout << "Oddycham skrzelami..." << std::endl; }
         void Plyn();
};

class CMammal : public CAnimal
{
   public:
         void Oddychaj()   // jak wyżej
               { std::cout << "Oddycham płucami..." << std::endl; }
         void Biegnij();
};

class CBird : public CAnimal
{
   public:
         void Oddychaj()   // i znowu jak wyżej :)
               { std::cout << "Oddycham płucami..." << std::endl; }
         void Lec();
};
```

Kompilator sam „domyśla się”, że nasza metody jest tak naprawdę redefinicją metody wirtualnej z klasy bazowej. Możemy jednak wyraźnie to zaznaczyć poprzez ponowne zastosowanie słowa `virtual`.

Według mnie jest to mało szczęśliwe rozwiązanie składniowe, ponieważ może często powodować pomyłki. Nie sposób bowiem odróżnić deklaracji przedefiniowanej metody wirtualnej od jej pierwotnej wersji (jeżeli jeszcze raz użyliśmy `virtual`) lub od zwykłej funkcji składowej (gdy nie skorzystaliśmy ze wspomnianego słówka).
Bardziej przejrzyście rozwiązano to na przykład w Delphi, gdzie nową wersję metody wirtualnej trzeba opatrzyć frazą **`override;`**.

Nowa wersja metody całkowicie zastępuje starą, która jest jednak dostępna i w razie potrzeby możemy ją wywołać. Służy do tego konstrukcja:

```
nazwa_klasy_bazowej::nazwa_metody([parametry]);
```

W powyższym przypadku byłoby to wywołanie `CAnimal::Oddychaj()`.

W Visual C++ zamiast *`nazwy_klasy_bazowej`* możliwe jest użycie specjalnego słowa kluczowego `__super`, opisanego tutaj.

Pojedynek: metody wirtualne przeciwko zwykłym

Czytając powyższe objaśnienie metod wirtualnych, zadawałeś sobie zapewne proste pytanie o głębokiej treści, a mianowicie: „Po co mi to?” ;-) Najlepszą odpowiedzią na nie będzie wyjaśnienie różnicy pomiędzy zwykłymi oraz wirtualnymi metodami.

Posłuży nam do tego następujący kod, tworzący obiekt jednej z klasy pochodnych i wywołujący jego metodę `Oddychaj()`:

```
CAnimal* pZwierzak = new CMammal;
pZwierzak->Oddychaj();
delete pZwierzak;
```

Zauważmy, że wskaźnik `pZwierzak`, poprzez który odwołujemy się do naszego obiektu, jest zasadniczo wskaźnikiem na klasę `CAnimal`. Stwarzany przez nas (poprzez instrukcję `new`) obiekt należy natomiast do klasy `CMammal`. Wszystko jest jednak w porządku. Klasa `CMammal` dziedziczy od klasy `CAnimal`, zatem każdy obiekt należący do tej pierwszej jednocześnie jest także obiektem tej drugiej. Wyjaśniliśmy to sobie całkiem niedawno, prezentując dziedziczenie.
Zajmijmy się raczej drugą linijką powyższego kodu, zawierającą wywołanie interesującej nas metody `Oddychaj()`. Różnica między zwykłymi a wirtualnymi funkcjami składowymi będzie miała okazję uwidocznić się właśnie tutaj. Wszystko bowiem zależy od tego, jaką metodą jest rzeczona funkcja `Oddychaj()`, zaś rezultatem rozważanej instrukcji może być zarówno wywołanie `CAnimal::Oddychaj()`, jak i `CMammal::Oddychaj()`! Dowiedzmy się więc, kiedy zajdzie każda z tych sytuacji.

Łatwiejszym przypadkiem jest chyba „niewirtualność" rozpatrywanej metody. Kiedy jest ona zwyczajną funkcją składową, wtedy kompilator nie traktuje jej w żaden specjalny sposób. Co to jednak w praktyce oznacza?…
To dosyć proste. W takich bowiem wypadkach decyzja, która metoda jest rzeczywiście wywoływana, zostaje podjęta już na etapie kompilacji programu. Nazywamy ją wtedy **wczesnym wiązaniem** (ang. *early binding*) funkcji. Do jej podjęcia są zatem wykorzystane jedynie te informacje, które są znane w **momencie kompilacji** programu; u nas jest to typ wskaźnika `pZwierzak`, czyli `CAnimal`. Nie jest przecież możliwe ustalenie, na jaki obiekt będzie on faktycznie wskazywał - owszem, może on należeć do klasy `CAnimal`, jednak równie dobrze do jej pochodnej, na przykład `CMammal`. Wiedza ta nie jest jednak dostępna podczas kompilacji[84], dlatego też tutaj zostaje asekuracyjnie wykorzystany jedynie znany typ `CAnimal`. Faktycznie wywoływaną metodą będzie więc `CAnimal::Oddychaj()`!

Huh, to raczej nie jest to, o co nam chodziło. Skoro już tworzymy obiekt klasy `CMammal`, to w zasadzie logiczne jest, że zależy nam na wywołaniu funkcji pochodzącej z tej właśnie klasy, a nie z jej bazy! Spotyka nas jednak przykra niespodzienka…
Czy uchroni od niej zastosowanie metod wirtualnych? Domyślasz się zapewne, iż tak właśnie będzie, i na dodatek masz tutaj absolutną rację :) Kiedy użyjemy magicznego słówka `virtual`, kompilator wstrzyma się z decyzją co do faktycznie przywoływanej metody. Jej podjęcie nastąpi dopiero w stosowanej chwili **podczas działania** gotowej aplikacji; nazywamy to **późnym wiązaniem** (ang. *late binding*) funkcji. W tym momencie będzie oczywiście wiadome, jaki obiekt naprawdę kryje się za naszym wskaźnikiem `pZwierzak` i to jego wersja metody zostanie wywołana. Uzyskamy zatem skutek, o jaki nam chodziło, czyli wywołanie funkcji `CMammal::Oddychaj()`.

Prezentowany tu problem wyraźnie podpada już pod idee polimorfizmu, które wyczerpująco poznamy niebawem.

Wirtualny destruktor

Atrybut `virtual` możemy przyłączyć do każdej zwyczajnej metody, a nawet takiej niezupełnie zwyczajnej :) Czasami zresztą zastosowanie go jest niemal powinnością…

[84] Tak naprawdę kompilator może w ogóle nie wiedzieć, że `CAnimal` posiada jakieś klasy pochodne!

Jeżeli chodzi o konstruktory, to stosowanie tego modyfikatora w stosunku do nich nie ma zbyt wielkiego sensu. Są one przecież domyślnie „jakby wirtualne”: wywołanie konstruktora z klasy pochodnej powoduje przecież uruchomienie także konstruktorów z klas bazowych. Ich przedefiniowanie nie jest przy tym niczym nadzwyczajnym, tak więc użycie słowa `virtual` w tym przypadku mija się z celem.

Zupełnie inaczej sprawa ma się z destruktorami. Tutaj użycie omawianego modyfikatora jest nie tylko możliwe, ale też prawie zawsze **konieczne** i **zalecane**. Nieobecność wirtualnego destruktora w klasie bazowej może bowiem prowadzić do tzw. wycieków pamięci, czyli bezpowrotnej utraty zaalokowanej pamięci operacyjnej.
Dlaczego tak się dzieje? Do wyjaśnienia posłużymy się po raz kolejny naszymi wysłużonymi klasami zwierząt :D Przypuśćmy, że czujemy potrzebę, aby dokładniej odpowiadały one rzeczywistości; by nie były tylko zbiorami danych, ale też zawierały obiektowe odpowiedniki narządów wewnętrznych, na przykład serca czy płuc. Poczynimy więc najpierw pewne zmiany w bazowej klasie `CAnimal`:

```
// klasa serca
class CHeart { /* ... */ };

// bazowa klasa zwierząt
class CAnimal
{
   // (pomijamy nieistotne, pozostałe składowe)

   protected:
         CHeart* m_pSerce;
   public:
         // konstruktor i destruktor
         CAnimal()   { m_pSerce = new CHeart;  }
         ~CAnimal()  { delete m_pSerce;        }
};
```

Serce jest oczywiście organem, który posiada każde zwierzę, zatem obecność wskaźnika na obiekt klasy `CHeart` jest tu uzasadniona. Odwołuje się on do obiektu tworzonego w konstruktorze, a niszczonego w destruktorze klasy `CAnimal`.
Naturalnie, nie samym sercem zwierzę żyje :) Ssaki na przykład potrzebują jeszcze płuc:

```
// klasa płuc
class CLungs { /* ... */ };

// klasa ssaków
class CMammal : public CAnimal
{
   protected:
         CLungs* m_pPluca;
   public:
         // konstruktor i destruktor
         CMammal()   { m_pPluca = new CLungs;  }
         ~CMammal()  { delete m_pPluca;        }
};
```

Podobnie jak wcześniej, obiekt specjalnej klasy jest tworzony w konstruktorze i zwalniany w destruktorze `CMammal`. W ten sposób nasze ssaki są zaopatrzone zarówno w serce (otrzymane od `CAnimal`), jak i niezbędne płuca, tak więc pożyją sobie jeszcze trochę i będą mogły nadal służyć nam jako przykład ;)

OK, gdzie zatem tkwi problem?… Powróćmy teraz do trzech linijek kodu, za pomocą których rozstrzygnęliśmy pojedynek między wirtualnymi a niewirtualnymi metodami:

```
CAnimal* pZwierzak = new CMammal;
pZwierzak->Oddychaj();
delete pZwierzak;
```

Przypomnijmy, że `pZwierzak` jest tu zasadniczo zmienną typu „wskaźnik na obiekt klasy `CAnimal`", ale tak naprawdę wskazuje na obiekt należący do pochodnej `CMammal`. Ów obiekt musi oczywiście zostać usunięty, za co powinna odpowiadać ostatnia linijka…

No właśnie - powinna. Szkoda tylko, że tego nie robi. To zresztą nie jest jej wina, przyczyną jest właśnie brak wirtualnego destruktora.
Jak bowiem wiemy, zniszczenie obiektu oznacza w pierwszej kolejności wywołanie tej kluczowej metody. Podlega ono identycznym regułom, jakie stosują się do wszystkich innych metod, a więc także efektom związanym z wirtualnością oraz wczesnym i późnym wiązaniem. Jeżeli więc nasz destruktor nie będzie oznaczony jako `virtual`, to kompilator potraktuje go jako zwyczajną metodę i zastosuje wobec niej technikę wczesnego wiązania. Zasugeruje się po prostu typem zmiennej `pZwierzak` (którym jest `CAnimal*`, a więc wskaźnik na obiekt klasy `CAnimal`) i wywoła wyłącznie destruktor klasy bazowej `CAnimal`! Destruktor ten wprawdzie usunie serce naszego ssaka, ale nie zrobi tego z płucami, bo i nie ma przecież o nich zielonego pojęcia.
Nie dość zatem, że tracimy przez to pamięć przeznaczoną na tenże narząd, to jeszcze pozwalamy, by wokół fruwały nam organy pozbawione właścicieli ;D

To oczywiście tylko obrazowy dowcip, jednak konsekwencje niepełnego zniszczenia obiektów mogą być dużo poważniejsze, szczególnie jeśli ich składniki odwoływały się do siebie nawzajem. Weźmy choćby wspomniane płuca - powinny one przecież dostarczać tlen do serca, a jeżeli samo serce już nie istnieje, no to zaczynają się nieliche problemy…

Rozwiązanie problemu jest rzecz jasna nadzwyczaj proste - wystarczy uczynić destruktor klasy bazowej `CAnimal` metodą wirtualną:

```
class CAnimal
{
   // (oszczędność jest cnotą, więc znowu pomijamy resztę składowych :D)

   public:
         virtual ~CAnimal()      { delete m_pSerce; }
};
```

Wtedy też operator `delete` będzie usuwał obiekt, na który faktycznie wskazuje podany mu wskaźnik. My zaś uchronimy się od perfidnych błędów.

Pamiętaj zatem, aby **zawsze** umieszczać **wirtualny destruktor** w **klasie bazowej**.

Zaczynamy od zera… dosłownie

Deklarując metody opatrzone modyfikatorem `virtual`, tworzymy grunt pod ich przyszłą, ponowną implementację w klasach dziedziczących. Można też powiedzieć, iż w pewnym sensie zmieniamy charakter zawierającej je klasy: jej rolą nie jest już przede wszystkim tworzenie obiektów, gdyż równie ważne staje się służenie jako baza dla klas pochodnych.

Niekiedy słuszne jest pójście jeszcze dalej, to znaczy całkowite pozbawienie klasy możliwości tworzenia z niej obiektów. Ma to nierzadko rozsądne uzasadnienie i takimi właśnie przypadkami zajmiemy się w tym paragrafie.

Czysto wirtualne metody

Wirtualna funkcja składowa umieszczona w klasie bazowej jest przygotowana na to, aby ustąpić miejsca swej bardziej wyspecjalizowanej wersji, zdefiniowanej w klasie pochodnej. Nie zmienia to jednak faktu, iż musiałaby ona jakoś implementować czynność, której przebiegu często nie sposób ustalić na tym etapie.

Posiadamy dobry przykład, ilustrujący taką właśnie sytuację. Chodzi mianowicie o metodę `CAnimal::Oddychaj()`. Wewnątrz klasy bazowej, z której mają dopiero dziedziczyć konkretne grupy zwierząt, niemożliwe jest przecież ustalenie uniwersalnego sposobu oddychania. Sensowna implementacja tej metody jest więc w zasadzie niemożliwa.
Sprawia to, iż jest ona wyświenitym kandydatem na **czysto wirtualną funkcję składową**.

Metody nazywane **czysto wirtualnymi** (ang. *pure virtual*) nie posiadają **żadnej implementacji** i są przeznaczone do **przedefiniowania** w klasach pochodnych.

Deklaracja takiej metody ma dość osobliwą postać. Oczywiście z racji nie posiadania żadnego kodu zbędne stają się nawiasy klamrowe wyznaczające jej blok, zatem całość przypomina zwykły prototyp funkcji. Samo oznaczenie, czyniące daną metodę czysto wirtualną, jest jednak raczej niecodzienne:

```
class CAnimal
{
   // (definicja klasy jest skromna z przyczyn oszczędnościowych :))

   public:
         virtual void Oddychaj() = 0;
};
```

Jest nim występująca na końcu fraza `= 0;`. Kojarzy się ona trochę z domyślną wartością funkcji, ale interpretacja taka upada w obliczu niezwracania przez metodę `Oddychaj()` żadnego rezultatu. Faktycznie funkcją czysto wirtualną możemy w ten sposób uczynić **każdą** wirtualną metodę, niezależnie od tego, czy zwraca jakąś wartość i jakiego jest ona typu. Sekwencja `= 0;` jest więc po prostu takim dziwnym oznaczeniem, stosowanym dla tego rodzaju metod. Trzeba się z nim zwyczajnie pogodzić :)

Twórcy C++ wyraźnie nie chcieli wprowadzać tutaj dodatkowego słowa kluczowego, ale w tym przypadku trudno się z nimi zgodzić. Osobiście uważam, że deklaracja w formie na przykład `pure virtual void Oddychaj();` byłaby znacznie bardziej przejrzysta.

Po dokonaniu powyższej operacji metoda `CAnimal::Oddychaj()` staje się zatem czysto wirtualną funkcją składową. W tej postaci określa już tylko samą czynność, bez podawania żadnego algorytmu jej wykonania. Zostanie on ustalony dopiero w klasach dziedziczących od `CAnimal`.

Można aczkolwiek podać implementację metody czysto wirtualnej, jednak będzie ona mogła być wykorzystywana tylko w kodzie metod klas pochodnych, które ją przedefiniowują, w formie *klasa_bazowa*`::`*nazwa_metody*`(`[*parametry*]`)`.

Abstrakcyjne klasy bazowe

Nie widać tego na pierwszy, drugi, ani nawet na dziesiąty rzut oka, ale zadeklarowanie jakiejś metody jako czysto wirtualnej powoduje jeszcze jeden, dodatkowy efekt. Otóż klasa, w której taką funkcję stworzymy, staje się **klasą abstrakcyjną**.

Klasa abstrakcyjna zawiera przynajmniej jedną czysto wirtualną metodę i z jej powodu nie jest przeznaczona do instancjowania (tworzenia z niej obiektów), a **jedynie do wyprowadzania** zeń **klas pochodnych**.

Ze względu na wyżej wymienioną definicję czysto wirtualne funkcje składowe określa się niekiedy mianem metod abstrakcyjnych. Nazwa ta jest szczególnie popularna wśród programistów języka Object Pascal.

Takie klasy budują zawsze najwyższe piętra w hierarchiach i są podstawami dla bardziej wyspecjalizowanych typów. W naszym przypadku mamy tylko jedną taką klasę, z której dziedziczą wszystkie inne. Nazywa się `CAnimal`, jednak dobry zwyczaj programistyczny nakazuje, aby klasy abstrakcyjne miały nazwy zaczynające się od litery `I`. Różnią się one bowiem znacznie od pozostałych klas. Zatem baza w naszej hierarchii będzie od tej pory zwać się `IAnimal`.

C++ bardzo dosłownie traktuje regułę, iż klasy abstrakcyjne nie są przeznaczone do instancjowania. Próba utworzenia z nich obiektu zakończy się bowiem błędem; kompilator nie pozwoli na obecność czysto wirtualnej metody w klasie tworzonego obiektu.
Możliwe jest natomiast zadeklarowanie wskaźnika na obiekt takiej klasy i przypisanie mu obiektu klasy potomnej, tak więc poniższy kod będzie jak najbardziej poprawny:

```
IAnimal* pZwierze = new CBird;
pZwierze->Oddychaj();
delete pZwierze;
```

Wywołanie metody `Oddychaj()` jest tu także dozwolone. Wprawdzie w bazowej klasie `IAnimal` jest ona czysto wirtualna, jednak w `CBird`, do obiektu której odwołuje się nasz wskaźnik, posiada ona odpowiednią implementację.

Wydawałoby się, że C++ reaguje nieco zbyt alergicznie na próbę utworzenia obiektu klasy abstrakcyjnej - w końcu sama kreacja nie jest niczym niepoprawnym. W ten sposób jednak mamy pewność, że podczas działania programu wszystko będzie działać poprawnie i że omyłkowo nie zostanie wywołana metoda z nieokreśloną implementacją.

Polimorfizm

Gdyby programowanie obiektowe porównać do wysokiego budynku, to u jego fundamentów leżałyby pojęcia „klasy” i „obiektu”, środkowe piętra budowałoby „dziedziczenie” oraz „metody wirtualne”, zaś u samego szczytu sytuowałby się „polimorfizm”. Jest to bowiem największe osiągnięcie tej metody programowania.

Z terminem tym spotykaliśmy się przelotnie już parę razy, ale teraz wreszcie wyjaśnimy sobie wszystko od początku do końca. Zacznijmy choćby od samego słowa: ‘polimorfizm’ pochodzi od greckiego wyrazu *polýmorphos*, oznaczającego ‘wielokształtny’ lub ‘wielopostaciowy’. W programowaniu będzie się więc odnosić do takich tworów, które można interpretować na różne sposoby - a więc należących jednocześnie do kilku różnych typów (klas).

Polimorfizm w programowaniu obiektowym oznacza wykorzystanie tego samego kodu do operowania na obiektach przynależnych różnym klasom, dziedziczącym od siebie.

Zjawisko to jest zatem ściśle związane z klasami i dziedziczeniem, aczkolwiek w C++ nie dotyczy ono każdej klasy, a jedynie określonych **typów polimorficznych**.

Typ polimorficzny to w C++ klasa zawierająca przynajmniej jedną **metodę wirtualną**.

W praktyce większość klas, do których chcielibyśmy stosować techniki polimorfizmu, spełnia ten warunek. W szczególności tą wymaganą metodą wirtualną może być chociażby destruktor.

Wszystko to brzmi bardzo ładnie, ale trudno nie zadać sobie pytania o praktyczne korzyści związane z wykorzystaniem polimorfizmu. Dlatego też moim celem będzie teraz drobiazgowa odpowiedź na to pytanie - innymi słowy, wreszcie doczekałeś się konkretów ;D

Ogólny kod do szczególnych zastosowań

Zjawisko polimorfizmu pozwala na znaczne uproszczenie większości algorytmów, w których dużą rolę odgrywa zarządzanie wieloma różnymi obiektami. Nie chodzi tu wcale o jakieś skomplikowane operacje sortowania, wyszukiwania, kompresji itp., tylko o często spotykane operacje wykonywania tej samej czynności dla wielu obiektów **różnych rodzajów**.

Opis ten jest w założeniu dość ogólny, bowiem sposób, w jaki używa się obiektowych technik polimorfizmu jest ściśle związany z konkretnymi programami. Postaram się jednak przytoczyć w miarę klarowne przykłady takich rozwiązań, abyś miał chociaż ogólne pojęcie o tej metodzie programowania i mógł ją stosować we własnych aplikacjach.

Sprowadzanie do bazy

Prosty przypadek wykorzystania polimorfizmu opiera się na elementarnej i rozsądnej zasadzie, którą nie raz już sprawdziliśmy w praktyce. Mianowicie:

Wskaźnik na obiekt klasy bazowej może wskazywać także na obiekt którejkolwiek z jego klas pochodnych.

Bezpośrednie przełożenie tej reguły na konkretne zastosowanie programistyczne jest dość proste. Przypuśćmy więc, że mamy taką oto hierarchię klas:

```
#include <string>
#include <ctime>

// klasa dowolnego dokumentu
class CDocument
{
   protected:
         // podstawowe dane dokumentu
         std::string m_strAutor;       // autor dokumentu
         std::string m_strTytul;       // tytuł dokumentu
         tm          m_Data;           // data stworzenia
   public:
         // konstruktory
         CDocument()
               { m_strAutor = m_strTytul = "???";
                 time_t Czas = time(NULL); m_Data = *localtime(&Czas); }
         CDocument(std::string strTytul)
               { CDocument(); m_strTytul = strTytul; }
         CDocument(std::string strAutor, std::string strTytul)
               { CDocument();
                 m_strAutor = strAutor;
                 m_strTytul = strTytul; }
```

```
        //-------------------------------------------------------------

        // metody dostępowe do pól
        std::string Autor() const   { return m_strAutor; }
        std::string Tytul() const   { return m_strTytul; }
        tm          Data()  const   { return m_Data;     }
};

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

// dokument internetowy
class COnlineDocument : public CDocument
{
   protected:
        std::string m_strURL;   // adres internetowy dokumentu
   public:
        // konstruktory
        COnlineDocument(std::string strAutor, std::string strTytul)
              { m_strAutor = strAutor; m_strTytul = strTytul; }
        COnlineDocument  (std::string strAutor,
                          std::string strTytul,
                          std::string strURL)
              { m_strAutor = strAutor;
                m_strTytul = strTytul;
                m_strURL   = strURL;   }

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

        // metody dostępowe do pól
        std::string URL()   const   { return m_strURL; }
};

// książka
class CBook : public CDocument
{
   protected:
        std::string m_strISBN;  // numer ISBN książki
   public:
        // konstruktory
        CBook(std::string strAutor, std::string strTytul)
              { m_strAutor = strAutor; m_strTytul = strTytul; }
        CBook (std::string strAutor,
               std::string strTytul,
               std::string strISBN)
              { m_strAutor = strAutor;
                m_strTytul = strTytul;
                m_strISBN  = strISBN; }

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

        // metody dostępowe do pól
        std::string ISBN()  const   { return m_strISBN; }
};
```

Z klasy `CDocument`, reprezentującej dowolny dokument, dziedziczą dwie następne: `COnlineDocument`, odpowiadająca tekstom dostępnym przez Internet, oraz `CBook`, opisująca książki.
Napiszmy również odpowiednią funkcję, wyświetlającą podstawowe informacje o podanym dokumencie:

```
#include <iostream>
```

```
void PokazDaneDokumentu(CDocument* pDokument)
{
   // wyświetlenie autora
   std::cout << "AUTOR: ";
   std::cout << pDokument->Autor() << std::endl;

   // pokazanie tytułu dokumentu
   // (sekwencja \" wstawia cudzysłów do napisu)
   std::cout << "TYTUL: ";
   std::cout << "\"" << pDokument->Tytul() << "\"" << std::endl;

   // data utworzenia dokumentu
   // (pDokument->Data() zwraca strukturę typu tm, do której pól
   //  można dostać się tak samo, jak do wszystkich innych - za
   //  pomocą operatora wyłuskania . (kropki))
   std::cout << "DATA : ";
   std::cout << pDokument->Data().tm_mday << "."
             << (pDokument->Data().tm_mon + 1) << "."
             << (pDokument->Data().tm_year + 1900) << std::endl;
}
```

Bierze ona jeden parametr, będący zasadniczo wskaźnikiem na obiekt typu `CDocument`. W jego charakterze może jednak występować także wskazanie na któryś z obiektów potomnych, zatem poniższy kod będzie absolutnie prawidłowy:

```
COnlineDocument* pTutorial = new COnlineDocument("Xion",       // autor
                             "Od zera do gier kodera",         // tytuł
                             "http://avocado.risp.pl");        // URL
PokazDaneDokumentu (pTutorial);
delete pTutorial;
```

W pierwszej linijce możnaby równie dobrze użyć typu wskazującego na obiekt `CDocument`, gdyż wskaźnik `pTutorial` i tak zostanie potraktowany w ten sposób przy przekazywaniu go do funkcji `PokazDaneDokumentu()`.

Efektem jego działania powyższego listingu będzie na przykład taki oto widok:

Screen 37. Informacje o dokumencie uzyskane z użyciem prostego polimorfizmu

Brak tu informacji o adresie internetowym dokumentu, ponieważ należy on do składowych specyficznych dla klasy `COnlineDocument`. Funkcja `PokazDaneDokumentu()` została natomiast stworzona do pracy z obiektami `CDocument`, zatem wykorzystuje jedynie informacje zawarte w klasie bazowej. Nie przeszkadza to jednak w przekazaniu jej obiektu klasy pochodnej - w takim przypadku dodatkowe dane zostaną po prostu zignorowane.

To raczej mało satysfakcjonujące rozwiązanie, ale lepsze skutki wymagają już użycia metod wirtualnych. Uczynimy to w kolejnym przykładzie.

Naturalnie, podobny rezultat otrzymalibyśmy podając naszej funkcji obiekt klasy `CBook` czy też **jakiejkolwiek innej** dziedziczącej od `CDocument`. Kod procedury jest więc uniwersalny i może być stosowany do wielu różnych rodzajów obiektów.

Eureka! Na tym przecież polega polimorfizm :)

Możliwe że zauważyłeś, iż żadna z tych przykładowych klas nie jest tutaj typem polimorficznym, a jednak podany wyżej kod działa bez zarzutu. Powodem tego jest jego względna prostota. Dokładniej mówiąc, nie jest konieczne sprawdzanie poprawności typów podczas działania programu, bo wystarczająca jest zwykła kontrola, dokonywana zwyczajowo podczas kompilacji kodu.

Klasy wiedzą same, co należy robić

Z poprzednim przykładem związany jest pewien mankament, nietrudno zresztą zauważalny. Niezależnie od tego, jakie dodatkowe dane o dokumencie zadeklarujemy w klasach pochodnych, nasza funkcja wyświetli tylko i wyłącznie te przewidziane w klasie `CDocument`. Nie uzyskamy więc nic ponad autora, tytuł oraz datę stworzenia dokumentu.

Trzeba jednak przyznać, że sami niejako jesteśmy sobie winni. Wyodrębniając czynność prezentacji obiektu poza sam obiekt postąpiliśmy niezgodnie z ideą OOPu, która nakazuje łączyć dane i operujący na nich kod.

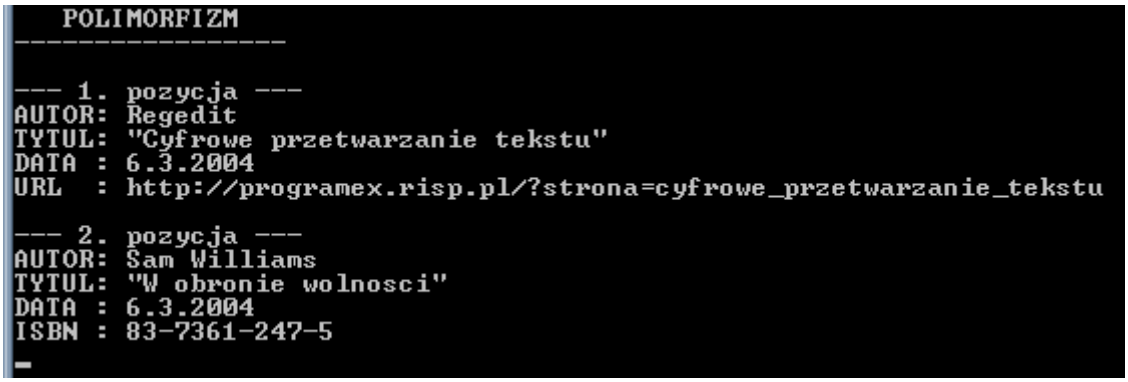
Zatem przykład z poprzedniego paragrafu to zdecydowanie zły przykład :D

O wiele lepszym rozwiązaniem jest dodanie do klasy `CDocument` odpowiedniej metody, odpowiedzialnej za czynność wypisywania. A już całkowitym ideałem będzie uczynienie jej funkcją wirtualną - wtedy klasy dziedziczące od `CDocument` będą mogły ustalić własny sposób prezentacji swoich danych.
Wszystkie te doskonałe pomysły praktycznie realizuje poniższy program przykładowy:

```
// Polymorphism - wykorzystanie techniki polimorfizmu


// *** documents.h ***

class CDocument
{
   // (większość składowych wycięto z powodu zbyt dużej objętości)

   public:
         virtual void PokazDane();
};

// (reszty klas nieuwzględniono z powodu dziury budżetowej ;D)
// (zaś ich implementacje są w pliku documents.cpp)


// *** main.cpp ***

#include <iostream>
#include <conio.h>
#include "documents.h"

void main()
{
   // wskaźnik na obiekty dokumentów
   CDocument* pDokument;

   // pierwszy dokument - internetowy
   std::cout << std::endl << "--- 1. pozycja ---" << std::endl;
   pDokument = new COnlineDocument("Regedit",
                                   "Cyfrowe przetwarzanie tekstu",
                                   "http://programex.risp.pl/?"
```

```
                                        "strona=cyfrowe_przetwarzanie_tekstu"
                                        );
    pDokument->PokazDane();
    delete pDokument;

    // drugi dokument - książka
    std::cout << std::endl << "--- 2. pozycja ---" << std::endl;
    pDokument = new CBook("Sam Williams",
                          "W obronie wolnosci",
                          "83-7361-247-5");
    pDokument->PokazDane();
    delete pDokument;

    getch();
}
```

Wynikiem jego działania będzie poniższe zestawienie:



Screen 38. Aplikacja prezentująca polimorfizm z wykorzystaniem metod wirtualnych

Zauważmy, że za wyświetlenie obu widniejących na nim pozycji odpowiada wywołanie pozornie tej samej funkcji:

```
pDokument->PokazDane();
```

Polimorficzny mechanizm metod wirtualnych sprawia jednak, że zawsze wywoływana jest **odpowiednia wersja** procedury `PokazDane()` - odpowiednia dla kolejnych obiektów, na które wskazuje `pDokument`.

Tutaj mamy wprawdzie tylko dwa takie obiekty, ale nietrudno wyobrazić sobie analogiczne działanie dla większej ich liczby, np.:

```
CDocument* apDokumenty[100];

for (unsigned i = 0; i < 100; ++i)
   apDokumenty[i]->PokazDane();
```

Poszczególne elementy tablicy `apDokumenty` mogą wskazywać na obiekty **dowolnych klas**, dziedziczących od `CDocument`, a i tak kod wyświetlający ich dane będzie ograniczał się do wywołania zaledwie **jednej** metody! I to właśnie jest piękne :D

Możliwe zastosowania takiej techniki można mnożyć w nieskończoność, zaś w grach jest po prostu nieoceniona. Pomyślmy tylko, że za pomocą podobnej tablicy i prostej pętli możemy wykonać dowolną czynność na zestawie przeróżnych obiektów. Rysowanie, wyświetlanie, kontrola animacji - wszystko to możemy wykonać poprzez jedną instrukcję! Niezależnie od tego, jak bardzo byłaby rozbudowana hierarchia naszych klas (np.

jednostek w grze strategicznej, wrogów w grze RPG, i tak dalej), zastosowanie polimorfizmu z metodami wirtualnymi upraszcza kod większości operacji do podobnie trywialnych konstrukcji jak powyższa.
Od tej pory do nas należy więc tylko zdefiniowanie odpowiedniego modelu klas i ich metod, gdyż zarządzanie poszczególnymi obiektami staje się, jak widać, banalne. Co ważniejsze, zastosowanie technik obiektowych nie tylko upraszcza kod, ale też pozwala na znacznie większą elastyczność.

Pamiętaj, że praktyka czyni mistrza! Poznanie teoretycznych aspektów programowania obiektowego jest wprawdzie niezbędne, ale najwięcej wartościowych umiejętności zdobędziesz podczas samodzielnego projektowania i kodowania programów. Wtedy szybko przekonasz się, że stosowanie technik polimorfizmu jest prawie że intuicyjne - nawet jeśli teraz nie jesteś zbytnio tego pewien.

Typy pod kontrolą

Uniwersalny kod dla wszystkich klas w hierarchii jest bardzo wygodnym rozwiązaniem. Okazjonalnie jednak zdarza się, że trzeba w nim uwzględnić także bardziej szczegółowe przypadki, co oznacza konieczność sprawdzania faktycznego typu obiektów, na które wskazują nasze wskaźniki.
Na szczęście C++ oferuje proste mechanizmy, umożliwiające realizację tego zadania.

Operator `dynamic_cast`

Konwersja wskaźnika do klasy pochodnej na wskaźnik do klasy bazowej jest czynnością dość naturalną, więc przebiega całkowicie automatycznie. Niepotrzebne jest nawet zastosowanie jakiejś formy rzutowania. Nie powinno to wcale dziwić - w końcu na tym polega sama idea dziedziczenia, że obiekt klasy potomnej jest także obiektem przynależnym klasie bazowej.
Inaczej jest z konwersją w odwrotną stronę - ta nie zawsze musi się przecież powieść. C++ powinien więc udostępniać jakiś sposób na sprawdzenie, czy taka zamiana jest możliwa, no i na samo jej przeprowadzanie. Do tych celów służy **operator rzutowania** `dynamic_cast`.

Jest to drugi z operatorów rzutowania, jakie mamy okazję poznać. Został on wprowadzony do języka C++ po to, by umożliwić kompleksową obsługę typów polimorficznych w zakresie konwersji „w dół” hierarchii klas. Jego przeznaczenie jest zatem następujące:

Operator `dynamic_cast` służy do rzutowania wskaźnika do obiektu klasy bazowej na wskaźnik do obiektu klasy pochodnej.

Powiedzieliśmy sobie również, że taka konwersja niekoniecznie musi być możliwa. Rolą omawianego operatora jest więc także sprawdzanie, czy rzeczywiście mamy do czynienia z wskaźnikiem do obiektu potomnego, przechowywanym przez zmienną będącą wskaźnikiem do typu bazowego.

Uff, wszystko to wydaje się bardzo zakręcone, zatem najlepiej będzie, jeżeli przyjrzymy się odpowiednim przykładom. Po raz kolejny posłużymy się przy tym naszą ulubioną systematyką klas zwierząt i napiszemy taką oto funkcję:

```
#include <stdlib.h>        // żeby użyć rand() i srand()
#include <ctime>           // żeby użyć time()

IAnimal* StworzLosoweZwierze()
{
   // zainicjowanie generatora liczb losowych
```

```
    srand (static_cast<unsigned>(time(NULL)));

    // wylosowanie liczby i stworzenie obiektu zwierza
    switch (rand() % 4)
    {
          case 0:     return new CFish;
          case 1:     return new CMammal;
          case 2:     return new CBird;
          case 3:     return new CHomeDog;
          default:    return NULL;
    }
}
```

Losuje ona liczbę i na jej podstawie tworzy obiekt jednej z czterech, zdefiniowanych jakiś czas temu, klas zwierząt. Następnie zwraca wskaźnik do niego jako wynik swego działania. Rezultat ten jest rzecz jasna typu `IAnimal*`, aby mógł „pomieścić" odwołania do jakiegokolwiek zwierzęcia, dziedziczącego z klasy bazowej `IAnimal`.

Powyższa funkcja jest bardzo prostym wariantem tzw. fabryki obiektów (ang. *object factory*). Takie fabryki to najczęściej osobne obiekty, które tworzą zależne do siebie byty np. na podstawie stałych wyliczeniowych, przekazywanych swoim metodom. Metody takie mogą więc zwrócić wiele różnych rodzajów obiektów, dlatego deklaruje się je z użyciem wskaźników na klasy bazowe - u nas jest to `IAnimal*`.

Wywołanie tej funkcji zwraca nam więc **dowolne** zwierzę i zdawałoby się, że nijak nie potrafimy sprawdzić, do jakiej klasy ono faktycznie należy. Z pomocą przychodzi nam jednak operator `dynamic_cast`, dzięki któremu możmue **spróbować** rzutowania otrzymanego wskaźnika na przykład do typu `CMammal*`:

```
IAnimal* pZwierze = StworzLosoweZwierze();
CMammal* pSsak = dynamic_cast<CMammal*>(pZwierze);
```

Taka próba powiedzie się jednak tylko w średnio połowie przypadków (dlaczego?[85]). Co zatem będzie, jeżeli `pZwierze` odnosi się do innego rodzaju zwierząt?…
Otóż w takim przypadku otrzymamy prostą informację o błędzie, mianowicie:

`dynamic_cast` zwróci wskaźnik pusty (o wartości `NULL`), jeżeli niemożliwe będzie dokonanie podanego rzutowania.

Aby ją wychwycić potrzebujemy oczywiście dodatkowego warunku, porównującego zmienną `pSsak` z tą specjalną wartością `NULL` (będącą zresztą *de facto* zerem):

```
if (pSsak != NULL)   // sprawdzenie, czy rzutowanie powiodło się
{
    // OK - rzeczywiście mamy do czynienia z obiektem klasy CMammal.
    // pSsak może być tu użyty tak samo, jak każdy inny wskaźnik
    // na obiekt klasy CMammal, na przykład:
    pSsak->Biegnij();
}
```

Warunek `if (pSsak != NULL)` może być zastąpiony przez `if (pSsak)`. Wówczas kompilator dokona automatycznej zamiany wartości `pSsak` na logiczną, co da fałsz, jeżeli jest ona równa zeru (czyli `NULL`) oraz prawdę w każdej innej sytuacji.

[85] Obiekt klasy `CMammal` jest tworzony zarówno poprzez `new CMammal`, jak i `new CHomeDog`. Klasa `CHomeDog` dziedziczy przecież po klasie `CMammal`.

Możliwe jest nawet większe skondensowanie kodu. Wystarczy wstawić linijkę z rzutowaniem bezpośrednio do warunku `if`, tzn. zastosować instrukcję:
`if (CMammal* pSsak = dynamic_cast<CMammal*>(pZwierzak))`
Pojedynczy znak `=` jest tutaj umieszczony celowo, gdyż w ten sposób całe przypisanie reprezentuje wynik rzutowania, który zostaje potem niejawnie przyrównany do zera.

Kontrola otrzymanego wyniku rzutowania jest konieczna; jeżeli bowiem spróbowaliśmy zastosować operator wyłuskania `->` do pustego wskaźnika, spowodowalibyśmy błąd ochrony pamięci (*access violation*).

Należy więc zawsze sprawdzać, czy rzutowanie `dynamic_cast` powiodło się, poprzez porównanie otrzymanego wskaźnika z wartością `NULL`.

I to jest w zasadzie wszystko, co należy wiedzieć o operatorze `dynamic_cast` :)

Incydentalnie trafiają się sytuacje, w których zastosowanie omawianego operatora wymaga włączenia specjalnej opcji kompilatora, uaktywniającej informacje o typie podczas działania programu. Są to rzadkie przypadki i prawie zawsze dotyczą wielodziedziczenia, niemniej warto wiedzieć, że takie niespodzianki mogą się czasem przytrafić.
Sposób włączenia informacji o typie w czasie działania programu jest opisany w następnym paragrafie.

Bliższych szczegółow na temat rzutowania `dynamic_cast` można doszukać się w MSDN.

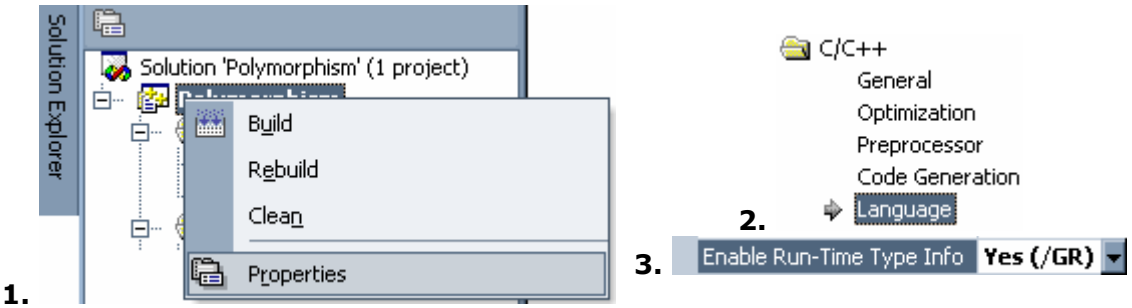
`typeid` i informacje o typie podczas działania programu

Oprócz `dynamic_cast` - operatora, który pozwala sprawdzić, czy dany wskaźnik do klasy bazowej wskazuje też na obiekt klasy pochodnej - C++ dysponuje nieco bardziej zaawansowanymi konstrukcjami, dostarczającymi wiadomości o typach obiektów i wyrażeń w programie. Są to tak zwane **informacje o typie czasu wykonania** (ang. *Run-Time Type Information*), oznaczane angielskim skrótowcem **RTTI**.

Znajomość opisywanej tu części RTTI, czyli operatora `typeid`, generalnie nie jest tak potrzebna, jak umiejętność posługiwania się operatorami rzutowania, ale daje kilka bardzo ciekawych możliwości, najczęściej nieosiągalnych inną drogą. Możesz aczkolwiek tymczasowo pominąć ten paragraf, by wrócić do niego później.

Skorzystanie z RTTI wymaga podjęcia dwóch wstępnych kroków:

- włączenia odpowiedniej opcji kompilatora:



Screen 39, 40 i 41. Trzy kroki do włączenia RTTI w Visual Studio .NET

W Visual Studio. NET należy w tym celu rozwinąć zakładkę *Solution Explorer*, kliknąć prawym przyciskiem myszy na nazwę swojego projektu i z menu podręcznego wybrać *Properties*. W pojawiającym się oknie dialogowym trzeba teraz przejść do strony *C/C++|Language* i przy opcji *Enable Run-Time Type Info* ustawić wariant *Yes (/GR)*.

- dołączenia do kodu standardowego nagłówka *typeinfo*, czyli dodania dyrektywy: `#include <typeinfo>`

W zamian za te wysiłki otrzymamy możliwość korzystania z operatora `typeid`, pobierającego informację o typie podanego mu wyrażenia. Składnia jego użycia jest następująca:

```
typeid(wyrażenie).informacja
```

Faktycznie instrukcja `typeid(wyrażenie)` zwraca strukturę, należącą do wbudowanego typu `std::type_info`. Struktura ta opisuje typ `wyrażenia` i zawiera takie oto składowe:

informacja	***opis***

`name()`	Jest to nazwa typu w czytelnej i przyjaznej dla człowieka formie. Możemy ją przechowywać i operować nią tak, jak każdym innym napisem. Przykład:
	`#include <typeinfo`>
	`#include <iostream>`
	`#include <ctime>`
	`int nX = 10; float fY = 3.14;`
	`time_t Czas = time(NULL); tm Data = *localtime(&Czas);`
	`std::cout << typeid(nX).name(); // wynik: "int"`
	`std::cout << typeid(fY).name(); // wynik: "float"`
	`std::cout << typeid(Data).name(); // wynik: "struct tm"`

`raw_name()`	Zwraca nazwę typu wewnętrznie używaną przez kompilator. Taka nazwa musi być unikalna, dlatego zawiera różne „dekoracyjne” znaki, jak ? czy @. Nie jest czytelna dla człowieka, ale może ewentualnie służyć w celach porównawczych.

Tabela 11. Informacje dostępne poprzez operator `typeid`

Oprócz pobierania nazwy typu w postaci ciągu znaków możemy używać operatorów `==` oraz `!=` do porównywania typów dwóch wyrażeń, na przykład:

```
unsigned uX;

if (typeid(uX) == typeid(unsigned))
   std::cout << "Świetnie, nasz kompilator działa ;D";
if (typeid(uX) != typeid(uX / 0.618))
   std::cout << "No proszę, tutaj też jest dobrze :)";
```

`typeid` mógłby więc służyć nam do sprawdzania klasy, do której należy polimorficzny obiekt wskazywany przez wskaźnik. Sprawdźmy zatem, jak by to mogło wyglądać:

```
IAnimal* pZwierze = new CBird;
std::cout << typeid(pZwierze).name();
```

Po wykonaniu tego kodu spotka nas raczej przykra niespodzienka - zamiast oczekiwanego rezultatu `"class CBird *"` otrzymamy `"class IAnimal *"`! Wygląda na to, że faktyczny typ obiektu, do którego odwołuje się `pZwierze`, nie został w ogóle wzięty pod uwagę.

Przypuszczenia te są słuszne. Otóż `typeid` jest „leniwym” operatorem i zawsze idzie po najmniejszej linii oporu. Typ wyrażenia `pZwierze` mógł zaś określić nie sięgając nawet do mechanizmów polimorficznych, ponieważ wyraźnie zadeklarowaliśmy go jako `IAnimal*`. Aby zmusić krnąbrny operator do większego wysiłku, musimy mu podać **sam obiekt**, a nie wskaźnik na niego, co czynimy w ten sposób:

```
std::cout << typeid(*pZwierze).name();
```

O występującym tu operatorze dereferencji - gwiazdce (`*`) powiemy sobie bliżej, gdy przejdziemy do dokładnego omawiania wskaźników jako takich. Na razie zapamiętaj, że przy jego pomocy „wyławiamy” obiekt poprzez wskaźnik do niego.
Naturalnie, teraz powyższy kod zwróci prawidłowy wynik `"class CBird"`.

Pełny opis operatora `typeid` znajduje się oczywiście w MSDN.

Alternatywne rozwiązania

RTTI jest często zbyt ciężką armatą, wytoczoną przeciw problemowi pobierania informacji o klasie obiektu podczas działania aplikacji. Przy niewielkim nakładzie pracy można samemu wykonać znacznie mniejszy, acz nierzadko wystarczający system.

Po co? Decydującym argumentem może być szybkość. Wbudowane mechanizmy RTTI, jak `dynamic_cast` i `typeid`, są dosyć wolne (szczególnie dotyczy to tego pierwszego). Własne, bardziej poręczne rozwiązanie może mieć spory wpływ na wydajność.

Do tego celu mogą posłużyć metody wirtualne oraz odpowiedni typ wyliczeniowy, posiadający listę wartości odpowiadających poszczególnym klasom. W przypadku naszych zwierząt mógłby on wyglądać na przykład tak:

```
enum ANIMAL { A_BASE,       // bazowa klasa IAnimal
              A_FISH,       // klasa CFish
              A_MAMMAL,     // klasa CMammal
              A_BIRD,       // klasa CBird
              A_HOMEDOG };  // klasa CHomeDog
```

Teraz wystarczy tylko zdefiniować proste metody wirtualne, które będą zwracały stałe właściwe swoim klasom:

```
// (pominąłem pozostałe składowe klas)

class IAnimal
{
   public:
         virtual ANIMAL Typ() const { return A_BASE; }
};

//----------------------bezpośrednie pochodne -------------------------

class CFish : public IAnimal
{
   public:
         ANIMAL Typ() const { return A_FISH: }
};

class CMammal : public IAnimal
{
   public:
         ANIMAL Typ() const { return A_MAMMAL; }
```

```
};

class CBird : public IAnimal
{
   public:
         ANIMAL Typ() const { return A_BIRD; }
};

// ---------------------- pośrednie pochodne ----------------------------

class CHomeDog : public CMammal
{
   public:
         ANIMAL Typ() const { return A_HOMEDOG; }
};
```

Po zastosowaniu tego rozwiązania możemy chociażby użyć instrukcji `switch`, by wykonać kod zależny od typu obiektu:

```
IAnimal* pZwierzak = StworzLosoweZwierze();

switch (pZwierzak->Typ())
{
   case A_FISH:    static_cast<CFish*>(pZwierzak)->Plyn();          break;
   case A_BIRD:    static_cast<CBird*>(pZwierzak)->Lec();           break;
   case A_MAMMAL:  static_cast<CMammal*>(pZwierzak)->Biegnij();     break;
   case A_HOMEDOG: static_cast<CHomeDog*>(pZwierzak)->Szczekaj();   break;
}
```

Podobne sprawdzenie, dokonywane przy użyciu `dynamic_cast` lub `typeid`, wymagałoby wielopiętrowej instrukcji `if`. Tutaj wystarczy bardziej naturalny `switch`, zaś do formalnego rzutowania możemy użyć prostego `static_cast`, które działa szybciej niż mechanizmy RTTI.

Trzeba jednak pamiętać, że aby bezpiecznie stosować `static_cast` do rzutowania w dół hierarchii klas, musimy mieć pewność, że taka operacja jest faktycznie wykonalna. Tutaj sprawdzamy rzeczywisty typ obiektu[86], zatem wszystko jest w porządku, lecz w innych przypadkach należy skorzystać z `dynamic_cast`.

Systemy identyfikacji i zarządzania typami, podobne do powyższego, są w praktyce używane bardzo często, szczególnie w wielkich projektach. Najbardziej zaawansowane warianty umożliwiają nawet tworzenie obiektów na podstawie nazwy klasy przechowywanej jako napis lub też dynamiczne odtworzenie hierarchii klas podczas działania programu. Trzeba jednak przyznać, iż jest to nierzadko sztuka dla samej sztuki, bez wielkiego praktycznego znaczenia.

„Równowaga przede wszystkim” - pamiętajmy tę sentencję :D

***

Gratulacje! Właśnie poznałeś wszystkie teoretyczne założenia programowania obiektowego i ich praktyczną realizację w C++. Wykorzystując zdobytą wiedzę, będziesz mógł efektywnie programować aplikacje z użyciem filozofii OOP.

[86] Sprawdzenie przy użyciu `typeid` także upoważniałoby nas do stosowania `static_cast` podczas rzutowania.

Słucham? Mówisz, że to wcale nie jest takie proste? Zgadza się, na początku myślenie w kategoriach obiektowych może rzeczywiście sprawiać ci trudności. Pomyślałem więc, że dobrze będzie poświęcić nieco czasu także na zagadnienia związane z samym projektowaniem aplikacji z użyciem poznanych technik. Zajmiemy się tym w nadchodzącym podrozdziale.

Projektowanie zorientowane obiektowo

A miało być tak pięknie… Programowanie obiektowe miało być przecież wyjątkowo naturalnym sposobem kodowania, a poprzednie paragrafy raczej nie bardzo o tym przekonywały, prawda? Jeżeli rzeczywiście odnosisz takie wrażenie, to być może zwyczajnie utonąłeś w powodzi szczegółów, dodajmy - niezbędnych szczegółów, koniecznych do stosowania OOPu w praktyce. Czas jednak wypłynąć na powierzchnię i ponownie spojrzeć na zagadnienie bardziej całościowo. Temu celowi będzie służyć niniejszy podrozdział.

Wiele podręczników opisujących programowanie obiektowe (czy nawet programowanie jako takie) wspomina skąpo, jeżeli w ogóle, o praktycznym stosowaniu prezentowanych mechanizmów, czyli po prostu o projektowaniu aplikacji z użyciem omawianych technik. Możnaby to wybaczyć tym publikacjom, których głównym celem jest „jedynie" kompletny opis danego języka. Jeżeli jednak mówimy o materiałach dla całkiem początkujących, będących w założeniu wprowadzeniem w świat programowania, wtedy zdecydowanie niewskazane jest pomijanie praktycznych stron projektowania i kodowania aplikacji. Na co bowiem przyda się znajomość budowy młotka, jeśli nie ułatwi to zadania, jakim jest wbicie gwoździa? :)

Staram się więc uniknąć tego błędu i przedstawiam programowanie obiektowe także od strony programisty-praktyka. Mam jednocześnie nadzieję, że w ten sposób przynajmniej częściowo uchronię cię przed wyważaniem otwartych drzwi w poszukiwaniu informacji w gruncie rzeczy oczywistych - które jednak wcale takie nie są, gdy się ich nie posiada. Naturalnie, nic nie zastąpi doświadczenia zdobytego samodzielnie podczas prawdziwego kodowania. Prezentowana tutaj wiedza teoretyczno-praktyczna może być jednak bardzo pomocnym punktem startowym, ułatwiającym koderskie życie przynajmniej na jego początku.
Cóż więc znajdziemy w aktualnym podrozdziale? Żałuję, ale nie będzie to przegląd kolejnych kroków, jakie należy czynić programując konkretną aplikację. Zamiast na mniej lub bardziej trywialnym programiku skoncentrujemy się raczej na ogólnym procesie budowania wewnętrznej, obiektowej struktury programu - czyli na tak zwanym **modelowaniu klas** i ich związków. Najpierw poznamy zatem trzy podstawowe rodzaje obiektów albo, jak kto woli, ról, w których one występują. Dalej zajmiemy się kwestią definiowania odpowiednich klas - ich interfejsu i implementacji, a wreszcie związkami pomiędzy nimi, dzięki którym programy stworzone według zasad OOPu mogą poprawnie funkcjonować.

Znajomość powyższego zestawu zagadnień powinna znacznie poprawić twoje szanse w starciu z problemami projektowymi, związanymi z programowaniem obiektowym. Być może ich rozwiązywanie nie będzie już wówczas wiedzą tajemną, ale normalną i, co ważniejsze, satysfakcjonującą częścią pracy kodera.
Nie przedłużając już więcej zacznijmy zatem właściwą treść tego podrozdziału.

Rodzaje obiektów

Każdy program zawiera w mniejszej lub większej części nowatorskie rozwiązania, stanowiące główne wyzwanie stojące przed jego twórcą. Niemniej jednak pewne cechy

prawie zawsze pozostają stałe - a do nich należy także podział obiektów składowych aplikacji na trzy fundamentalne grupy.

Podział ten jest bardzo ogólny i niezbyt sztywny, ale przez to stosuje się w zasadzie do każdego projektu. Będzie on zresztą punktem wyjścia dla nieco bardziej szczegółowych kwestii, opisanych później.
Pomówmy więc kolejno o każdym rodzaju z owej podstawowej trójki.

Singletony

Większość obiektów jest przeznaczonych do istnienia w wielu egzemplarzach, różniących się przechowywanymi danymi, lecz wykonujących te same działania poprzez metody. Istnieją jednakże wyjątki od tej reguły, a należą do nich właśnie singletony.

Singleton ('jedynak') to klasa, której **jedyna instancja** (obiekt) spełnia kluczową rolę w całym programie.

W danym momencie działania aplikacji istnieje więc co najwyżej **jeden egzemplarz** klasy, będącej singletonem.

Obiekty takie są dosłownie jedyne w swoim rodzaju i dlatego zwykle przechowują one najważniejsze dane programu oraz wykonają większość newralgicznych czynności. Najczęściej są też „rodzicami" i właścicielami pozostałych obiektów.
W jakich sytuacjach przydają się takie twory? Otóż jeżeli podzielilibyśmy nasz projekt na jakieś składowe (sposób podziału jest zwykle sprawą mocno subiektywną), to dobrymi kandydatami na singletony byłyby przede wszystkim te składniki, które obejmowałyby **najszerszy zakres** funkcji. Może to być obiekt aplikacji jako takiej albo też reprezentacje poszczególnych podsystemów - w grach byłyby to: grafika, dźwięk, sieć, AI, itd., w edytorach: moduły obsługi plików, dokumentów, formatowania itp.

Niekiedy zastosowanie singletonów wymuszają warunki zewnętrzne, np. jakieś dodatkowe biblioteki, używane przez program. Tak jest chociażby w przypadku funkcji Windows API odpowiedzialnych za zarządzanie oknami.

Siłą rzeczy singletony stanowią też „punkty zaczepienia" dla całego modelu klas, gdyż ich pola są w większości odwołaniami do innych obiektów: niekiedy do wielu drobnych, ale częściej do kilku kolejnych zarządców, czyli następnego, niższego poziomu hierarchii zawierania się obiektów.

O relacji zawierania się (agregacji) będziemy jeszcze szerzej mówić.

Przykłady wykorzystania

Najbardziej oczywistym przykładem singletonu może być całościowy **obiekt programu**, a więc klasa w rodzaju `CApplication` czy `CGame`. Będzie ona nadrzędnym obiektem wobec wszystkich innych, a także przechowywała będzie globalne dane dotyczące aplikacji jako całości. To może być chociażby ścieżka do jej katalogu, ale także kluczowe informacje otrzymane od bibliotek Windows API, DirectX czy jakichkolwiek innych.

Jeżeli chodzi o inne możliwe singletony, to z pewnością będą to zarządcy poszczególnych modułów; w grach są to obiekty klas o tak wiele mówiących nazwach jak `CGraphicsSystem`, `CSoundSystem`, `CNetworkSystem` itp., podobne twory można też wyróżnić w programach użytkowych.
Wszystkie te klasy występują w pojedynczych instancjach, gdyż unikatowa jest ich rola. Kwestią otwartą jest natomiast ich ewentualna podległość najbardziej nadrzędnemu obiektowi aplikacji - na przykład w ten sposób:

```
class CGame
{
   private:
         CGraphicsSystem* m_pGFX;
         CSoundSystem*    m_pSFX;
         CNetworkSystem*  m_pNet;
         // itd.

   // (resztę składowych pominiemy)
};

// jedna jedyna instancja powyższej klasy
extern CGame* g_pGra;                                  // [87]
```

Równie dobrze mogą być bowiem samodzielnymi obiektami, dostępnymi poprzez swoje własne zmienne globalne - bez pośrednictwa obiektu głównego. Obydwa podejścia są w zasadzie równie dobre (może z lekkim wskazaniem na pierwsze, jako że nie zapewnia takiej swobody w dostępie do podsystemów z zewnątrz).

Dlaczego jednak w ogóle stosować singletony, jeżeli i tak będą one tylko pojedynczymi kopiami swoich pól? Przecież podobne efekty można uzyskać stosując zmienne globalne oraz zwyczajne funkcje w miejsce pól i metod takiego obiektu-jedynaka.
To jednak tylko część prawdy. Namnożenie zmiennych i funkcji poza zasadniczą, obiektową strukturą programu narusza zasady OOPu, i to aż podwójnie. Po pierwsze, nie unikniemy w ten sposób wyraźnego oddzielenia danych od kodu, a po drugie nie zapewnimy im ochrony przed niepowołanym dostępem, co zwiększa ryzyko błędów. Wreszcie, mieszamy wtedy dwa style programowania, a to nieuchronnie prowadzi do bałaganu w kodzie, jego niespójności, trudności w rozbudowie i konserwacji oraz całej rzeszy innych plag, przy których te egipskie mogą zdawać się dziecinną igraszką ;D

Używanie singletonów jest zatem nieodzowne. Przydałoby się więc znaleźć jakiś dobry sposób ich implementacji, bo chyba domyślasz się, że zwykłe zmienne globalne nie są tutaj szczytem marzeń. No, a jeśli nawet nie zastanowiłeś się nad tym, to właśnie masz precedens porównawczy - przedstawię bowiem nieco lepszą drogę na realizację pomysłu pojedynczych obiektów w C++.

Praktyczna implementacja z użyciem składowych statycznych

Nawet najlepszy pomysł nie jest zbyt wiele wart, jeżeli nie można jego skutków zobaczyć w działaniu. Singletony można na szczęście zaimplementować aż na kilka sposobów, różniących się wygodą i bezpieczeństwem.

Najprostszy, z wykorzystaniem globalnego wskaźnika na obiekt lub globalnej zmiennej obiektowej, posiada kilka wad, związanych przede wszystkim z kontrolą nad tworzeniem oraz niszczeniem obiektu. Dlatego lepiej zastosować tutaj inne rozwiązanie, oparte na składowych statycznych klas.

Statyczne składowe są przypisane do klasy jako całości, a nie do jej poszczególnych instancji (obiektów).

Deklarujemy je przy pomocy słowa kluczowego `static`. Wówczas pełni więc ono inną funkcję niż ta, którą znaliśmy dotychczas.

[87] Pamiętajmy, że zmienne zadeklarowane w pliku nagłówkowym z użyciem `extern` wymagają jeszcze przydzielenia do odpowiedniego modułu kodu poprzez deklarację bez wspomnianego słówka. Powyższy sposób nie jest zresztą najlepszą metodą na zaimplementowanie singletonu - bardziej odpowiednią poznamy za chwilę.

Podstawową cechą składowych statycznych jest to, że do skorzystania z nich **nie jest potrzebny** żaden obiekt macierzystej klasy. Odwołujemy się do nich, podając po prostu nazwę klasy oraz oznaczenie składowej, w ten oto sposób:

```
nazwa_klasy::składowa_statyczna
```

Możliwe jest także tradycyjne użycie obiektu danej klasy lub wskaźnika na niego oraz operatorów wyłuskania . lub ->. We wszystkich przypadkach efekt będzie ten sam. Musimy jakkolwiek pamiętać, że nadal obowiązują tutaj specyfikatory praw dostępu, więc jeśli powyższy kod umieścimy poza metodami klasy, to będzie on poprawny tylko dla składowych zadeklarowanych jako `public`.

Bliższe poznanie statycznych elementów klas wymaga rozróżnienia spośród nich pól i metod. Działanie modyfikatora `static` jest bowiem nieco inne dla danych oraz dla kodu. I tak statyczne pola są czymś w rodzaju zmiennych **globalnych dla klasy**. Można się do nich odwoływać z każdej metody, a także z klas pochodnych i/lub z zewnątrz - zgodnie ze specyfikatorami praw dostępu. Każde odniesienie do statycznego pola będzie jednak dostępem do **tej samej zmiennej**, rezydującej w **tym samym miejscu pamięci**. W szczególności poszczególne obiekty danej klasy nie będą posiadały własnej kopii takiego pola, bo będzie ono istniało tylko w **jednym egzemplarzu**.
Podobieństwo do zmiennych globalnych przejawia się w jeszcze jednym aspekcie: mianowicie statyczne pola muszą zostać w podobny sposób przydzielone do któregoś z modułów kodu w programie. Ich deklaracja w klasie jest bowiem odpowiednikiem deklaracji `extern` dla zwykłych zmiennych. Odpowiednia definicja w module wygląda zaś następująco:

```
typ nazwa_klasy::nazwa_pola [= wartość_początkowa];
```

Kwalifikatora *nazwa_klasy*:: możemy tutaj wyjątkowo użyć nawet wtedy, kiedy nasze pole nie jest publiczne. Spostrzeżmy też, iż nie korzystamy już ze słowa `static`, jako że poza definicją klasy ma ono odmienne znaczenie.

Statyczność metod polega natomiast na ich niezależności od jakiegokolwiek obiektu danej klasy. Metody opatrzone kwalifikatorem `static` możemy bowiem wywoływać **bez konieczności posiadania instancji** klasy. W zamian za to musimy jednak zaakceptować fakt, iż nie posiadamy dostępu do wszelkich niestatycznych składników (zarówno pól, jak i metod) naszej klasy. To aczkolwiek dość naturalne: jeśli wywołanie funkcji statycznej może obejść się bez obiektu, to skąd moglibyśmy go wziąć, aby skorzystać z niestatycznej składowej, która przecież takiego obiektu wymaga? Otóż właśnie nie mamy skąd, gdyż **w metodach statycznych nie jest dostępny wskaźnik `this`**, reprezentujący aktualny obiekt klasy.

No dobrze, ale w jaki sposób statyczne składowe klas mogą nam pomóc w implementacji singletonów?… Cóż, to dosyć proste. Zauważ, że takie składowe są unikalne w skali całej klasy - tak samo, jak unikalny jest pojedynczy obiekt singletonu. Możemy zatem użyć ich, by sprawować kontrolę nad naszym jedynym i wyjątkowym obiektem.
Najpierw zadeklarujemy więc statyczne pole, którego zadaniem będzie przechowywanie wskaźnika na ów kluczowy obiekt:

```
// *** plik nagłówkowy ***

// klasa singletonu
class CSingleton
{
   private:
```

```
        // statyczne pole, przechowujące wskaźnik na nasz jedyny obiekt
        static CSingleton* ms_pObiekt;          // [88]

    // (tutaj będą dalsze składowe klasy)
};

// *** moduł kodu ***
// trzeba rzecz jasna dołączyć tutaj nagłówek z definicją klasy

// inicjujemy pole wartością zerową (NULL)
CSingleton* CSingleton::ms_pObiekt = NULL;
```

Deklarację pola umieściliśmy w sekcji `private`, aby chronić je przed niepowołaną zmianą. W takiej sytuacji potrzebujemy jednak metody dostępowej do niego, która zresztą także będzie statyczna:

```
// *** wewnątrz klasy CSingleton ***

public:
    static CSingleton* Obiekt()
    {
        // tworzymy obiekt, jeżeli jeszcze nie istnieje
        // (tzn. jeśli wskaźnik ms_pObiekt ma początkową wartość NULL)
        if (ms_pObiekt == NULL) CSingleton();

        // zwracamy wskaźnik na nasz obiekt
        return ms_pObiekt;
    }
```

Oprócz samego zwracania wskaźnika metoda ta sprawdza, czy żądany przez nasz obiekt faktycznie istnieje; jeżeli nie, jest tworzony. Jego kreacja następuje więc przy pierwszym użyciu.
Odbywa się ona poprzez bezpośrednie wywołanie konstruktora… którego na razie nie mamy (jest domyślny)! Czym prędzej naprawmy zatem to niedopatrzenie, przy okazji definiując także destruktor:

```
// *** wewnątrz klasy CSingleton ***

private:
    CSingleton()      { ms_pObiekt = this; }
public:
    ~CSingleton()     { ms_pObiekt = NULL; }
```

Spore zdziwienie może budzić niepubliczność konstruktora. W ten sposób jednak zabezpieczamy się przed utworzeniem więcej niż jednej kopii naszego singletonu. Uprawniona do wywołania prywatnego konstruktora jest bowiem tylko składowa klasy, czyli metoda `CSingleton::Obiekt()`. Wszelkie zewnętrzne próby stworzenia obiektu klasy `CSingleton` zakończą się więc błędem kompilacji, zaś jedyny jego egzemplarz będzie dostępny wyłącznie poprzez wspomnianą metodę.

Powyższy sposób jest zatem odpowiedni dla obiektu stojącego na samym szczycie hierarchii w aplikacji, a więc dla klas w rodzaju `CApplication`, `CApp` czy `CGame`. Jeżeli zaś chcemy mieć wygodny dostęp do obiektów leżących niżej, zawartych wewnątrz innych, wtedy nie możemy oczywiście uczynić konstruktora prywatnym. Wówczas warto więc skorzystać z innych rozwiązań, których jednak nie chciałem tutaj przedstawiać ze

[88] Przedrostek `s_` wskazuje, że dana zmienna jest statyczna. Tutaj został on połączony ze zwyczajowym `m_`, dodawanym do nazw prywatnych pól.

względu konieczność znacznie większej znajomości języka C++ do ich poprawnego zastosowania[89].

Musimy jeszcze pamiętać, aby usunąć obiekt, gdy już nie będzie nam potrzebny - robimy to w zwyczajny sposób, poprzez operator `delete`:

```
delete CSingleton::Obiekt();
```

To konieczne - skoro chcemy zachować kontrolę nad tworzeniem obiektu, to musimy także wziąć na siebie odpowiedzialność za jego zniszczenie.

Na koniec wypadałoby zastanowić się, czy stosowanie powyższego rozwiązania (albo podobnych, gdyż istnieje ich więcej) jest na pewno konieczne. Być może sądzisz, że można się spokojnie bez nich obyć - i chwilowo masz rzeczywiście rację! Kiedy nasze programy są zdeterminowane od początku do końca, zawarte w całości w funkcji `main()`, łatwo jest zapanować nad życiem singletonu. Gdy jednak rozpoczniemy programować aplikacje okienkowe dla Windows, sterowane zewnętrznymi zdarzeniami, wtedy przebieg programu nie będzie już taki oczywisty. Powyższy sposób na implementację singletonu będzie wówczas znacznie użyteczniejszy.

Obiekty zasadnicze

Drugi rodzaj obiektów skupia te, które stanowią największy oraz najważniejszy fragment modelu w każdym programie. Obiekty zasadnicze są jego żywotną tkanką, wykonującą wszelkie zadania przewidziane w aplikacji.

Obiekty zasadnicze to główny budulec programu stworzonego według zasad OOP. Wchodząc w zależności między sobą oraz przekazując dane, realizują one wszystkie funkcje aplikacji.

Budowanie sieci takich obiektów jest więc lwią częścią procesu tworzenia obiektowej struktury programu. Definiowanie odpowiednich klas, związków między nimi, korzystanie z dziedziczenia, metod wirtualnych i polimorfizmu - wszystko to dotyczy właśnie obiektów zasadniczych. Zagadnienie ich właściwego stosowania jest zatem niezwykle szerokie - zajmiemy się nim dokładniej w kolejnych paragrafach tego podrozdziału.

Obiekty narzędziowe

Ostatnia grupa obiektów jest oczkiem w głowie programistów, zajmujących się jedynie „klepaniem kodu" wedle projektu ustalonego przez kogoś innego. Z kolei owi projektanci w ogóle nie zajmują się nimi, koncentrując się wyłącznie na obiektach zasadniczych. W swojej karierze jako twórcy oprogramowania będziesz jednak często wcielał się w obie role, dlatego znajomość wszystkich rodzajów obiektów z pewnością okaże się pomocna.

Czym więc są obiekty należące do opisywanego rodzaju? Naturalnie, najlepiej wyjaśni to odpowiednia definicja :D

Obiekty narzędziowe, zwane też **pomocniczymi** lub **konkretnymi**[90], reprezentują pewien nieskomplikowany typ danych. Zawierają pola służące przechowywaniu jego danych oraz metody do wykonywania nań prostych operacji.

[89] Jeden z najlepszych sposobów został opisany w rozdziale 1.3, *Automatyczne singletony*, książki *Perełki programowania gier, tom 1*.
[90] Autorem tej ostatniej, dziwnej nazwy jest Bjarne Stroustrup i tylko dlatego ją tutaj podaję :)

Nazwa tej grupy obiektów dobrze oddaje ich rolę: są one tylko pomocniczym konstrukcjami, ułatwiającymi realizację niektórych algorytmów. Często zresztą traktuje się je podobnie jak typy podstawowe - zwłaszcza w C++.

Obiekty narzędziowe posiadają wszakże kilka znaczacych cech:

- istnieją same dla siebie i nie wchodzą w interakcje z innymi, równolegle istniejącymi obiektami. Mogą je wprawdzie zawierać w sobie, ale nie komunikują się samodzielnie z otoczeniem
- ich czas życia jest ograniczony do zakresu, w którym zostały zadeklarowane. Zazwyczaj tworzy się je poprzez zmienne obiektowe, w takiej też postaci (a nie poprzez wskaźniki) zwracają je funkcje
- nierzadko zawierają publiczne pola, jeżeli możliwe jest ich bezpieczne ustawianie na dowolne wartości. W takim wypadku typy narzędziowe definiuje się zwykle przy użyciu słowa `struct`, gdyż uwalnia to od stosowania specyfikatora `public`, który w typach strukturalnych jest domyślnym (w klasach, definiowanych poprzez `class`, domyślne prawa to `private`; poza tym oba słowa kluczowe niczym się od siebie nie różnią)
- posiadają najczęściej kilka konstruktorów, ale ich przeznaczenie ogranicza się zazwyczaj do wstępnego ustawienia pól na wartości podane w parametrach. Destruktory są natomiast rzadko używane - zwykle wtedy, gdy obiekt sam alokuje dodatkową pamięć i musi ją zwolnić
- metody obiektów narzedziowych są zwykle proste obliczeniowo i krótkie w zapisie. Ich implementacja jest więc umieszczana bezpośrednio w definicji klasy. Bezwzględnie stosuje się też metody stałe, jeżeli jest to możliwe
- obiekty należące do opisywanego rodzaju prawie nigdy nie wymagają użycia dziedziczenia, a więc także metod wirtualnych i polimorfizmu
- jeżeli ma to sens, na rzecz tego rodzaju obiektów dokonywane jest przeładowywanie operatorów, aby mogły być użyte w stosunku do nich. O tej technice programistycznej będziemy mówić w jednym z dalszych rozdziałów
- nazewnictwo klas narzędziowych jest zwykle takie samo, jak normalnych typów sklarnych. Nie stosuje się więc zwyczajowego przedrostka `C`, a całą nazwę zapisuje tą samą wielkością liter - małymi (jak w Bibliotece Standardowej C++) lub wielkimi (według konwencji Microsoftu)

Bardzo wiele typów danych może być reprezentowanych przy pomocy odpowiednich obiektów narzędziowych. Z jednym z takich obiektów masz zresztą stale do czynienia: jest nim typ `std::string`, będący niczym innym jak właśnie klasą, której rolą jest odpowiednie „opakowanie” łańcucha znaków w przyjazny dla programisty interfejs.

Takie obudowywanie nazywamy **enkapsulacją**.

Klasa ta jest także częścią Standardowej Biblioteki Typów C++, którą poznamy szczegółowo po zakończeniu nauki samego języka. Należą do niej także inne typy, które z pewnością możemy uznać za narzędziowe, jak na przykład `std::complex`, reprezentujący liczbę zespoloną czy `std::bitset`, będący ciągiem bitów. Matematyka dostarcza zresztą największej liczby kandydatów na potencjalne obiekty narzędziowe. Wystarczy pomyśleć o wektorach, macierzach, punktach, prostokątach, prostych, powierzchniach i jeszcze wielu innych pojęciach. Nie są one przy tym jedynie obrazowym przykładem, lecz niedzownym elementem programowania - gier w szczególności. Większość bibliotek zawiera je więc gotowe do użycia; sporo programistów definiuje dlań jednak własne klasy.

Zobaczmy zatem, jak może wyglądać taki typ w przypadku trójwymiarowego wektora:

```
#include <cmath>

struct VECTOR3
```

```
{
   // współrzędne wektora
   float x, y, z;

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

   // konstruktory
   VECTOR3()                                    { x = y = z = 0.0; }
   VECTOR3(float fX, float fY, float fZ)        { x = fX; y = fY; z = fZ; }

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

   // metody
   float Dlugosc() const { return sqrt(x * x + y * y + z * z); }
   void Normalizuj()
   {
         float fDlugosc = Dlugosc();

         // dzielimy każdą współrzędną przez długość
         x /= fDlugosc; y /= fDlugosc; z /= fDlugosc;
   }

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

   // tutaj można by się pokusić o przeładowanie operatorów +, -, *, /,
   // =, +=, -=, *=, /=, == i != tak, żeby przy ich pomocy wykonywać
   // działania na wektorach. Ponieważ na razie tego nie umiemy, więc
   // musimy z tym poczekać :)
};
```

Najwięcej kontrowersji wzbudza pewnie to, że pola `x`, `y`, `z` są publicznie dostępne. Ma to jednak solidne uzasadnienie: ich zmiana jest rzeczą naturalną dla wektora, zaś zakres dopuszczalnych wartości nie jest niczym ograniczony (mogą nimi być dowolne liczby rzeczywiste). Ochrona, którą zwykle zapewniamy przy pomocy metod dostępowych, byłaby zatem niepotrzebnym pośrednikiem.
Użycie powyższej klasy/struktury (jak kto woli…) wymaga oczywiście utworzenia jej instancji. Przy prostym zestawie danych, jaki ona reprezentuje, nie potrzeba jednak poświęcać pieczołowitej uwagi na tworzenie i niszczenie obiektów, zatem wystarczą nam zwykłe zmienne obiektowe zamiast wskaźników. Nawet więcej - możemy potraktować `VECTOR3` identycznie jak typy wbudowane i napisać na przykład funkcję obliczającą oba rodzaje iloczynów wektorów:

```
float IloczynSkalarny(VECTOR3 vWektor1, VECTOR3 vWektor2)
{
   // iloczyn skalarany jest sumą iloczynów odpowiednich współrzędnych
   // obu wektorów

   return  (vWektor1.x * vWektor2.x
           + vWektor1.y * vWektor2.y
           + vWektor1.z * vWektor2.z);
}

VECTOR3 IloczynWektorowy(VECTOR3 vWektor1, VECTOR3 vWektor2)
{
   VECTOR3 vWynik;

   // iloczyn wektorowy ma za to bardziej skomplikowaną formułkę :)
   vWynik.x = vWektor1.y * vWektor2.z - vWektor2.y * vWektor1.z;
   vWynik.y = vWektor2.x * vWektor1.z - vWektor1.x * vWektor1.z;
   vWynik.z = vWektor1.x * vWektor2.y - vWektor2.x * vWektor1.y;
```

```
        return vWynik;
    }
```

Te operacje mają zresztą niezliczone zastosowania w programowaniu trójwymiarowych gier, zatem ich implementacja ma głęboki sens :)

Spokojnie możemy w tych funkcjach pobierać i zwracać obiekty typu `VECTOR3`. Koszt obliczeniowy tych działań będzie bowiem niemal taki sam, jak dla pojedynczych liczb.

W przypadku parametrów funkcji stosujemy jednak referencje, które optymalizują kod, uwalniając od przekazania nawet tych skromnych kilkunastu bajtów. Zapoznamy się z nimi w następnym rozdziale.

Łańcuchy znaków czy wymysły matematyków to nie są naturalnie wszystkie koncepcje, które można i trzeba realizować jako obiekty narzędziowe. Do innych należą chociażby wszelkie reprezentacje daty i czasu, kolorów, numerów o określonym formacie oraz wszystkie pozostałe, nieelementarne typy danych.

Szczególnym przypadkiem obiektów pomocniczych są tak zwane inteligentne wskaźniki (ang. *smart pointers*). Ich zadaniem jest zapewnienie dodatkowej funkcjonalności zwykłym wskaźnikom - obejmuje to na przykład zwolnienie wskazywanej przez nie pamięci w sytuacjach wyjątkowych czy też zliczanie odwołań do „opakowanych” nimi obiektów.

Definiowanie odpowiednich klas

Tworzenie obiektowego modelu programu przebiega zwykle dwuetapowo. Jednym z zadań jest identyfikacja klas, które będą się nań składały, oraz pól i metod, które zostaną zawarte w ich definicjach. Drugim jest określenie związków pomiędzy tymi klasami, dzięki którym aplikacja mogłaby realizować zaplanowane czynności.
Przestrzeganie powyższej kolejności nie jest ściśle konieczne. Oczywiście, mając już kilka zdefiniowanych klas, można pewnie prościej połączyć je właściwymi relacjami. Równie dobre jest jednak wyjście od tychże relacji i korzystanie z nich przy definiowaniu klas. Obydwa wspomniane procesy często więc odbywają się jednocześnie.
Ponieważ jednak lepiej jest opisać każdy z nich osobno, zatem od któregoś należy zacząć :) Zdecydowałem tedy, że najpierw poszukamy właściwych klas oraz ich składowych, a dopiero potem zajmiemy się łączeniem ich w odpowiedni model.

Zaprojektowanie kompletnego zbioru klas oznacza konieczność dopracowywania dwóch aspektów każdej z nich:

- **abstrakcji**, czyli opisu tego, **co** dana klasa ma robić
- **implementacji**, to znaczy określenia, **jak** ma to robić

Teraz naturalnie zajmiemy się kolejno obydowami kwestiami.

Abstrakcja

Jeżeli masz pomysł na grę, aplikację użytkową czy też jakikolwiek inny produkt programisty, to chyba najgorszą rzeczą, jaką możesz zrobić, jest natychmiastowe rozpoczęcie jego kodowania. Słusznie mówi się, że co nagle, to po diable; niezbędne jest więc stworzenie **model abstrakcyjnego** zanim przystąpi się do właściwego programowania.

Model abstrakcyjny powinien opisywać założone działanie programu bez precyzowania szczegółów implementacyjnych.

Sama nazwa wskazuje zresztą, że taki model powinien **abstrahować** od kodu. Jego zadaniem jest bowiem odpowiedź na pytanie „Co program ma robić?”, a w przypadku technik obiektowych, „Jakich klas będzie do tego potrzebował i jakie czynności będą przez nie wykonywane?”.
Tym kluczowym sprawom poświęcimy rzecz jasna nieco miejsca.

Identyfikacja klas

Klasy i obiekty stanowią składniki, z których budujemy program. Aby więc rozpocząć tę budowę, należałoby mieć przynajmniej kilka takich cegiełek. Trzeba zatem zidentyfikować możliwe klasy w projekcie.

Muszę cię niestety zmartwić, gdyż w zasadzie nie ma uniwersalnego i zawsze skutecznego przepisu, który pozwałby na wykrycie wszelkich klas, potrzebnych do realizacji programu. Nie powinno to zresztą dziwić: dzisiejsze programy dotykają przecież prawie wszystkich nauk i dziedzin życia, więc podanie niezawodnego sposobu na skonstruowanie każdej aplikacji jest zadaniem porównywalnym z opracowaniem metody pisania książek, które zawsze będą bestsellerami, lub też kręcenia filmów, które na pewno otrzymają Oscara. To oczywiście nie jest możliwe, niemniej dziedzina informatyka poświęcona projektowaniu aplikacji (zwana **inżynierią oprogramowania**) poczyniła w ostatnich latach duże postępy.

Chociaż nadal najlepszą gwarancją sukcesu jest posiadane doświadczenie, intuicja oraz odrobina szczęścia, to jednak początkujący adept sztuki tworzenia programów (taki jak ty :)) nie pozostanie bez pomocy. Programowanie obiektowe zostało przecież wymyślone właśnie po to, aby ułatwić nie tylko kodowanie programów, ale także ich projektowanie - a na to składa się również wynajdywanie klas odpowiednich dla realizowanej aplikacji. Otóż sama idea OOPu jest tutaj sporym usprawnieniem. Postęp, który ona przynosi, jest bowiem związany z oparciem budowy programu o **rzeczowniki**, zamiast czasowników, właściwym programowaniu strukturalnemu. Myślenie kategoriami tworów, bytów, przedmiotów, urządzeń - ogólnie **obiektów**, jest naturalne dla ludzkiego umysłu. Na rzeczownikach opiera się także język naturalny, i to w każdej części świata.
Również w programowaniu bardziej intuicyjne jest podejście skoncentrowane na **wykonawcach** czynności, a nie na czynnościach jako takich. Przykładowo, porównaj dwa poniższe, abstrakcyjne kody:

```
// 1. kod strukturalny
hPrinter = GetPrinter();
PrintText (hPrinter, "Hello world!");

// 2. kod obiektowy
pPrinter = GetPrinter();
pPrinter->PrintText ("Hello world!");
```

Mimo że oba wyglądają podobnie, to wyraźnie widać, że w kodzie strukturalnym ważniejsza jest sama czynność drukowania, zaś jej wykonawca (drukarka) jest kwestią drugorzędną. Natomiast kod obiektowy wyraźnie ją wyróżnia, a wywołanie metody `PrintText()` można przyrównać do wciśnięcia przycisku zamiast wykonywania jakiejś mało trafiającej do wyobraźni operacji.
Jeżeli masz wątpliwość, które podejście jest właściwsze, to pomyśl, co zobaczysz, patrząc na to urządzenie obok monitora - czynność (drukowanie) czy przedmiot (drukarkę)[91]?…

No, ale dosyć już tych luźnych dygresji. Mieliśmy przecież zająć się poszukiwaniem właściwych klas dla naszych programów obiektowych. Odejście od tematu w poprzednim

[91] Oczywiście nie dotyczy to tych, którzy drukarki nie mają, bo oni nic nie zobaczą :D

akapicie było jednak tylko pozorne, gdyż „niechcący" znaleźliśmy całkiem prosty i logiczny sposób, wspomagający identyfikację klas.
Mianowicie, powiedzieliśmy sobie, że OOP przesuwa środek ciężkości programowania z czasowników na rzeczowniki. Te z kolei są także podstawą języka naturalnego, używanego przez ludzi. Prowadzi to do prostego wniosku i jednocześnie drogi do całkiem dobrego rozwiązania dręczacego nas problemu:

Skuteczną pomocą w poszukiwaniu klas odpowiednich dla tworzonego programu może być **opis jego funkcjonowania** w języku naturalnym.

Taki opis stosunkowo łatwo jest sporządzić, pomaga on też w uporządkowaniu pomysłu na program, czyli klarowanym wyrażeniu, o co nam właściwie chodzi :) Przykład takiego raportu może wyglądać choćby w ten sposób:

> Program Graph jest aplikacją przeznaczoną do rysowania wszelkiego rodzaju schematów i diagramów graficznych. Powinien on udostępniać szeroką paletę przykładowych kształtów, używanych w takich rysunkach: bloków, strzałek, drzew, etykiet tekstowych, figur geometrycznych itp. Edytowany przez użytkownika dokument powinien być ponadto zapisywalny do pliku oraz eksportowalny do kilku formatów plików graficznych.

Nie jest to z pewnością zbyt szczegółowa dokumentacja, ale na jej podstawie możemy łatwo wyróżnić sporą ilość klas. Należą do nich przede wszystkim:

- dokument
- schemat
- różne rodzaje obiektów umieszczanych na schematach

Warto też zauważyć, że powyższy opis ukrywa też nieco informacji o związkach między klasami, np. to, że schemat zawiera w sobie umieszczone przez użytkownika kształty.

Zbiór ten z pewnością nie jest kompletny, ale stanowi całkiem dobre osiągnięcie na początek. Daje też pewne dalsze wskazówki co do możliwych kolejnych klas, jakimi mogą być poszczególne typy kształtów składających się na schemat.

Tak więc analiza opisu w języku naturalnym jest dosyć efektywnym sposobem na wyszukiwanie potencjalnych klas, składających się na program. Skuteczność tej metody zależy rzecz jasna w pewnym stopniu od umiejętności twórcy aplikacji, lecz jej stosowanie szybko przyczynia się także do podniesienia poziomu biegłości w projektowaniu programów.

Analizowanie opisu funkcjonalnego programu nie jest oczywiście jedynym sposobem poszukiwania klas. Do pozostałych należy chociażby sprawdzanie klasycznej „listy kontrolnej", zawierającej często występujące klasy lub też próba określenia działania jakiejś konkretnej funkcji i wykrycia związanych z nią klas.

Abstrakcja klasy

Kiedy już w przybliżeniu znamy kilka klas z naszej aplikacji, możemy spróbować określić je bliżej. Pamiętajmy przy tym, że definicja klasy składa się z dwóch koncepcyjnych części:

- publicznego **interfejsu**, dostępnego dla użytkowników klasy
- prywatnej **implementacji**, określającej sposób realizacji zachowań określonych w interfejsie

Całą sztuką w modelowaniu pojedynczej klasy jest skoncentrowanie się na pierwszym z tych składników, będącym jej **abstrakcją**. Oznacza to zdefiniowanie roli, spełnianej przez klasę, bez dokładnego wgłębiania się w to, jak będzie ona tę rolę odgrywała.

Taka abstrakcja może być również przedstawiona w postaci krótkiego, najczęściej jednozdaniowego opisu w języku naturalnym, np.:

> Klasa *Dokument* reprezentuje pojedynczy schemat, który może być edytowany przez użytkownika przy użyciu naszego programu.

Zauważmy, że powyższe streszczenie nic nie mówi choćby o formie, w jakiej nasz dokument-schemat będzie przechowywany w pamięci. Czy to będzie bitmapa, rysunek wektorowy, zbiór innych obiektów albo może jeszcze coś innego?… Wszystkie te odpowiedzi mogą być poprawne, jednak na etapie określania abstrakcji klasy są one poza obszarem naszego zainteresowania.

Abstrakcja klasy jest określeniem roli, jaką ta klasa pełni w programie.

Jawne formułowanie opisu podobnego do powyższego może wydawać się niepotrzebne, skoro i tak przecież będzie on wymagał uszczegółowienia. Posiadanie go daje jednak możliwość prostej kontroli poprawności definicji klasy. Jeżeli nie spełnia ona założonych ról, to najprawdopodobniej zawiera błędy.

Składowe interfejsu klasy

Publiczny interfejs klasy to zbiór metod, które mogą wywoływać jej użytkownicy. Jego określenie jest drugim etapem definiowania klasy i wyznacza zadania, jakie należy wykonać podczas jej implementacji.

Nasza klasa *Dokument* będzie naturalnie zawierała kilka publicznych metod. Co ciekawe, sporo informacji o nich możemy „wyciągnąć" i wydedukować z już raz analizowanego opisu całego programu. Na jego podstawie dają się sprecyzować takie funkcje jak:

- *Otwórz* - otwierającą dokument zapisany w pliku
- *Zapisz* - zachowująca dokument w postaci pliku
- *Eksportuj* - metoda eksportująca dokument do pliku graficznego z możliwością wyboru docelowego formatu

Z pewnością w toku dalszego projektowania aplikacji (być może w trakcie definicji kolejnych klas albo ich związków?) można by znaleźć także inne metody, których umieszczenie w klasie będzie słusznym posunięciem. W każdej sytuacji musimy jednak pamiętać, aby postać klasy zgadzała się z jej abstrakcją.

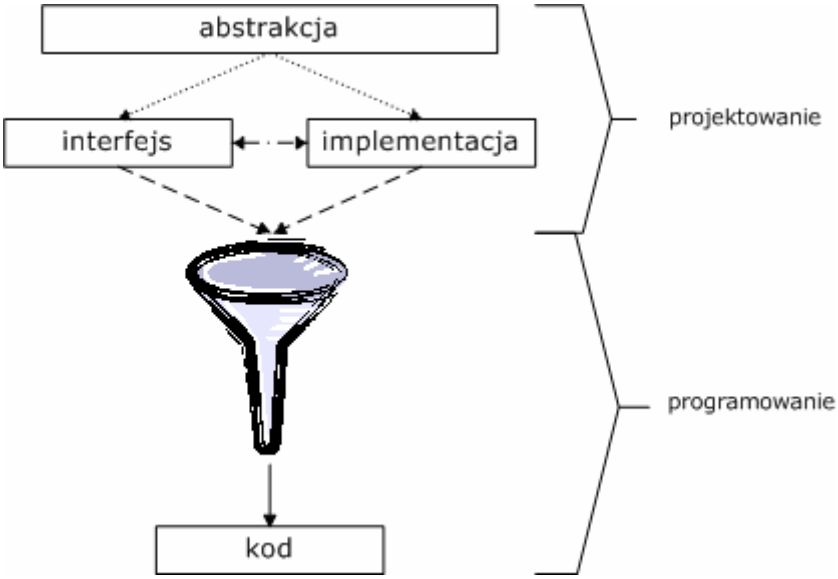
Mówię o tym, gdyż nie powinieneś zapominać, że projektowanie jest procesem cyklicznym, w którym może występować wiele iteracji oraz kilka podejść do tego samego problemu.

Implementacja

Implementacja klasy wyznacza drogę, po jakiej przebiega realizacja zadań klasy, określonych w abstrakcji oraz przybliżonych poprzez jej interfejs. Składają się na nią wszystkie wewnętrzne składniki klasy, niedostępne jej użytkowników - a więc **prywatne pola**, a także **kod poszczególnych metod**.

Dogmaty ścisłej inżynierii oprogramowania mówią, aby dokładne implementacje poszczególnych metod (zwane **specyfikacjami algorytmów**) były dokonywane jeszcze podczas projektowania programu. Do tego celu najczęściej używa się pseudokodu, o którym już kiedyś wspominałem. W nim zwykle zapisuje się wstępne wersje algorytmów metod.
Jednak według mnie ma to sens chyba tylko wtedy, kiedy nad projektem pracuje wiele osób albo gdy nie jesteśmy zdecydowani, w jakim języku programowania będziemy go ostatecznie realizować. Wydaje się, że obie sytuacje na razie nas nie dotyczą :)

W praktyce więc implementacja klasy jest dokonywana podczas programowania, czyli po prostu pisania jej kodu. Można by zatem spierać się, czy faktycznie należy ona jeszcze do procesu projektowania. Osobiście uważam, że to po prostu jego przedłużenie, praktyczna kontynuacja, realizacja - różnie można to nazywać, ale generalnie chodzi po prostu o zaoszczędzenie sobie pracy. Łączenie projektowania z programowaniem jest w tym wypadku uzasadnione.



Schemat 28. Proces tworzenia klasy

Odkładanie implementacji na koniec projektowania, w zasadzie „na styk" z kodowaniem programu, jest zwykle konieczne. Zaimplementowanie klasy oznacza przecież zadeklarowanie i zdefiniowanie wszystkich jej składowych - pól i metod, publicznych i prywatnych. Do tego wymagana jest już pełna wiedza o klasie - nie tylko o tym, co ma robić, jak ma to robić, ale także o jej związkach z innymi klasami.

Związki między klasami

Potęgą programowania obiektowego nie są autonomiczne obiekty, ale współpracujące ze sobą klasy. Każda musi więc wchodzić z innymi przynajmniej w jedną **relację**, czyli związek.

Obecnie zapoznamy się z trzema rodzajami takich związków. Spajają one obiekty poszczególnych klas i umożliwiają realizację założonych funkcji programu.

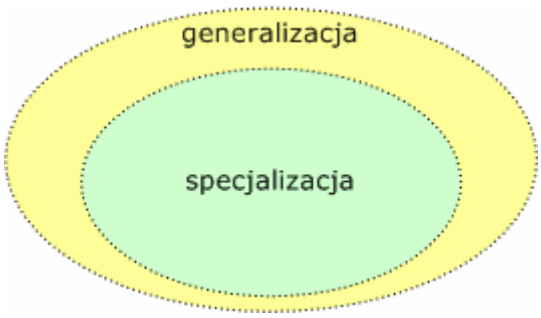
Dziedziczenie i zawieranie się

Pierwsze dwa typy relacji będziemy rozpatrywać razem z tego względu, iż przy ich okazji często występują pewne nieporzumienia. Nie zawsze jest bowiem oczywiste, którego z nich należy użyć w niektórych sytuacjach. Postaram się więc rozwiać te wątpliwości, zanim jeszcze zdążysz o nich pomyśleć ;)

Związek generalizacji-specjalizacji

Relacja ta jest niczym innym, jak tylko znanym ci już dobrze dziedziczeniem. Generalizacja-specjalizacja (ang. *is-a relationship*) to po prostu bardziej uczona nazwa dla tego związku.

W dziedziczeniu występują dwie klasy, z których jedna jest nadrzędna, zaś druga podrzędna. Ta pierwsza to klasa bazowa, czyli **generalizacja**; reprezentuje ona szeroki zbiór jakichś obiektów. Wśród nich można jednak wyróżnić takie, które zasługują na odrębny typ, czyli klasę pochodną - **specjalizację**.



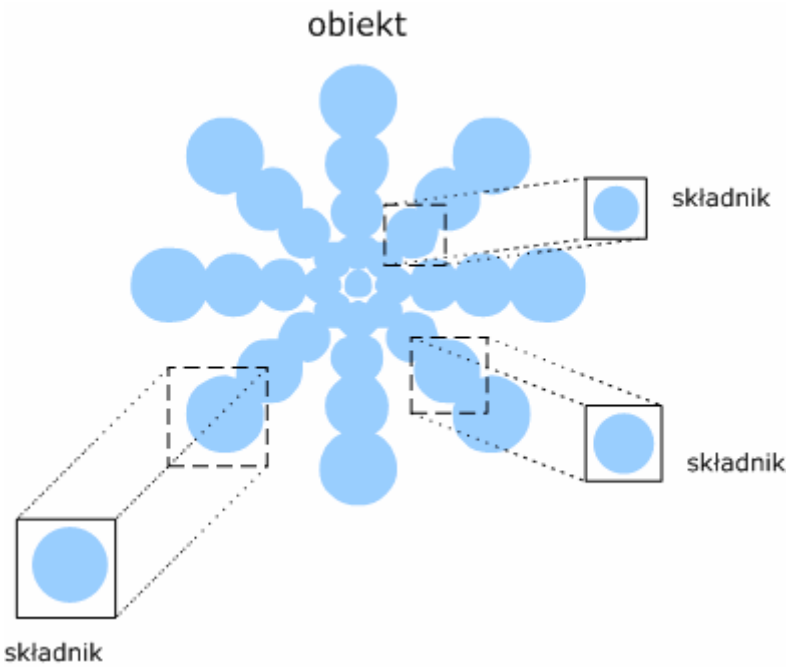
Schemat 29. Ilustracja związku generalizacji-specjalizacji

Klasa bazowa jest często nazywana **nadtypem**, zaś pochodna - **podtypem**. Na schemacie bardzo dobrze widać, dlaczego :D

Najistotniejszą konsekwencją użycia tego rodzaju relacji jest przejęcie przez klasę pochodną całej funkcjonalności, zawartej w klasie bazowej. Jako że jest ona jej bardziej szczegółowym wariantem, możliwe jest też rozszerzenie odziedziczonych możliwości, lecz nigdy - ich ograniczenie.
Klasa pochodna jest więc po prostu pewnym rodzajem klasy bazowej.

Związek agregacji

Agregacja (ang. *has-a relationship*) sugeruje **zawieranie się** jednego obiektu w innym. Mówiąc inaczej, obiekt będący całością składa się z określonej liczby obiektów-składników.



Schemat 30. Ilustracja związku agregacji

Przykładów na podobne zachowanie nie trzeba daleko szukać. Wystarczy chociażby rozejrzeć się po dysku twardym we własnym komputerze: nie dość, że zawiera on foldery

i pliki, to jeszcze same foldery mogą zawierać inne foldery i pliki. Podobne zjawisko występuje też na przykład dla kluczy i wartości w Rejestrze Windows.

Implementacja tej relacji w C++ oznacza umieszczenie w deklaracji obiektu agregatu pola, które będzie reprezentowało jego składnik, np.:

```
// składnik
class CIngredient { /* ... */ };

// obiekt nadrzędny
class CAggregate
{
   private:
         // pole ze składowym składnikiem
         CIngredient* m_pSkladnik;
   public:
         // konstruktor i destruktor
         CAggregate()      { m_pSkladnik = new CIngredient; }
         ~CAggregate()     { delete m_pSkladnik;            }
};
```

Można by tu także zastosować zmienną obiektową, ale wtedy związek stałby się **obligatoryjny**, czyli musiał zawsze występować. Natomiast w przypadku wskaźnika istnienie obiektu nie jest konieczne przez cały czas, więc może być on tworzony i niszczony w razie potrzeby.

Trzeba jednak uważać, aby po każdym zniszczeniu obiektu ustawiać jego wskaźnik na wartość `NULL`. W ten sposób będziemy mogli łatwo sprawdzać, czy nasz składnik istnieje, czy też nie. Unikniemy więc błędów ochrony pamięci.

Odwieczny problem: być czy mieć?

Rozróżnienie pomiędzy dziedziczeniem a zawieraniem może czasami nastręczać pewnych trudności. W takich sytuacjach istnieje na szczęście jedno proste rozwiązanie.

Otóż jeżeli relację pomiędzy dwoma obiektami lepiej opisuje określenie „ma” („zawiera”, „składa się” itp.), to należy zastosować agregację. Kiedy natomiast klasy są naturalnie połączone poprzez stwierdzenie „jest”, wtedy odpowiedniejszym rozwiązaniem jest dziedziczenie.
Co to znaczy? Dokładnie to, co widzisz i o czym myślisz. Należy po prostu sprawdzić, które ze sformułowań:

Klasa1 **jest** rodzajem *Klasa2*.
Klasa1 **zawiera obiekt** typu *Klasa2*.

jest poprawne, wstawiając oczywiście nazwy swoich klas w oznaczonych miejscach, np.:

Kwadrat **jest** rodzajem *Figury*.
Samochód **zawiera obiekt** typu *Koło*.

Mamy więc kolejny przykład na to, że programowanie obiektowe jest bliskie ludzkiemu sposobowi myślenia, co może nas tylko cieszyć :)

Związek asocjacji

Najbardziej ogólnym związkiem między klasami jest **przyporządkowanie**, czyli właśnie asocjacja (ang. *uses-a relationship*). Obiekty, których klasy są połączone taką relacją, posiadają po prostu możliwość wymiany informacji między sobą podczas działania programu.

Praktyczna realizacja takiego związku to zwykle użycie przynajmniej jednego wskaźnika, a najprostszy wariant wygląda w ten sposób:

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

class CBar
{
   private:
         // wskaźnik do połączonego obiektu klasy CFoo
         CFoo* m_pFoo;
   public:
         void UstanowRelacje(CFoo* pFoo) { m_pFoo = pFoo; }
};
```

Łatwo tutaj zauważyć, że zawieranie się jest szczególnym przypadkiem asocjacji dwóch obiektów.

Połączenie klas może oczywiście przybierać znacznie bardziej pogmatwane formy, my zaś powinniśmy je wszystkie dokładnie poznać :D Pomówmy więc o dwóch aspektach tego rodzaju związków: krotności oraz kierunkowości.

Krotność związku

Pod dziwną nazwą **krotności** kryje się po prostu **liczba obiektów**, biorących udział w relacji. Trzeba bowiem wiedzieć, że przy asocjacji dwóch klas możliwe są różne ilości obiektów, występujących z każdej strony. Klasy są przecież tylko typami, z nich są dopiero tworzone właściwe obiekty, które w czasie działania aplikacji będą się ze sobą komunikowały i wykonywały zadania programu.

Możemy więc wyróżnić cztery ogólne rodzaje krotności związku:

- **jeden do jednego**. W takim przypadku pojedynczemu obiektowi jednej z klas odpowiada również pojedynczy obiekt drugiej klasy. Przyporządkowanie jest zatem **jednoznaczne**.
  Z takimi relacjami mamy do czynienia bardzo często. Weźmy na przykład dowolną listę osób - uczniów, pracowników itp. Każdemu numerowi odpowiada tam jedno nazwisko oraz każde nazwisko ma swój unikalny numer. Podobnie „działa" też choćby tablica znaków ANSI.
- **jeden do wielu**. Tutaj pojedynczy obiekt jednej z klas jest przyporządkowany kilku obiektom drugiej klasy. Wygląda to podobnie, jak włożenie skarpety do kilku szuflad naraz - być może w prawdziwym świecie byłoby to trudne, ale w programowaniu wszystko jest możliwe ;)
- **wiele do jednego**. Ten rodzaj związku oznacza, że kilka obiektów jednej z klas jest połączonych z pojedynczym obiektem drugiej klasy.
  Dobrym przykładem są tu rozdziały w książce, których może być wiele w jednej publikacji. Każdy z nich jest jednak przynależny tylko jednemu tomowi.
- **wiele do wielu**. Najbardziej rozbudowany rodzaj relacji to złączenie wielu obiektów od jednej z klas oraz wielu obiektów drugiej klasy.
  Wracając do przykładu z książkami możemy stwierdzić, że związek między autorem a jego dziełem jest właśnie takim typem relacji. Dany twórca może przecież napisać kilka książek, a jednocześnie jedno wydawnictwo może być redagowane przez wielu autorów.

Implementacja wielokrotnych związków polega zwykle na tablicy lub innej tego typu strukturze, przechowującej wskaźniki do obiektów danej klasy. Dokładny sposób zakodowania relacji zależy rzecz jasna także od tego, jaką ilość obiektów rozumiemy pod pojęciem „wiele"…

Pojedyncze związki są natomiast z powodzeniem programowane za pomocą pól, będących wskaźnikami na obiekty.

Widzimy więc, że poznanie obsługi obiektów poprzez wskaźniki w poprzednim rozdziale było zdecydowanie dobrym pomysłem :)

Tam i (być może) z powrotem

Gdy do obiektu jakiejś klasy dodamy pole - wskaźnik na obiekt innej klasy, wtedy utworzymy między nimi relację asocjacji. Związek ten będzie **jednokierunkowy**, gdyż jedynie obiekt posiadający wskaźnik stanie się jego aktywną częścią i będzie inicjował komunikację z drugim obiektem. Ten drugi obiekt może w zasadzie „nie wiedzieć", że jest częścią relacji!

W **związku jednokierunkowym** z pierwszego obiektu możemy otrzymać drugi, lecz odwrotna sytuacja nie jest możliwa.

Naturalnie, niekiedy będziemy potrzebowali obustronnego, wzajemnego dostępu do obiektów relacji. W takim przypadku należy zastosować związek **dwukierunkowy**.

W **związku dwukierunkowym** oba obiekty mają do siebie wzajemny dostęp.

Taka sytuacja często ułatwia pisanie bardziej skomplikowanego kodu oraz organizację przeplywu danych. Jej implementacja napotyka jednak ma pewną, zdawałoby się nieprzekraczalną przeszkodę. Popatrzmy bowiem na taki oto kod:

```
class CFoo
{
   private:
         // wskaźnik do połączonego obiektu CBar
         CBar* m_pBar;
};

class CBar
{
   private:
         // wskaźnik do połączonego obiektu CFoo
         CFoo* m_pFoo;
};
```

Zdawałoby się, że poprawnie realizuje on związek dwukierunkowy klas `CFoo` i `CBar`. Próba jego kompilacji skończy się jednak niepowodzeniem, a to z powodu wskaźnika na obiekt klasy `CBar`, zadeklarowanego wewnątrz `CFoo`. Kompilator analizuje bowiem kod sekwencyjnie, wiersz po wierszu, zatem na etapie definicji `CFoo` nie ma jeszcze bladego pojęcia o klasie `CBar`, więc nie pozwala na zadeklarowanie wskaźnika do niej.
Łatwo przewidzieć, że zamiana obu definicji miejscami w niczym tu nie pomoże. Dochodzimy do paradoksu: aby zdefiniować pierwszą klasę, potrzebujemy drugiej klasy, zaś by zdefniować drugą klasą, potrzebujemy definicji pierwszej klasy! Sytuacja wydaje się być zupełnie bez wyjścia…

A jednak rozwiązanie istnieje, i jest do tego bardzo proste. Skoro kompilator nie wie, że `CBar` jest klasą, trzeba mu o tym zawczasu powiedzieć. Aby jednak znowu nie wpaść w błędne koło, nie udzielimy o `CBar` żadnych bliższych informacji; zamiast definicji zastosujemy **deklarację zapowiadającą**:

```
class CBar;    // rzeczona deklaracja
```

```
// (dalej definicje obu klas, jak w kodzie wyżej)
```

Po tym zabiegu kompilator będzie już wiedział, że `CBar` jest typem (dokładnie klasą) i pozwoli na zadeklarowanie odpowiedniego wskaźnika jako pola klasy `CFoo`.

Niektórzy, by uniknąć takich sytuacji, od razu deklarują deklarują wszystkie klasy przed ich zdefiniowaniem.

Widzimy więc, że związki dwukierunkowe, jakkolwiek wygodniejsze niż jednokierunkowe, wymagają nieco więcej uwagi. Są też zwykle mniej wydajne przy łączeniu nim dużej liczby obiektów. Prowadzi to do prostego wniosku:

Nie należy stosować związków dwukierunkowych, jeżeli w konkretnym przypadku wystarczą relacje jednokierunkowe.

***

Projektowanie aplikacji nawet z użyciem technik obiektowych nie zawsze jest prostym zadaniem. Ten podrozdział powinien jednak stanowić jakąś pomoc w tym zakresie. Nie da się jednak ukryć, że praktyka jest zawsze najlepszym nauczycielem, dlatego zdecydowanie nie powinieneś jej unikać :) Samodzielne zaprojektowanie i wykonanie choćby prostego programu obiektowego będzie bardziej pouczające niż lektura najobszerniejszych podręczników.

Kończacy się podrozdział w wielu miejscach dotykał zagadnień inżynierii oprogramowania. Jeżeli chciałbyś poszerzyć swoją wiedzę na ten temat (a warto), to zapraszam do Materiału Pomocniczego C, *Podstawy inżynierii oprogramowania*.

Podsumowanie

Kolejny bardzo długi i bardzo ważny rozdział :) Zawiera on bowiem dokończenie opisu techniki programowania obiektowego.

Rozpoczęliśmy od mechanizmu dziedziczenia oraz jego roli w ponownym wykorzystywaniu kodu. Zobaczyliśmy też, jak tworzyć proste i bardziej złożone hierarchie klasy.
Dalej było nawet ciekawiej: dzięki metodom wirtualnym i polimorfizmu przekonaliśmy się, że programowanie z użyciem technik obiektowych jest efektywniejsze i prostsze niż dotychczas.
Na koniec zostałeś też obdarzony sporą porcją informacji z zakresu projektowania aplikacji. Dowiedziałeś się więc o rodzajach obiektów, sposobach znajdowania właściwych klas oraz związkach między nimi.

W następnym rozdziale - ostatnim w podstawowym kursie C++ - przypatrzymy się wskaźnikom jako takim, już niekoniecznie w kontekście OOPu. Pomówimy też o pamięci, jej alokowaniu i zwalnianiu.

Pytania i zadania

Na końcu rozdziału nie może naturalnie zabraknąć odpowiedniego pakietu pytań oraz ćwiczeń :)

Pytania

1. Na czym polega mechanizm dziedziczenia i jakie zjawisko jest jego głównym skutkiem?
2. Jaka jest różnica między specyfikatorami praw dostępu do składowych, `private` oraz `protected`?
3. Co nazywamy płaską hierarchią klas?
4. Czym różni się metoda wirtualna od zwykłej?
5. Co jest szczególną cechą klasy abstrakcyjnej?
6. Kiedy klasa jest typem polimorficznym?
7. Na czym polegają polimorficzne zachowania klas w C++?
8. Co to jest RTTI? Na jakie dwa sposoby mechanizm ten umożliwia sprawdzenie klasy obiektu, na który wskazuje dany wskaźnik?
9. Jakie trzy rodzaje obiektów można wyróżnić w programie?
10. Czym jest abstrakcja klasy, a czym jej implementacja?
11. Podaj trzy typy relacji między klasami.

Ćwiczenia

1. Zaprojektuj dowolną, dwupoziomową hierarchię klas.
2. (**Trudne**) Napisz obiektową wersję gry Kółko i krzyżyk z rozdziału 1.5. *Wskazówki*: dobrym kandydatem na obiekt jest oczywiście plansza. Zdefiniuj też klasę graczy, przechowującą ich imiona (niech program pyta się o nie na początku gry).