2

ZAAWANSOWANA OBIEKTOWOŚĆ

Nuda jest wrogiem programistów.
Bjarne Stroustrup

C++ jest zasłużonym członkiem licznej obecnie rodziny języków obiektowych. Oferuje on wszystkie koniecznie mechanizmy, służące praktycznej realizacji idei programowania zorientowanego obiektowo. Poznaliśmy je w dwóch rozdziałach poprzedniej części kursu. Między C++ a innymi językami OOP występują jednak pewne różnice. Nasz język ma wiele specyficznych dla siebie możliwości, które mają za zadanie ułatwienie życia programiście. Często też przyczyniają się do powstania obiektywnie lepszych programów.

W tym rozdziale poznamy tę właśnie stronę OOPu w C++. Przedstawione tu zagadnienia, choć w zasadzie niezbędne do wystarczającej znajomości języka, są w dużej części przydatnymi udogonieniami. Nie niezbędnymi, lecz wielce interesującymi i praktycznymi. Poznanie ich sprawi, że nasze obiektowe programy będą wygodne w konstruowaniu i późniejszej modyfikacji. Programowanie stanie się po prostu łatwiejsze i przyjemniejsze - a to chyba będzie bardzo znaczącym osiągnięciem.
Zobaczmy więc, jakie wyjątkowe konstrukcje OOP oferuje nam C++.

O przyjaźni

W czasie pierwszych spotkań z programowaniem obiektowym wspominałem dość często o jego zaletach, wymieniając wśród nich podział kodu na drobne i łatwe to zarządzania kawałki. Tymi fragmentami (także pod względem koncepcyjnym) są oczywiście klasy. Plusem, jaki niesie za soba stosowanie klas, jest wyodrębnienie kodu i danych w obiekty zajmujące się konkretnymi zadaniami i reprezentującymi konkretne obiekty. Instancje klas współpracują ze sobą i dzięki temu wypełniają zadania aplikacji. Tak to wygląda - przynajmniej w teorii :)

Atutem klas jest niezależność, zwana fachowo hermetyzacją lub enkapsulacją. Objawia się ona tym, iż dana klasa posiada pewien zestaw pól i metod, z którym tylko wybrane są dostępne dla świata zewnętrznego. Jej wewnętrzne sprawy są całkowicie chronione; służą ku temu specyfikatory dostępu, jak `private` i `protected`.
Opatrzone nimi składowe są w zasadzie całkiem odseparowane od świata zewnętrznego, bo ten jest dla nich potencjalnie groźny. Upubliczniając swoje pole klasa narażałaby przecież swoje dane na przypadkowe lub celowe, ale zawsze niepożądane modyfikacje. To tak jakby wyjść z domu i zostawić drzwi niezamknięte na klucz: nie jest to wpradzie bezpośrednie zaproszenie dla złodzieja, ale taka okazja może go uczynić - w myśl znanego powiedzenia.
Ale przecież nie wszyscy są źli - każdy ma przynajmniej kilku **przyjaciół**. Przyjaciel jest to osoba, na którą można liczyć; o której wiemy, że nie zrobi nam nic złego. Większość ludzi uważa, że przyjaźń jest w życiu bardzo ważna - i nie muszą nas do tego

przekonywać żadni socjologowie. Wszyscy wiemy to dobrze z własnego, życiowego doświadczenia.

No dobrze, ale co to ma wspólnego z programowaniem?… Otóż bardzo wiele, zwłaszcza z programowaniem obiektowym. Mianowicie, **klasa także może mieć przyjaciół**: mogą być nimi globalne funkcje, metody innych klas, a także inne klasy w całości. Cóż to jednak znaczy, że klasa ma jakiegoś przyjaciela?… Wyjaśnijmy więc, że:

Przyjaciel (ang. *friend*) danej klasy ma **dostęp do jej wszystkich składników** - także tych **chronionych**, a nawet **prywatnych**.

Jeżeli zatem klasa posiada przyjaciela, to oznacza to, że dała mu „klucze" (dostęp) do swojego „mieszkania" (niepublicznych składowych). Przyjaciel klasy ma do nich prawie takie samo prawo, jak metody tejże klasy. Pewne drobne różnice wyjaśnimy sobie przy okazji osobnego omówienia zaprzyjaźnionych funkcji i klas.

Dowiedzmy się teraz, jak zaprzyjaźnić z klasą jakiś inny element programu. Jest oczywiście i jak zwykle bardzo proste ;) Należy bowiem umieścić w definicji klasy tzw. **deklarację przyjaźni** (ang. *friend declaration*):

```
friend deklaracja_przyjaciela;
```

Słowem kluczowym `friend` poprzedzamy w niej *`deklarację_przyjaciela`*. Tą deklaracją może być:

- prototyp funkcji globalnej
- prototyp metody ze zdefiniowanej wcześniej klasy
- nazwa zadeklarowanej wcześniej klasy

Oto najprostszy i niezbyt mądry przykład:

```
class CFoo
{
   private:
         std::string m_strBardzoOsobistyNapis;

   public:
         // konstruktor
         CFoo()      { m_strBardzoOsobistyNapis = "Kocham C++!"; }

   // deklaracja przyjaźni z funkcją
   friend void Wypisz(CFoo*);
};

// zaprzyjaźniona funkcja
void Wypisz(CFoo* pFoo)
{
   std::cout << pFoo->m_strBardzoOsobistyNapis;
}
```

Zaprzyjaźniony byt - w tym przypadku funkcja - ma tu pełen dostęp do prywatnego pola klasy `CFoo`. Może więc wypisać jego zawartość dla każdego obiektu tej klasy, jaki zostanie mu podany.

Deklaracja przyjaźni w tym przykładzie wydaje się być umieszczona w sekcji `public` klasy `CFoo`. Tak jednak nie jest, gdyż:

Deklaracja przyjaźni może być umieszczona w **każdym miejscu definicji klasy** i zawsze ma **to samo znaczenie**.

Jest więc obojętne, gdzie się ona pojawi. Zwykle piszemy ją albo na początku, albo na końcu klasy, wyróżniając na przykład zmniejszonym wcięciem. Pokazujemy w ten sposób, że nie podlega ona specyfikatorom dostępu.
Nie ma więc czegoś takiego jak „publiczna deklaracja przyjaźni” lub „prywatna deklaracja przyjaźni”. Przyjaciel pozostaje przyjacielem niezależnie od tego, czy się nim chwalimy, czy nie.

Skoro teraz wiemy już z grubsza, czym są przyjaciele klas, omówimy sobie osobno zaprzyjaźnianie funkcji globalnych oraz innych klas i ich metod.

Funkcje zaprzyjaźnione

Najpierw zobaczymy, jak zaprzyjaźnić klasę z funkcją - tak, aby funkcja miała dostęp do niepublicznych składników z danej klasy.

Deklaracja przyjaźni z funkcją

Chcąc uczynić jakąś funkcję przyjacielem klasy, musimy w definicji klasy podać deklarację zaprzyjaźnionej funkcji, poprzedzając ją słowem kluczowym `friend`.

Ilustracją tego faktu nie będzie poniższy przykład. Mamy w nim klasę opisującą okrąg - `CCircle`. Zaprzyjaźniona z nią funkcja `PrzecinajaSie()` sprawdza, czy podane jej dwa okręg mają punkty wspólne:

```
#include <cmath>

class CCircle
{
   private:
         // środek okręgu
         struct { float x, y; } m_ptSrodek;

         // jego promień
         float m_fPromien;

   public:
         // konstruktor
         CCircle (float fPromien, float fX = 0.0f, float fY = 0.0f)
               { m_fPromien = fPromien;
                 m_ptSrodek.x = fX;
                 m_ptSrodek.y = fY;    }

   // deklaracja przyjaźni z funkcją
   friend bool PrzecinajaSie(CCircle&, CCircle&);
};


// zaprzyjaźniona funkcja
bool PrzecinajaSie(CCircle& Okrag1, CCircle& Okrag2)
{
   // obliczamy odległość między środkami
   float fRoznicaX = Okrag2.m_ptSrodek.x - Okrag1.m_ptSrodek.x;
   float fRoznicaY = Okrag2.m_ptSrodek.y - Okrag1.m_ptSrodek.y;
   float fOdleglosc = sqrt(fRoznicaX*fRoznicaX + fRoznicaY*fRoznicaY);
```

```
      // odległość ta musi być mniejsza od sumy promieni, ale większa
      // od ich bezwzględnej różnicy
      return (fOdleglosc < Okrag1.m_fPromien + Okrag2.m_fPromien
              && fOdleglosc > abs(Okrag1.m_fPromien - Okrag2.m_fPromien);
   }
```

Bardzo dobrze widać tu ideę przyjaźni: funkcja `PrzecinajaSie()` ma dostęp do składowych `m_ptSrodek` oraz `m_fPromien` z obiektów klasy `CCircle` - mimo że są prywatne pola klasy. `CCircle` deklaruje jednak przyjaźń z funkcją `PrzecinajaSie()`, a zatem udostępnia jej swoje osobiste dane.

Zauważmy jeszcze, że w deklaracji przyjaźni podajemy **cały prototyp** funkcji, a nie tylko jej nazwę. Możliwe jest bowiem zdefiniowanie kilku funkcji o tej nazwie, np. tak:

```
bool PrzecinajaSie(CCircle&, CCircle&);
bool PrzecinajaSie(CRectangle&, CRectangle&);
bool PrzecinajaSie(CPolygon&, CPolygon&);
// itd. (wraz z ewentualnymi kombinacjami krzyżowymi)
```

Klasa będzie jednak przyjaźniła się tylko z tą funkcją, której deklarację zamieścimy po słowie `friend`. Zapamiętajmy po prostu, że:

Jedna zwykła deklaracja przyjaźni oznacza przyjaźń z **jedną funkcją**.

Na co jeszcze trzeba zwrócić uwagę

Wszystko wydawałoby się raczej proste. Nie zaszkodzi jednak powiedzieć wprost o pewnych „oczywistych” faktach związanych z zaprzyjaźnionymi funkcjami.

Funkcja zaprzyjaźniona nie jest metodą

Jedno słówko `friend` może bardzo wiele zmienić. Porównajmy choćby te dwie klasy:

```
class CFoo
{
   public:
         void Funkcja();
};

class CBar
{
   public:
         friend void Funkcja();
};
```

Różnią się one tylko tym słówkiem… ale jest to różnica znacząca. W pierwszej klasie `Funkcja()` jest jej metodą: zadeklarowaliśmy ją tak, jak wszystkie normalne metody klas. Znamy to już dobrze, gdyż proces definiowania metod poznaliśmy przy pierwszym spotkanie z OOPu. Do pełni szczęścią na leży jeszcze tylko zdefiniować ciało emtody `CFoo::Funkcja()` i wszystko będzie w porządku.
Deklaracja w drugiej klasie jest natomiast opatrzona słówkiem `friend`, które zupełnie zmienia jej znaczenie. `Funkcja()` **nie jest tu metodą klasy `CBar`**. Jest wprawdzie zaprzyjaźniona z nią, ale nie jest jej składnikiem: nie ma dostępu do wskaźnika `this`. Aby z tej zaprzyjaźnionej funkcji mógł być w ogóle jakiś użytek, trzeba jej zapewnić dostęp do obiektu klasy `CBar`, bo jej samej nikt go „nie da”. Wobec braku parametrów funkcji pewnie będzie to wymagało zadeklarowania globalnej zmiennej obiektowej typu `CBar`.

Pamiętaj zatem, iż:

Funkcje zaprzyjaźnione z klasą **nie są jej składnikami**. Nie posiadają dostępu do **wskaźnika `this` tej klasy**, gdyż **nie są jej metodami**.

W praktyce więc należy jakoś podać takiej funkcji obiekt klasy, która się z nią przyjaźni. Zobaczyliśmy w poprzednim przykładzie, że prawie zawsze odbywa się to poprzez parametry. Referencja do obiektu klasy `CCircle` była parametrem zaprzyjaźnionej z nią funkcji `PrzecinajaSie()`. Tylko posiadając dostęp do obiektu klasy, która się z nią przyjaźni, funkcja zaprzyjaźniona może odnieść jakąś korzyść ze swojego uprzywilejowanego statusu.

Deklaracja przyjaźni jest też deklaracją funkcji

Mamy też drugi ważny fakt związany z deklaracją funkcji zaprzyjaźnionej.

Deklaracja przyjaźni jako prototyp funkcji

Otóż, taka deklaracja przyjaźni jest jednocześnie **deklaracją funkcji** jako takiej. Musimy zauważyć, że w zaprezentowanych przykładach funkcje, które były przyjacielami klasy, zostały zdefiniowane dopiero po definicji tejże klasy. Wcześniej kompilator nic o nich nie wiedział - a mimo to pozwolił na ich zaprzyjaźnienie! Czy to jakaś niedoróbka?

Ależ skąd! Kompilator uznaje po prostu deklarację przyjaźni z funkcją także za deklarację samej funkcji. Linijka ze słowem friend pełni więc funkcję prototypu funkcji, która może być swobodnie zdefiniowana w zupełnie innym miejscu. Z kolei wcześniejsze prototypowanie funkcji, przed deklaracją przyjaźni, nie jest konieczne. Mówiąc po ludzku, w poniższym kodzie:

```
bool PrzecinajaSie(CCircle&, CCircle&);

class CCircle
{
   // (ciach - szczegóły)

   friend bool PrzecinajaSie(CCircle&, CCircle&);
};

// gdzieś dalej definicja funkcji...
```

początkowy prototyp funkcji `PrzecinajaSie()`, umieszczony przed definicją `CCircle`, nie jest koniecznie wymagany. Bez niego kompilator skorzysta po prostu z deklaracji przyjaźni jak z normalnej deklaracji funkcji.

Deklaracja przyjaźni z funkcją może być **jednocześnie deklaracją samej funkcji**. Wcześniejsza wiedza kompilatora o istnieniu zaprzyjaźnianej funkcji **nie jest niezbędna**, aby funkcja ta mogła zostać zaprzyjaźniona.

Dodajemy definicję

Najbardziej zaskakujące jest jednak to, że deklarując przyjaźń z jakąś funkcją możemy tę funkcję jednocześnie… zdefiniować! Nic nie stoi na przeszkodzie, aby po zakończeniu deklaracji nie stawiać średnika, lecz otworzyć nawias klamrowy i wpisać treść funkcji:

```
class CVector2D
{
   private:
         float m_fX, m_fY;
```

```
    public:
          CVector2D(float fX = 0.0f, float fY = 0.0f)
                { m_fX = fX;      m_fY = fY; }

    // zaprzyjaźniona funkcja dodająca dwa wektory
    friend CVector2D Dodaj(CVector2D& v1, CVector2D& v2)
          { return CVector2D(v1.m_fX + v2.m_fX, v1.m_fY + v2.m_fY); }
};
```

Nie zapominajmy, że nawet wówczas funkcja zaprzyjaźniona nie jest metodą klasy - pomimo tego, że jej umieszczenie wewnątrz definicji klasy sprawia takie wrażenie. W tym przypadku funkcja `Dodaj()` jest nadal funkcją globalną - wywołujemy ją bez pomocy żadnego obiektu, choć oczywiście przekazujemy jej obiekty `CVector2D` w parametrach i taki też obiekt otrzymujemy z powrotem:

```
CVector2D vSuma = Dodaj(CVector2D(1.0f, 2.0f), CVector2D(0.0f, -1.0f));
```

Umieszczenie definicji funkcji zaprzyjaźnionej w bloku definicji klasy ma jednak pewien skutek. Otóż funkcja staje się wtedy funkcją *inline*, czyli jest rozwijana w miejscu swego wywołania. Przypomina pod tym względem metody klasy, ale jeszcze raz powtarzam, że **metodą nie jest**.

Może najlepiej będzie, jeśli zapamiętasz, że:

Wszystkie **funkcje zdefiniowane wewnątrz definicji klasy** są **automatycznie *inline***, jednak tylko te **bez słówka `friend` są jej metodami**. **Pozostałe są funkcjami globalnymi**, lecz **zaprzyjaźnionymi z klasą**.

Klasy zaprzyjaźnione

Zaprzyjaźnianie klas z funkcjami globalnymi wydaje się może nieco dziwnym rozwiązaniem (gdyż częściowo łamie zaletę OOPu - hermetyzację), ale niejednokrotnie bywa przydatnym mechanizmem. Bardziej obiektowym podejściem jest przyjaźń klas z innymi klasami - jako całościami lub tylko z ich pojedynczymi metodami.

Przyjaźń z pojedynczymi metodami

Wiemy już, że możemy zadeklarować przyjaźń klasy z funkcją globalną. Teraz dowiemy się, że przyjacielem może być także inny rodzaj funkcji - metoda klasy.

Ponownie spojrzyj na odpowiedni przykład:

```
// deklaracja zapowiadająca klasy CCircle
class CCircle;

class CGeometryManager
{
    public:
          bool PrzecinajaSie(CCircle&, CCircle&);
};

class CCircle
{
    // (pomijamy resztę)

    friend bool CGeometryManager::PrzecinajaSie(CCircle&, CCircle&);
};
```

Tym razem funkcja `PrzecinajaSie()` stała się składową klasy `CGeometryManager`. To bardziej obiektowe rozwiązanie - tym bardziej dobre, że nie przeszkadza w zadeklarowaniu przyjaźni z tą funkcją. Teraz jednak klasa z `CCircle` przyjaźni się z metodą innej klasy - `CGeometryManager`. Odpowiednią zmianę (dość naturalną) widać więc w deklaracji przyjaźni.

Przyjaźń z metodami innych klas byłaby bardzo podobna do przyjaźni z funkcjami globalnymi gdyby nie jeden szkopuł. Kompilator musi mianowicie **znać deklarację zaprzyjaźnianej metody** (`CGeometryManager::PrzecinajaSie()`) już wcześniej. To zaś wiążę się z koniecznością zdefiniowania jej macierzystej klasy (`CGeometryManager`). Do tego potrzebujemy jednak informacji o klasie `CCircle`, aby mogła ona wystąpić jako typ agrumentu metody `PrzecinajaSie()`. Rozwiązaniem jest **deklaracja zapowiadająca**, w które informujemy kompilator, że `CCircle` jest klasą, nie mówiac jednak niczego więcej. Z takimi deklaracjami spotkaliśmy się już wcześniej i jeszcze spotkamy się nie raz - szczególnie w kontekście przyjaźni międzyklasowej.

„Chwileczkę! A co z tą zaprzyjaźnianą metodą, `CGeometryManager::PrzecinajaSie()`? Czyżby miała ona nie posiadać dostępu do wskaźnika `this`, mimo że jest funkcją składową klasy?…”
Odpowiedź brzmi: i tak, i nie. Wszystko zależy bowiem od tego, o który wskaźnik `this` nam dokładnie chodzi. Jeżeli o ten pochodzący od `CGeometryManager`, to wszystko jest w jak najlepszym porządku: metoda `PrzecinajaSie()` posiada go oczywiście, zatem ma dostęp do składników swojej macierzystej klasy. Jeśli natomiast mamy na myśli klasę `CCircle`, to faktycznie metoda `PrzecinajaSie()` nie ma dojścia do wskaźnika `this`… tej klasy! Zgadza się to całkowicie z faktem, iż funkcja zaprzyjaźniona **nie jest metodą klasy, która się z nią przyjaźni** - tak więc nie posiada wskaźnika `this` tej klasy (tutaj `CCircle`). Funkcja może być jednak metodą **innej klasy** (tutaj `CGeometryManager`), a dostęp do jej składników będzie mieć zawsze - takie są przecież podstawowe założenia programowania obiektowego.

Przyjaźń z całą klasą

Deklarując przyjaźń jednej klasy z metodami innej klasy, można pójść o krok dalej. Dlaczego na przykład nie powiązać przyjaźnią od razu wszystkich metod pewnej klasy z naszą?… Oczywiście możnaby pracowicie zadeklarować przyjaźń ze wszystkimi metodami tamtej klasy, ale jest prostsze rozwiązanie. Może zaprzyjaźnić jedną klasę z drugą.

Deklaracja przyjaźni z całą klasą jest nad wyraz prosta:

```
friend class nazwa_zaprzyjaźnionej_klasy;
```

Zastępuje ona deklaracje przyjaźni ze wszystkimi metodami klasy o podanej *`nazwie`*, wyszczególnionymi osobno. Taka forma jest poza tym nie tylko krótsza, ale też ma kilka innych zalet.
Wpierw jednak spójrzmy na przykład:

```
class CPoint;

class CRect
{
   private:
         // ...

   public:
         bool PunktWewnatrz(CPoint&);
};
```

```
class CPoint
{
   private:
         float m_fX, m_fY;

   public:
         CPoint(float fX = 0.0f, float fY = 0.0f)
               { m_fX = fX; m_fY = fY; }

   // deklaracja przyjaźni z Crect
   friend class CRect;
};
```

Wyznanie przyjaźni, który czyni klasa `CPoint`, sprawia, że zaprzyjaźniona klasa `CRect` ma pełen dostęp do jej składników niepublicznych. Metoda `CRect::PunktWewnatrz()` może więc odczytać współrzędne podanego punktu i sprawdzić, czy leży on wewnątrz prostokąta opisanego przez obiektt klasy `CRect`.

Zauważmy jednocześnie, że klasa `CPoint` nie ma tutaj podobnego dostępu do prywatnych składowych `CRect`. Klasa `CRect` nie zadeklarowała bowiem przyjaźni z klasą `CPoint`. Wynika stąd bardzo ważna zasada:

Przyjaźń klas w C++ **nie jest automatycznie wzajemna**. Jeżeli klasa A deklaruje przyjaźń z klasą B, to klasa B nie jest od razu także przyjacielem klasy A. Obiekty klasy B mają więc dostęp do niepublicznych danych klasy A, lecz nie odwrotnie.

Dość często aczkolwiek życzymy sobie, aby klasy wzajemnie deklarowały sobie przyjaźń. Jest to jak najbardziej możliwe: po prostu w obu klasach muszą być deklaracje przyjaźni:

```
class CBar;

class CFoo
{
   friend class CBar;
};

class CBar
{
   friend class CFoo;
};
```

Wymaga to zawsze zastosowania deklaracji zapowiadającej, gdyż kompilator musi wiedzieć, że dana nazwa jest klasą, zanim pozwoli na jej zastosowanie w konstrukcji `friend class`. Nie musi natomiast znać całej definicji klasy, co było wymagane dla przyjaźni z pojedynczymi metodami. Gdyby tak było, to wzajemna przyjaźń klas nie byłaby możliwa. Kompilator zadowala się na szczęście samą informacją „`CBar` jest klasą”, bez wnikania w szczegóły, i przyjmuje deklarację przyjaźni z klasą, o której w zasadzie nic nie wie.

Kompilator nie przyjmie natomiast deklaracji przyjaźni z pojedynczą metodą nieznanej bliżej klasy. Sprawia to, że wybiórcza przyjaźń dwóch klas nie jest możliwa, bo wymagałaby niemożliwego: zdefiniowania pierwszej klasy przed definicją drugiej oraz zdefiniowania drugiej przed definicją pierwszej. To oczywiście niemożliwe, a kompilator nie zadowoli się niestety samą deklaracją zapowiadającą - jak to czyni przy deklarowaniu całkowitej przejaźni (`friend class` *`klasa`*`;`).

Jeszcze kilka uwag

Przyjaźń nie jest szczegolnie zawiłym aspektem programowania obiektowego w C++. Wypada jednak nieco uściślić jej wpływ na pozostałe elementy OOPu.

Cechy przyjaźni klas w C++

Przyjaźń klas w C++ ma trzy znaczące cechy, na które chcę teraz zwrócić uwagę.

Przyjaźń nie jest automatycznie wzajemna

W prawdziwym życiu ktoś, kogo uważamy za przyjaciela, ma zwykle to samo zdanie o nas. To więcej niż naturalne.

W programowaniu jest inaczej. Można to uznać za kolejny argument, iż jest ono zupełnie oderwane od rzeczywistości, a można po prostu przyjąć to do wiadomości. A prawda jest taka:

Klasa deklarująca przyjaźń udostępnia przyjacielowi swoje niepubliczne składowe - lecz nie powoduje to od razu, że klasa zaprzyjaźniona jest tak samo otwarta.

Powiedziałem już, że chcąc stworzyć wzajemny związek przyjaźni trzeba umieścić odpowiednie deklaracje w obu klasach. Wymaga to zawsze zapowiadającej deklaracji przynajmniej jeden z powiązanych klas.

Przyjaźń nie jest przechodnia

Inaczej mówiąc: przyjaciel mojego przyjaciela nie jest moim przyjacielem. Przekładając to na C++:

Jeżeli klasa A deklaruje przyjaźń z klasą B, zaś klasa B z klasą C, to **nie znaczy to**, że klasa C jest od razu przyjacielem klasy A.

Gdybyśmy chcieli, żeby tak było, powinniśmy wyraźnie to zadeklarować:

```
friend class C;
```

Przyjaźń nie jest dziedziczna

Przyjaźń nie jest również dziedziczona. Tak więc przyjaciel klasy bazowej nie jest automatycznie przyjacielem klasy pochodnej. Aby tak było, klasa pochodna musi sama zadeklarować swojego przyjaciela.
Można to uzasadnić na przykład w ten sposób, że **deklaracja przyjaźni nie jest składnikiem klasy** - tak jak metoda czy pole. Nie można więc go odziedziczyć. Inne wytłumaczenie: deklaracja `friend` nie ma przypisanego specyfikatora dostępu (`public`, `private`…), zatem nie wiadomo by było, co z nią zrobić w procesie dziedziczenia; jak wiemy, składniki `private` nie są dziedziczone, a pozostałe owszem[106].

Dwie ostatnie uwagi możemy też uogólnić do jednej:

Klasa ma **tylko tych przyjaciół**, których **sama sobie zadeklaruje**.

[106] Jest tak, gdy stosujemy dziedziczenie publiczne (`class pochodna : public bazowa`), ale tak robimy niemal zawsze.

Zastosowania

Mówiąc o zastosowaniach przyjaźni, musimy rozgraniczyć zaprzyjaźnione klasy i funkcje globalne.

Wykorzystanie przyjaźni z funkcją

Do czego mogą przydać się zaprzyjaźnione funkcje?… Teoretycznie korzyści jest wiele, ale w praktyce na przód wysuwa się jedno główne zastosowanie. To przeciążanie operatorów.

O tym użytecznym mechaniźmie języka będziemy mówić w dalszej części tego rozdziału. Teraz mogę powiedzieć, że jest to sposób na zdefiniowanie własnych działań podejmowanych w stosunku do klas, których obiekty występują w wyrażeniach z operatorami: arytmetycznymi, bitowymi, logicznymi, i tak dalej. Precyzyjniej: chodzi o stworzenie funkcji, które zostaną wykonywane na argumentach operatorów, będących naszymi klasami. Takie funkcje potrzebują często dostępu do prywatnych składników klas, na rzecz których przeciążamy operatory. Tutaj właśnie przydają się funkcje globalne, jako że zapewniają taki dostęp, a jednocześnie swobodę definiowania kolejności argumentów operatora.
Jeśli nie bardzo to rozumiesz, nie przejmuj się. Przeciążanie operatorów jest w rzeczywistości bardzo proste, a zaprzyjaźnione funkcji globalne upraszczają to jeszcze bardziej. Wkrótce sam się o tym przekonasz.

Korzyści czerpane z przyjaźni klas

A co można zyskać zaprzyjaźniając klasy? Tutaj trudniej o konkretną odpowiedź. Wszystko zależy od tego, jak zaprojektujemy swój obiektowy program. Warto jednak wiedzieć, że mamy taką właśnie możliwość, jak zaprzyjaźnianie klas. Jak wszystkie z pozoru nieprzydatne rozwiązania, okaże się ona użyteczna w najmniej spodziewanych sytuacjach.

***

Tą pocieszającą konkluzją zakończyliśmy omawianie przyjaźni klas i funkcji w C++. Kolejnym elementem OOPu, na jakim skupimy swoją uwagę, będą konstruktory. Ich rola w naszym ulubionym języka jest bowiem wcale niebagatelna i nieogranicza się tylko do inicjalizacji obiektów… Zobaczmy sami.

Konstruktory w szczegółach

Konstruktory pełnią w C++ wyjątkowo dużo ról. Choć oczywiście najważniejsza (i w zasadzie jedyną poważną) jest inicjalizacja obiektów - instancji klas, to niejako przy okazji mogą one dokonywać kilku innych, przydatnych operacji. Wszystkie one wiążą się z tym głównym zadaniem.
W tym podrozdziale nie będziemy więc mówić o tym, co robi konstruktor (bo to wiemy), ale jak może to robić. Innymi słowy, dowiesz się, jak wykorzystać różne rodzaje konstruktorów do własnych szczytnych celów programistycznych.

Mała powtórka

Najpierw jednak przyda się małe powtórzenie wiedzy, która będzie nam teraz przydatna. Przy okazji może ją trochę usystematyzujemy; powinno się też wyjaśnić to, co do tej pory mogło być dla ciebie ewentualnie niejasne.

Zaczniemy od przypomnienia konstruktorów, a później procesu inicjalizacji.

Konstruktory

Konstruktor jest specjalną metodą klasy, wywoływaną podczas tworzenia obiektu. Nie jest on, jak się czasem błędnie sądzi, odpowiedzialny za alokację pamięci dla obiektu, lecz tylko za wstępne ustawienie jego pól. Niejako przy okazji może on aczkolwiek podejmować też inne czynności, jak zwykła metoda klasy.

Cechy konstruktorów

Konstruktory tym jednak różnią się od zwykłych metod, iż:

- nie posiadają wartości zwracanej. Konstruktor nic nie zwraca (bo i komu?...), nawet typu pustego, czyli `void`. Zgoda, można się spierać, że wynikiem jego działania jest obiekt, lecz konstruktor nie jest jedynym mechanizmem, który bierze udział w jego tworzeniu: liczy się jeszcze alokacja pamięci. Dlatego też przyjmujemy, że konstruktor nie zwraca wartości. Widać to zresztą w jego deklaracji
- nie mogą być wywoływane za pośrednictwem wskaźnika na funkcje. Przyczyna jest prosta: nie można pobrać adresu konstruktora
- mają mnóstwo ograniczeń co do przydomków w deklaracjach:
  - ✓ nie można ich czynić metodami stałymi (`const`)
  - ✓ nie mogą być metodami wirtualnymi (`virtual`), jako że sposób ich wywoływania w warunkach dziedziczenia jest zupełnie odmienny od obu typów metod: wirtualnych i niewirtualnych. Wspominałym o tym przy okazji dziedziczenia.
  - ✓ nie mogą być metodami statycznymi klas (`static`). Z drugiej strony posiadają unikalną cechę metod statycznych, jaką jest możliwość wywołania bez konieczności posiadania obiektu macierzystej klasy. Konstruktory mają jednak dostęp do wskaźnika `this` na tworzony obiekt, czego nie można powiedzieć o zwykłych metodach statycznych
- nie są dziedziczone z klas bazowych do pochodnych

Widać więc, że konstruktor to bardzo dziwna metoda: niby zwraca jakąś wartość (tworzony obiekt), ale nie deklarujemy mu wartości zwracanej; nie może być wirtualny, ale w pewnym sensie jest; nie może być statyczny, ale posiada cechy metod statycznych; jest funkcją, ale nie można pobrać jego adresu, itd. To wszystko wydaje się nieco zakręcone, lecz wiemy chyba, że nie przeszkadza to wcale w normalnym używaniu konstruktorów. Zamiast więc rozstrząsać fakty, czym te metody są, a czym nie, zajmijmy się ich definiowaniem.

Definiowanie

W C++ konstruktor wyróżnia się jeszcze tym, że jego nazwa odpowiada nazwie klasy, na rzecz której pracuje. Przykładowa deklaracja konstruktora może więc wyglądać tak:

```
class CFoo
{
   private:
         int m_nPole;

   public:
         CFoo(int nPole)    { m_nPole = nPole; }
};
```

Jak widzimy, nie podajemy tu żadnej wartości zwracanej.

Przeciążanie

Zwykłe metody klasy także można przeciążać, ale w przypadku konstruktorów dzieje się to nadzwyczaj często. Znowu posłużymy się przykładem wektora:

```
class CVector2D
{
   private:
         float m_fX, m_fY;

   public:
         // konstruktor, trzy sztuki
         CVector2D()           { m_fX = m_fY = 0.0f; }
         CVector2D(float fDlugosc)
                               { m_fX = m_fY = fDlugosc / sqrt(2); }
         CVector2D(float fX, float fY) { m_fX = fX; m_fY = fY; }
};
```

Definiując przeciążone konstruktory powinniśmy, analogicznie jak w przypadku innych metod oraz zwykłych funkcji, wystrzegać się niejednoznaczności. W tym przypadku powstałaby ona, gdyby ostatni wariant zapisać jako:

```
CVector2D(float fX = 0.0f, float fY = 0.0f);
```

Wówczas mógłby on być wywołany z jednym argumentem, podobnie jak konstruktor nr 2. Kompilator nie zdecyduje, który wariant jest lepszy i zgłosi błąd.

Konstruktor domyślny

Konstruktor domyślny (ang. *default constructor*), zwany też domniemanym, jest to taki konstruktor, który **może być wywołany bez podawania parametrów**.

W klasie powyżej jest to więc pierwszy z konstruktorów. Gdybyśmy jednak całą trójkę zastąpili jednym:

```
CVector2D(float fX = 0.0f, float fY = 0.0f)  { m_fX = fX; m_fY = fY; }
```

to on także byłby konstruktorem domyślnym. Ilość podanych do niego parametrów **może być** bowiem równa zeru. Widać więc, że konstruktor domyślny nie musi być akurat tym, który faktycznie nie posiada parametrów w swej deklaracji (tzw. parametrów formalnych).
Naturalnie, klasa może mieć tylko jeden konstruktor domyślny. W tym przypadku oznacza to, że konstruktor w formie `CVector2D()`, `CVector2D(float fDlugosc = 0.0f)` czy jakikolwiek inny tego typu nie jest dopuszczalny. Powstałaby bowiem niejednoznaczność, a kompilator nie wiedziałby, którą metodę powinien wywoływać.

Za wygeneroowanie domyślnego konstruktora może też odpowiadać sam kompilator. Zrobi to jednak **tylko wtedy**, gdy sami **nie podamy jakiegolwiek innego konstruktora**. Z drugiej strony, nasz własny konstruktor domyślny zawsze przesłoni ten pochodzący od kompilatora. W sumie mamy więc trzy możliwe sytuacje:
- nie podajemy żadnego własnego konstruktora - kompilator automatycznie generuje domyślny konstruktor publiczny
- podajemy własny konstruktor domyślny (jeden i tylko jeden) - jest on używany
- podajemy własne konstruktory, ale żaden z nich nie może być domyślny, czyli wywoływany przez parametrów - wówczas klasa nie ma konstruktora domyślnego

Tak więc tylko w dwóch pierwszych sytuacjach klasa posiada domyślny konstruktor. Jaka jest jednak korzyść z jego obecności? Otóż jest ona w sumie niewielka:
- tylko obiekty posiadające konstruktor domyślny mogą być elementami tablic. Podkreślam: chodzi o **obiekty**, nie o wskaźniki do nich - te mogą być łączone w tablice bez względu na konstruktory

- tylko klasę posiadającą konstruktor domyślny można dziedziczyć bez dodatkowych zabiegów przy konstruktorze klasy pochodnej

Tę drugą zasadę wprowadziłem przy okazji dziedziczenia, choć nie wspominałem o owych „dodatkowych zabiegach”. Będą one treścią tego podrozdziału.

Kiedy wywoływany jest konstruktor

Popatrzmy teraz na sytuacje, w których pracuje knnstruktor. Nie jest ich zbyt wiele, tylko kilka.

Niejawne wywołanie

Niejawne wywołanie (ang. *implicit call*) występuje wtedy, gdy to kompilator wywołuje nasz konstruktor. Jest parę takich sytuacji:

- najprostsza: gdy deklarujemy zmienną obiektową, np.:

```
CFoo Foo;
```

- w momencie tworzenia obiektu, który zawiera w sobie pola będące zmiennymi obiektowymi innych klas
- w chwili tworzenia obiektu klasy pochodnej jest wywoływany konstruktor klasy bazowej

Jawne wywołanie

Konstruktor możemy też wywołać jawnie. Mamy wtedy wywołanie niejawne (ang. *explicit call*), które występuje np. w takich sytuacjach:

- przy konstruowaniu obiektu operatorem `new`
- przy jawnym wywołaniu konstruktora: `nazwa_klasy([parametry])`

W tym drugim przypadku mamy tzw. obiekt chwilowy. Zwracaliśmy taki obiekt, kopiując go do rezultatu funkcji `Dodaj()`, prezentując funkcje zaprzyjaźnione.

Inicjalizacja

Teraz powiemy sobie więcej o inicjalizacji. Jest to bowiem proces ściśle związany z aspektami konstruktorów, które omówimy w tym podrozdziale.

Inicjalizacja (ang. *initialization*) jest to nadanie obiektowi **wartości początkowej** w chwili jego tworzenia.

Kiedy się odbywa

W naturalny sposób inicjalizację wiążemy z deklaracją zmiennych. Odbywa się ona jednak także w innych sytuacjach.

Dwie kolejne związane z funkcjami. Otóż jest to:

- przekazanie wartości poprzez parametr
- zwrócenie wartości jako rezultatu funkcji

Wreszcie, ostatnia sytuacja związana jest inicjalizacją obiektów klas - poznamy ją za chwilę.

Jak wygląda

Inicjalizacja w ogólności wygląda mniej więcej tak:

```
typ zmienna = inicjalizator;
```

inicjalizator może mieć jednak różną postać, w zależności od *typu* deklarowanej *zmiennej*.

Inicjalizacja typów podstawowych

W przypadku zmiennych typow elementarnych sprawa jest najprostsza. W inicjalizatorze podajemy po prostu odpowiednią wartość, jaka zostanie przypisana temu typowi, np.:

```
unsigned nZmienna = 42;
float fZmienna = 10.5;
```

Zauważmy, że bardzo często inicjalizacja związana jest niejawną konwersją wartości do odpowiedniego typu. Tutaj na przykład `42` (typu `int`) zostanie zamienione na typ unsigned, zaś `10.5` (`double`) na typ `float`.

Agregaty

Bardziej złozone typy danych możemy inicjalizować w specjalny sposób, jako tzw. **agregaty**. Agregatem jest tablica innych agregatów (względnie elementów typów podstawowych) lub obiekt klasy, która:
- nie dziedziczy z żadnej klasy bazowej
- posiada tylko składniki publiczne (public, ewentualnie bez specyfikatora w przypadku typów struct)
- nie posiada funkcji wirtualnych
- nie posiada zadeklarowanego konstruktora

Agregaty możemy inicjalizować w specjalny sposób, podając wartości wszystkich ich elementów (pól). Znamy to już tablic, np.:

```
int aTablica[13] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41 };
```

Podobnie może to się odbywać także dla struktur (tudzież klas), spełniających cztery podane warunki:

```
struct VECTOR3 { float x, y, z; };
VECTOR3 vWektor = { 6.0f, 12.5f, 0.0f };
```

W przypadku bardziej skomplikowanych, „zagnieżdżonych" agregatów, będziemy mieli więcej odpowiednich par nawiasów klamrowych:

```
VECTOR3 aWektory[3] = { { 0.0f, 2.0f, -3.0f },
                        { -1.0f, 0.0f, 0.0f },
                        { 8.0f, 6.0f, 4.0f } };
```

Można je aczkolwiek opuścić i napisać te 9 wartości jednym ciągiem, ale przyznasz chyba, że w tej postaci inicjalizacja wygląda bardziej przejrzyście. Po inicjalizatorze widać przynajmniej, że inicjujemy tablicę trój-, a nie dziewięcioelementową.

Inicjalizacja konstruktorem

Ostatni sposób to inicjalizacja obiektu jego własnym konstruktorem - na przykład:

```
std::string strZmienna = "Hmm...";
```

Tak, to jest jak najbardziej taki właśnie przykład. W rzeczywistości kompilator rozwinie go bowiem do:

```
std::string strZmienna("Hmm...");
```

gdyż w klasie `std::string` istnieje odpowiedni konstruktor przyjmujący jeden argument typu napisowego[107]:

```
string(const char[]);
```

Konstruktor jest tu więc wywoływany niejawnie - jest to tak zwany konstruktor konwertujący, któremu przyjrzymy się bliżej w tym rozdziale.

Listy inicjalizacyjne

W definicji konstruktora możemy wprowadzić dodatkowy element - tzw. **listę inicjalizacyjną**:

```
nazwa_klasy::nazwa_klasy([parametry]) : lista_inicjalizacyjna
{
   ciało_konstruktora
}
```

Lista inicjalizacyjna (ang. *initializers' list*) ustala sposób inicjalizacji obiektów tworzonej klasy.

Za pomocą takiej listy możemy zainicjalizować pola klasy (i nie tylko) jeszcze **przed wywołaniem samego konstruktora**. Ma to pewne konsekwencje i bywa przydatne w określonych sytuacjach.

Inicjalizacja składowych

Dotychczas dokonywaliśmy inicjalizacji pól klasy w taki oto sposób:

```
class CVector2D
{
   private:
         float m_fX, m_fY;

   public:
         CVector2D(float fX = 0.0f, float fY = 0.0f)
               { m_fX = fX; m_fY = fY; }
};
```

Przy pomocy listy inicjalizacyjnej zrobimy to samo mniej więcej tak:

```
CVector2D(float fX = 0.0f, float fY = 0.0f) : m_fX(fX), m_fY(fY) { }
```

Jaka jest różnica?

- konstruktor może u nas być pusty. To najprawdopodobniej sprawi, że kompilator zastosuje wobec niego jakąś optymalizację
- działania `m_fX(fX)` i `m_fY(fY)` (zwróćmy uwagę na składnię), mają charakter inicjalizacji pól, podczas gdy przypisania w ciele konstruktora są przypisaniami właśnie
- lista inicjalizacyjna jest wykonywana jeszcze **przed wejściem** w ciało konstruktora i wykonaniem zawartych tam instrukcji

Drugi i trzeci fakt jest bardzo ważny, ponieważ dają nam one możliwość umieszczania w klasie takich pól, które nie moga obyć się bez inicjalizacji, a więc:

[107] W rzeczywistości ten konstruktor wygląda znacznie obszerniej, bo w grę wchodzą jeszcze szablony z biblioteki STL. Nic jednak nie stałoby na przeszkodzie, aby tak to właśnie wyglądało.

- stałych (pól z przydomkiem `const`)
- stałych wskaźników (*typ** `const`)
- referencji
- obiektów, których klasy nie mają domyślnych konstruktorów

Lista inicjalizacyjna gwarantuje, że zostaną one zainicjalizowane we właściwym czasie - podczas tworzenia obiektu:

```
class CFoo
{
   private:
         const float m_fPole;
         // nie może być: const float m_fPole = 42; !!

   public:
         // konstruktor - inicjalizacja pola
         CFoo() : m_fPole(42)
         {
               /* m_fPole = 42;  // też nie może być - za późno!
                                 // m_fPole musi mieć wartość już
                                 // na samym początku wykonywania
                                 // konstruktora */
         }
};
```

Mówiłem też, że inicjalizacja przy pomocy listy inicjalizacyjnej jest szybsza od przypisań w ciele konstruktora. Powinniśmy więc stosować ją, jeżeli mamy taką możliwość, a decyzja na którejś z dwóch rozwiązań nie robi nam różnicy. Zauważmy też, że zapis na liście inicjalizacyjnej jest po prostu krótszy.

W liście inicjalizacyjnej możemy umieszczać nie tylko „czyste” stałe i argumenty konstruktora, lecz także złożone wyrażenia - nawet z wywołaniami metod czy funkcji globalnych. Nie ma więc żadnych ograniczeń w stosunku do przypisania.

Wywołanie konstruktora klasy bazowej

Lista inicjalizacyjna pozwala zrobić coś jeszcze zanim właściwy konstruktor ruszy do pracy. Pozwala to nie tylko na inicjalizację składowych klasy, które tego wymagają, ale także - a może przede wszystkim - wywołanie konstruktorów klas bazowych.

Przy pierwszym spotkaniu z dziedziczeniem mówiłem, że klasa, która ma być dziedziczona, powinna posiadać bezparametrowy konstruktor. Było to spowodowane kolejnością wywoływania konstruktorów: jak wiemy, najpierw pracuje ten z klasy bazowej (poczynając od najstarszego pokolenia), a dopiero potem ten z klasy pochodnej. Kompilator musi więc wiedzieć, jak wywołać konstruktor z klasy bazowej. Jeżeli nie pomożemy mu w decyzji, to uprze się na konstruktor domyslny - bezparametrowy.

Teraz będziemy już wiedzieć, jak można pomóc kompilatorowi. Służy do tego właśnie lista inicjalizacyjna. Oprócz inicjalizacji pól klasy możemy też wywoływać w niej konstruktory klas bazowych. W ten sposób zniknie konieczność posiadania przez nie konstruktora domyślnego.
Oto jak może to wyglądać:

```
class CIndirectBase
{
   protected:
         int m_nPole1;
```

```
   public:
         CIndirectBase(int nPole1) : m_nPole1(nPole) { }
};

class CDirectBase : public CIndirectBase
{
   public:
         // wywołanie konstruktora klasy bazowej
         CDirectBase(int nPole1) : CIndirectBase(nPole1) { }
};

class CDerived : public CDirectBase
{
   protected:
         float m_fPole2;

   public:
         // wywołanie konstruktora klasy bezpośrednio bazowej
         CDerived(int nPole1, float fPole2)
               : CDirectBase(nPole1), m_fPole2(fPole2) { }
};
```

Zwróćmy uwagę szczególnie na klasę `CDerived`. Jej konstruktor wywołuje konstruktor z klasy bazowej bezpośredniej - `CDirectBase`, lecz nie z pośredniej - `CIndirectBase`. Nie ma po prostu takiej potrzeby, gdyż za relacje między konstruktorami klas `CDirectBase` i `CIndirectBase` odpowiada tylko ta ostatnia. Jak zresztą widać, wywołuje ona jedyny konstruktor `CIndirectBase`.

Spójrzmy jeszcze na parametry wszystkich konstruktorów. Jak widać, zachowują one parametry konstruktorów klas bazowych - zapewne dlatego, że same nie potrafią podać dla nich sensownych danych i będą ich żądać od twórcy obiektu. Uzyskane dane przekazują jednak do swoich przodków; powstaje w ten sposób swoista sztafeta, w której dane z konstruktora najniższego poziomu dziedziczenia trafiają w końcu do klasy bazowej. Po drodze są one przekazywane z rąk do rąk i ewentualnie zostawiane w polach klas pośrednich.
Wszystko to dzieje się za pośrednictwem list inicjalizacyjnej. W praktyce ich wykorzystanie eliminuje więc bardzo wiele sytuacji, które wymagają definiowania ciała konstruktora. Sam się zresztą przekonasz, że całe mnóstwo pisanych przez ciebie klas bedzie zawierało puste konstruktory, realizujące swoje funkcje wyłącznie poprzez listy inicjalizacyjne.

Konstruktory kopiujące

Teraz porozmawiamy sobie o kopiowaniu obiektów, czyli tworzeniu ich koncepcyjnych duplikatów. W C++ mamy na to dwie wydzielone rodzaje metod klas:

- konstruktory kopiujące, tworzące nowe obiekty na podstawie już istniejących
- przeciążone operatory przypisania, których zadaniem jest skopiowanie stanu jednego obiektu do drugiego, już istniejącego

Przeciążaniem operatorów zajmiemy się dalszej części rozdziału. W tej sekcji przyjrzymy się natomiast konstruktorom kopiującym.

O kopiowaniu obiektów

Wydawałoby się, że nie ma nic prostszego od skopiowania obiektu. Okazuje się jednak, że często nieodzowne są specjalne mechanizmy temu służące… Sprawdźmy to.

Pole po polu

Gdy mówimy o kopiowaniu obiektów i nie zastanawiamy się nad tym dłużej, to sądzimy, że to po prostu skopiowanie danych - zawartości pól - z jednego obszaru pamięci do drugiego. Przykładowo, spójrzmy na dwa wektory:

```
CVector2D vWektor1(1.0f, 2.0f, 3.0f);
CVector2D vWektor2 = vWektor1;
```

Całkiem słusznie oczekujemy, że po wykonaniu kopiowania `vWektor1` do `vWektor2` oba obiekty będą miały identyczne wartości pól. W przypadku takich struktur danych jak wektory, jest to zupełnie wystarczające. Dlaczego? Otóż wszystkie ich pola są całkowicie odrębnymi zmiennymi - nie mają żadnych koneksji z otaczającym je światem. Trudno przecież oczekiwać, żeby liczby typu `float` robiły cokolwiek innego poza przechowywaniem wartości. Ich proste skopiowanie jest więc właściwym sposobem wykonania kopii wektora - czyli obiektu klasy `CVector2D`.

Samowystarczalne obiekty mogą być kopiowane poprzez dosłowne przepisanie wartości swoich pól.

Gdy to nie wystarcza…

Nie wszyscy obiekty podpadają jednak pod ustanowioną wyżej kategorię. Czy pamiętasz może klasę `CIntArray`, którą pokazałem, omawiając wskaźniki? Jeśli nie, to spójrz jeszcze raz na jej definicję (usprawnioną wykorzystaniem list inicjalizacyjnych):

```
class CIntArray
{
   // domyślny rozmiar tablicy
   static const unsigned DOMYSLNY_ROZMIAR = 5;

   private:
         // wskaźnik na właściwą tablicę oraz jej rozmiar
         int* m_pnTablica;
         unsigned m_uRozmiar;

   public:
         // konstruktory
         CIntArray()                        // domyślny
               : m_uRozmiar(DOMYSLNY_ROZMIAR),
                 m_pnTablica(new int [m_uRozmiar])   { }
         CIntArray(unsigned uRozmiar)  // z podaniem rozmiaru tablicy
               : m_uRozmiar(uRozmiar);
                 m_pnTablica(new int [m_uRozmiar])   { }

         // destruktor
         ~CIntArray()      { delete[] m_pnTablica; }

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

         // pobieranie i ustawianie elementów tablicy
         int Pobierz(unsigned uIndeks) const
               { if (uIndeks < m_uRozmiar)   return m_pnTablica[uIndeks];
                 else                        return 0;                  }
         bool Ustaw(unsigned uIndeks, int nWartosc)
               { if (uIndeks >= m_uRozmiar) return false;
                 m_pnTablica[uIndeks] = uWartosc;
                 return true;                               }

         // inne
```

```
            unsigned Rozmiar() const      { return m_uRozmiar; }
   };
```

Pytanie brzmi: jak skopiować tablicę typu `CIntArray`?... Niby nic prostszego:

```
   CIntArray aTablica1;
   CIntArray aTablica2 = aTablica1; // hmm...
```

W rzeczywistości mamy tu bardzo poważny błąd. Metoda „pole po polu” zupełnie nie sprawdza się w przypadku tej klasy. Problemem jest pole m_pnTablica: jesli skopiujemy ten wskaźnik, to otrzymamy nic innego, jak tylko **kopię wskaźnika**. Będzie się on odnosił **do tego samego obszaru pamięci**. Zamiast więc dwóch fizycznych tablic mamy tylko jedną, a obiekty Tablica1 i Tablica2 to jedynie kopie opakowań dla wskaźnika na tę tablicę. Odwołując się do danych, zapisanych w rzekomo odrębnych tablicach klasy `CIntArray`, faktycznie będziemy odnosić się do **tych samych elementów**! To poważny błąd, co gorsza niewykrywalny aż do momentu wyprodukowania błędnych rezultatów przez program.

Coś więc trzeba z tym zrobić - domyślasz się, że rozwiązaniem są tytułowe konstruktory kopiujące. Jeszcze zanim je poznamy, powinieneś zapamiętać:

Jeżeli obiekt pracuje na jakimś zewnętrznym zasobie (np. pamięci operacyjnej) i posiada do niego odwołanie (np. wskaźnik), to jego klasę koniecznie należy wyposażyć w konstruktor kopiujący. Bez niego zostanie bowiem podczas kopiowanie obiektu zostanie skopiowane samo odwołanie do zasobu (czyli wskaźnik) zamiast stworzenia jego duplikatu (czyli alokacji nowej porcji pamięci).

Trzeba też wiedzieć, że konieczność zdefiniowania konstruktora kopiującego zwykle automatycznie pociąga za sobą wymóg obecności przeciążonego operatora przypisania.

Konstruktor kopiujący

Zobaczmy zatem, jak działają te cudowne konstruktory kopiujące. Jednak oprócz zachwycania się nimi poznamy także sposób ich użycia (definiowania) w C++.

Do czego służy konstruktor kopiujący

Konstruktor kopiujący (ang. *copy constructor*) służy do **tworzenia nowego obiektu** danej klasy na **podstawie już istniejącego**, innego obiektu tej klasy.

Konstruktor ten, jak wszystkie konstruktory, wkracza do akcji podczas kreowania nowego obiektu klasy. Czym się w takim razie różni od zwykłego konstruktora?... Przypomnijmy dwie sporne linijki z poprzedniego paragrafu:

```
   CIntArray aTablica1;
   CIntArray aTablica2 = aTablica1;
```

Pierwsza z nich to normalne stworzenie obiektu klasy `CIntArray`. Pracuje tu zwykły konstruktor, domyślny zresztą.
Natomiast druga linijka może być także zapisana jako:

```
   CIntArray aTablica2 = CIntArray(aTablica1);
```

albo nawet:

```
   CIntArray aTablica2(aTablica1);
```

W niej pracuje konstruktor kopiujący, gdyż dokonujemy tu **inicjalizacji nowego obiektu starym**.

Konstruktor kopiujący jest wywoływany w momencie **inicjalizacji nowotworzonego obiektu przy pomocy innego obiektu** tej samej klasy. Z tego powodu taki konstruktor jest również zwany **inicjalizatorem kopiującym**.

Zaraz, jak to - przecież nie zdefiniowaliśmy dotąd żadnego specjalnego konstruktora! Jak więc mógł on być użyty w kodzie powyżej?
Owszem, to prawda, ale kompilator wykonał robotę za nas. Jeśli nie zdefiniujemy własnego konstruktora kopiującego, to klasa zostanie obdarzona jego najprostszym wariantem. Będzie on wykonywał zwykłe kopiowanie wartości - dla nas całkowicie niewystarczające.

Musimy zatem wiedzieć, jak definiować własne konstruktory kopiujące.

Konstruktor kopiujący a przypisanie - różnica mała lecz ważna

Możesz spytać: a co kompilator zrobi w takiej sytuacji:

```
CIntArray aTablica1;
CIntArray aTablica2;
aTablica1 = aTablica2;           // a co to jest?...
```

Czy w trzeciej linijce także zostanie wywołany konstruktor kopiujący?…

Otóż nie. Nie jest bowiem inicjalizacja (a wtedy przecież pracuje konstruktor kopiujący), lecz zwykłe przypisanie. **Nie tworzymy** tu nowego obiektu, lecz przypisujemy jeden już istniejący obiekt do drugiego istniejącego obiektu. Wobec braku aktu kreacji nie ma tu miejsca dla żadnego konstruktora.
Zamiast tego kompilator posługuje się operatorem przypisania. Jeżeli go przeciążymy (a nauczymy się to robić już w tym rozdziale), zdefiniujemy własną akcję dla przypisywania obiektów. W przypadku klasy `CIntArray` jest to niezbędne, bo nawet obecność konstruktora kopiującego nie spowoduje, że zaprezentowany wyżej kod będzie poprawny. Konstruktorów nie dotyczy przecież przypisanie.

Dlaczego konstruktor kopiujący

Ale w takim razie po co nam konstruktor kopiujący? Przecież jego praca jest w większości normalnych sytuacji równoważna:
- wywołaniu zwykłego konstruktora (czyli normalnemu stworzeniu obiektu)
- wywołaniu operatora przypisania

Czy tak?

Cóż, niezupełnie. W zasadzie zgadza się to tylko dla takich obiektów, dla których wystarczające jest kopiowanie „pole po polu”. Dla nich faktycznie nie potrzeba specjalnego konstruktora kopiującego. Jeśli jednak mamy do czynienia z taką klasą, jak `CIntArray`, konstruktor taki jest konieczny. Sposób jego pracy będzie się różnił od zwykłego przypisania - weźmy choćby pod uwagę to, że konstruktor pracuje na pustym obiekcie, natomiast przypisanie oznacza zastąpienie jednego obiektu drugim…

Dokładniej wyjaśnimy tę sprawę, gdy poznamy przeciążanie operatorów. Teraz zobaczmy, jak możemy zdefiniować własny konstruktor kopiujący.

Definiowanie konstruktora kopiującego

Składnię definicji konstruktora kopiującego możemy zapisać tak:

```
nazwa_klasy::nazwa_klasy([const] nazwa_klasy& obiekt)
```

```
{
   ciało_konstruktora
}
```

Bierze on jeden parametr, będący **referencją do obiektu swojej macierzystej klasy**. Obiekt ten jest podstawą kopiowania - inaczej mówiąc, jest to ten obiekt, którego kopię ma zrobić konstruktor. W inicjalizacji:

```
CIntArray aTablica2 = aTablica1;
```

parametrem konstruktora kopiującego będzie więc `aTablica1`, zaś tworzonym obiektem-kopią `Tablica2`. Widać to nawet lepiej w równoważnej linijce:

```
CIntArray aTablica2(aTablica1);
```

Pozostaje jeszcze kwestia słówka `const` w deklaracji parametru konstruktora. Choć teoretycznie jest ona opcjonalna, to w praktyce trudno znaleźć powód na uzasadnienie jej nieobecności. Bez niej konstruktor kopiujący mógłby bowiem potencjalnie **zmodyfikować kopiowany obiekt**!... Innym skutkiem byłaby też niemożność kopiowania obiektów chwilowych.
Zapamiętaj więc:

Parametr konstruktora kopiującego praktycznie zawsze musi być **stałą referencją**.

Inicjalizator klasy `CIntArray`

Gdy wiemy już, do czego służą konstruktory kopiujące i jak się je definiuje, możemy tę wiedzę wykorzystać. Zdefiniujmy inicjalizator dla klasy, która tak bardzo go potrzebuje - `CIntArray`.

Nie będzie to trudne, jeżeli zastanowimy się wpierw, co ten konstruktor ma robić. Otóż powinien on zaalokować pamięć równą rozmiarowi kopiowanej tablicy oraz przekopiować z niej dane do nowego obiektu. Proste? Zatem do dzieła:

```
#include <cmemory>

CIntArray::CIntArray(const CIntArray& aTablica)
{
   // alokujemy pamięć
   m_uRozmiar = aTablica.m_uRozmiar;
   m_pnTablica = new int [m_uRozmiar];

   // kopiujemy pamięć ze starej tablicy do nowej
   memcpy (m_pnTablica, aTablica.m_pnTablica, m_uRozmiar * sizeof(int));
}
```

Po dodaniu tego prostego kodu tworzenie tablicy na podstawie innej, już istniejącej:

```
CIntArray aTablica2 = aTablica1;
```

jest już całkowicie poprawne.

Konwersje

Trzecim i ostatnim aspektem konstruktorów, jakim się tu zajmiemy, bedzie ich wykorzystanie do konwersji typów. Temat ten jest jednak nieco szerszy niż wykorzystanie samych tylko konstruktorów, więc omówimy go sobie w całości.

Konwersje niejawne (ang. *implicit conversions*) mogą nam ułatwić programowanie - jak większość rzeczy w C++ :) W tym przypadku pozwalają na przykład uchronić się od konieczności definiowania wielu przeciążonych funkcji.
Najlepszą ilustracją będzie tu odpowiedni przykład. Akurat tak się dziwnie składa, że podręczniki programowania podają tu najczęściej jakąś klasę złożonych liczb. Nie warto naruszac tej dobrej tradycji - zatem spójrzmy na taką oto klasę liczby wymiernej:

```
class CRational
{
   private:
         // licznik i mianownik
         int m_nLicznik;
         int m_nMianownik;

   public:
         // konstruktor
         CRational(int nLicznik, int nMianownik)
               : m_nLicznik(nLicznik), m_nMianownik(nMianownik) { }

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

         // metody dostępowe
         int Licznik() const            { return m_nLicznik; }
         void Licznik(int nLicznik)     { m_nLicznik = nLicznik; }
         int Mianownik() const          { return m_nMianownik; }
         void Mianownik(int nMianownik)
               { m_nMianownik = (nMianownik != 0 ? nMianownik : 1); }
};
```

Napiszemy teraz funkcję mnożącą przez siebie dwie takie liczby (czyli dwa ułamki). Jeśli nie spaliśmy na lekcjach matematyki w szkole podstawowej, to będzie ona wyglądać chociażby tak:

```
CRational Pomnoz(const CRational& Liczba1, const CRational& Liczba2)
{
   return CRational(Liczba1.Licznik() * Liczba2.Licznik(),
                    Liczba1.Mianownik() * Liczba2.Mianownik());
}
```

Możemy teraz używać naszej funkcji na przykład w ten sposób:

```
CRational Raz(1, 2), Dwa(2, 3);
CRational Wynik = Pomnoz(Raz, Dwa);
```

Niestety, jest pewna niedogodność. Nie możemy zastosować np. takiego wywołania:

```
CRational Wynik = Pomnoz(Raz, 5);
```

Drugi argument nie może być bowiem typu int, lecz musi być obiektem typu `CRational`. To niezbyt dobrze: wiemy przecież, że `5` (i każda liczba całkowita) jest także liczbą wymierną.

My to wiemy, ale kompilator nie. W tej sekcji poznamy zatem sposoby na informowanie go o takich faktach - czyli właśnie niejawne konwersje.

Sposoby dokonywania konwersji

Sprecyzujmy, o co nam właściwie chodzi. Otóż chcemy, aby liczby całkowite (typu `int`) mogły być przez kompilator interpretowane jako obiekty naszej klasy `CRational`.

Fachowo mówimy, że chcemy zdefiniować sposób konwersji typu int na typ `CRational`.

Właśnie o takich konwersjach będziemy mówić w niniejszym paragrafie. Poznamy dwa sposoby na realizację automatycznej zamiany typów w C++.

Konstruktory konwertujące

Pierwszym z nich jest tytułowy **konstruktor konwertujący**.

Konstruktor z jednym obowiązkowym parametrem

Konstruktor konwertujący może przyjmować dokładnie **jeden parametr określonego typu** i wykonywać jego konwersję na **typ swojej klasy**.

Jest to ten mechanizm, którego aktualnie potrzebujemy. Zdefiniujmy więc konstruktor konwertujący w klasie `CRational`:

```
CRational::CRational(int nLiczba)
   : m_nLicznik(nLiczba), m_nMianownik(1)    { }
```

Od tej pory wywołanie typu:

```
CRational Wynik = Pomnoz(Raz, 5);
```

albo nawet:

```
CRational Wynik = Pomnoz(14, 5);
```

jest całkowicie poprawne. Kompilator wie bowiem, w jaki sposób zamienić „obiekt” typu `int` na obiekt typu `CRational`.

To samo osiągnać można nawet prościej. Zasada „jeden argument” dla konstruktora konwertującego działa tak samo jak „brak argumentów” dla konstruktora domyślnego. A zatem dodatkowe argumenty mogą być, lecz muszą mieć wartości domyślne.
W naszej klasie możemy więc po prostu zmodyfikować normalny konstruktor:

```
CRational(int nLicznik, int nMianownik = 1)
   : m_nLicznik(nLicznik), m_nMianownik(nMianownik)     { }
```

W ten sposób za jednym zamachem mamy normalny konstruktor, jak też konwertujący. Ba, można pójść nawet jeszcze dalej:

```
CRational(int nLicznik = 0, int nMianownik = 1)
   : m_nLicznik(nLicznik), m_nMianownik(nMianownik)     { }
```

Ten konstruktor może być wywołany bez parametrów, z jednym lub dwoma. Jest on więc jednocześnie domyślny i konwertujący. Duży efekt małym kosztem.

Konstruktor konwertujący nie musi koniecznie definiować konwersji z typu podstawowego. Może wykorzystywać dowolny typ. Popatrzmy na to:

```
class CComplex
{
   private:
         // część rzeczywista i urojona
         float m_fRe;
         float m_fIm;

   public:
```

```
        // zwykły konstruktor (który jest również domyślny
        // oraz konwertujący z float do CComplex)
        CComplex(float fRe = 0, float fIm = 0)
              : m_fRe(fRe), m_fIm(fIm)      { }

        // konstruktor konwertujący z CRational do CComplex
        CComplex(const CRational& Wymierna)
              : m_fRe(Wymierna.Licznik()
                     / (float) Wymierna.Mianownik()),
                m_fIm(0)                                  { }

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

        // metody dostępowe
        float Re() const          { return m_fRe; }
        void Re(float fRe)        { m_fRe = fRe; }
        float Im() const          { return m_fIm; }
        void Im(float fIm)        { m_fIm = fIm; }
};
```

Klasa `CComplex` posiada zdefiniowane konstruktory konwertujące zarówno z `float`, jak i `CRational`. Poza tym, że odpowiada to oczywistemu faktowi, iż liczby rzeczywiste i wymierne są także zespolone, pozwala to na napisanie takiej funkcji:

```
CComplex Dodaj(const CComplex& Liczba1, const CComplex& Liczba2)
{
   return CComplex(Liczba1.Re() + Liczba2.Re(),
                   Liczba2.Im() + Liczba2.Im());
}
```

oraz wywoływanie jej zarówno z parametrami typu `CComplex`, jaki `CRational` i `float`:

```
CComplex Wynik;
Wynik = Dodaj(CComplex(1, 5), 4);
Wynik = Dodaj(CRational(10, 3), CRational(1, 3));
Wynik = Dodaj(1, 2);
// itd.
```

Można zapytać: „Czy konstruktor konwertujący z `float` do `CComplex` jest konieczny? Przecież jest już jeden, z `float` do `CRational`, i drugi - z `CRational` do `CComplex`. Oba robią w sumie to, co trzeba!” Tak, to byłaby prawda. W sumie jednak jest to bardzo głęboko ukryte. Istotą niejawnych konwersji jest właśnie to, że są niejawne: programista nie musi się o nie martwić. Z drugiej strony oznacza to, że pewien kod jest wykonywany „za plecami” kodera. Przy jednej niedosłownej zamianie nie jest to raczej problemem, ale przy większej ich liczbie trudno byłoby zorientować się, co tak naprawdę jest zamieniane w co.
Oprócz tego jest jeszcze bardziej prozaiczny powód: gdyby pozwalać na wielokrotne konwersje, kompilator musiałby sprawdzać mnóstwo potencjalnych dróg konwersji. Znacznie wydłużyłoby to czas kompilacji.

Nie jest więc dziwne, że:

Kompilator C++ dokonuje zawsze **co najwyżej jednej** niejawnej konwersji.

Nie jest przy tym ważne, czy do konwersji stosujemy konstruktory czy też operatory konwersji, które poznamy w następnym akapicie.

Słówko `explicit`

Dowiedzieliśmy się, że **każdy jednoargumentowy konstruktor** definiuje konwersję typu swojego parametru do typu klasy konstruktora. W ten sposób możemy określać, jak kompilator ma zamienić jakiś typ (na przykład wbudowany lub inną klasę) w typ naszych obiektów.

Łatwo przeoczyć fakt, że tą drogą jednoargumentowy konstruktor (który jest w sumie konstruktorem jak każdy inny…) nabiera nowego znaczenia. Już nie tylko inicjalizuje obiekt swej klasy, ale i podaje sposób konwersji.
Dotąd mówiliśmy, że to dobrze. Nie zawsze jednak tak jest. Czasem piszemy w klasie jednoparametrowy konstruktor wcale nie po to, aby ustalić jakąkolwiek konwersję. Nierzadko bowiem tego wymaga logika naszej klasy. Spójrzmy chociażby na konstruktor z `CIntArray`:

```
CIntArray(unsigned uRozmiar)
   : m_uRozmiar(uRozmiar);
     m_pnTablica(new int [m_uRozmiar])       { }
```

Przyjmuje on parametr typu `int` - rozmiar tablicy. Niestety (tak, niestety!) jest tutaj także konstruktorem konwertującym z typu `int` na typ `CIntArray`. Z tegoż powodu zupełnie poprawne staje się bezsensowne przypisanie[108] w rodzaju:

```
CIntArray aTablica;
aTablica = 10;        // Ojć! Tworzymy 10-elementową tablicę!
```

W powyższym kodzie tworzona jest tablica o odpowiedniej liczbie elementów i przypisywana zmiennej `Tablica`. Na pewno nie możemy na to pozwolić - takie przypisanie to przecież ewidentny błąd, który powinien zostać wykryty przez kompilator.

Jednak musimy mu o tym powiedzieć i w tym celu posługujemy się słówkiem `explicit` ('jawny'):

```
explicit CIntArray(unsigned uRozmiar)
   : m_uRozmiar(uRozmiar);
     m_pnTablica(new int [m_uRozmiar])       { }
```

Gdy opatrzymy nim deklarację konstruktora jednoargumentowanego, będzie to znakiem, iż **nie chcemy**, aby wykonywał on niejawną konwersję. Po zastosowaniu tego manewru sporny kod nie będzie się już kompilował. I bardzo dobrze.

Jeżeli potrzebujesz **konstruktora jednoparametrowego**, który będzie działał **wyłacznie jako zwykły** (a nie też jako konwertujący), umieść w jego deklaracji słowo kluczowe `explicit`.

Jak wiemy konstruktor konwertujący może mieć więcej argumentów, jeśli ma też parametry opcjonalne. Do takich konstruktorów również można stosować `explicit`, jeśli jest to konieczne.

Operatory konwersji

Teraz poznamy drugi sposób konwersji typów - funkcje (operatory) konwertujące.

[108] A także podobna do niego inicjalizacja oraz każde użycie liczby `int` w miejsce tablicy `CIntArray`.

Stwarzamy sobie problem

Zostawmy wyższą matematykę liczb zespolonych w klasie CComplex i zajmijmy się klasą CRational. Jak wiemy, reprezentowane przez nią liczby wymierne są także liczbami rzeczywistymi. Byłoby zatem dobrze, abyśmy mogli przekazywać je w tych miejscach, gdzie wymagane są liczby zmiennoprzecinkowe, np.:

```
float abs(float x);
float sqrt(float x);
// itd.
```

Niestety, nie jest to możliwe. Obecnie musimy sami dzielić licznik przez mianownik, aby otrzymać liczbę typu float z typu CRational. Dlaczego jednak kompilator nie miałby tutaj pomóc? Zdefiniujmy niejawną konwersję z typu CRational do float!
W tym momencie napotkamy poważny problem. Konwersja do typu CRational była jak najbardziej możliwa poprzez konstruktor, natomiast zamiana z typu CRational na float nie może być już tak zrealizowana. Nie możemy przecież dodać konstruktora konwertującego do „klasy” float, bo jest to elementarny typ podstawowy. Zresztą, nawet jeśli nasz docelowy typ byłby klasą, to nie zawsze byłoby to możliwe. Konieczna byłaby bowiem modyfikacja definicji tej klasy, a to jest możliwe tylko dla naszych własnych klas.

Tak więc konstruktory konwertujące na niewiele nam się zdadzą. Potrzebujemy innego sposobu…

Definiowanie operatora konwersji

Tą nową metodą jest operator konwersji. Metodą w sensie dosłownym - musimy bowiem zdefiniować go jako metodę klasy CRational:

```
CRational::operator float()
{
   return m_nLicznik / static_cast<float>(m_nMianownik);
}
```

Ogólnie więc funkcja w postaci:

```
klasa::operator typ()
{
   ciało_funkcji
}
```

definiuje sposób, w jaki dokonywna jest konwersja *klasy* do podanego *typu*. Zatem:

> **Operatorów konwersji** możemy używać, aby zdefiniować niejawną **konwersję typu swojej klasy na inny, dowolny typ**.

Zyskujemy to, na czym nam zależało. Odtąd możemy swobodnie przekazywać liczby wymierne w tych miejscach, gdzie funkcje żądają liczb rzeczywistych:

```
CRational Liczba(3, 4);
float fPierwiastek = sqrt(Liczba);
```

Jest to zasługa operatorów konwersji.

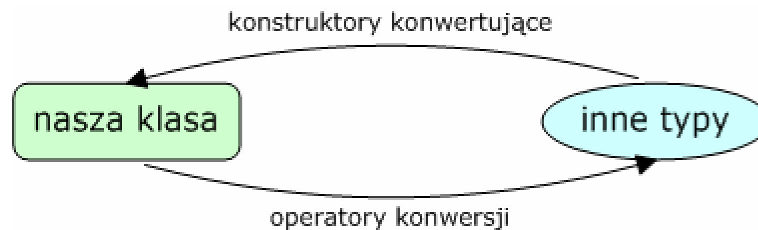
> Operatory konwersji, w przeciwieństwie do konstruktorów, są dziedziczone i mogą być metodami wirtualnymi.

Wybór odpowiedniego sposobu

Mamy więc dwa sposoby konwersji typów. Nasuwa się pytanie: który wybrać? Pytanie to jest zasadne, bowiem jeśli w konwersji dwóch typów użyjemy obu dróg (konstruktor oraz operator), to powstanie wieloznaczność. Gdy kompilator będzie zmuszony sięgnąć po konwersję, nie będzie mógł zdecydować się na żaden sposób i zaprotestuje.

Aby odpowiedzieć na to ważne pytanie, przypomnijmy, jak działają obie metody konwersji:

- konstruktor konwertujący dokonuje zamiany innego typu w obiekt naszej klasy
- operator konwersji zamienia obiekt naszej klasy w obiekt innego typu



Schemat 38. Sposoby dokonywania niejawnych konwersji w C++

Wszystko zależy więc od tego, który z typów - źródłowy, docelowy - jest klasą, do której definicji mamy dostęp:

- jeżeli jesteśmy w posiadaniu definicji klasy docelowej, to możemy zastosować konstruktor konwertujący
- jeśli mamy dostęp do klasy źródłowej, możliwe jest zastosowanie operatora konwersji

W przypadku gdy oba warunki są spełnione (tzn. chcemy wykonać konwersję z własnoręcznie napisanej klasy do innej własnej klasy), wybór sposobu jest w dużej mierze dowolny. Trzeba jednak pamiętać, że:

- konstruktory nie są dziedziczone, więc w jeśli chcemy napisac konwersję typu do klasy pochodnej, potrzebujemy osobnego konstruktora w tej klasie
- konstruktory nie są metodami wirtualnymi, w przeciwieństwie do operatorów konwersji
- argument konstruktora konwertującego musi mieć typ ściśle dopasowany do zadeklarowanego

W sumie więc wnioski z tego są takie (czytaj: przechodzimy do sedna :D):

- chcąc wykonać konwersję typu podstawowego (lub klasy bibliotecznej) do typu własnej klasy, stosujemy konstruktor konwertujący
- chcąc dokonać konwersji typu własnej klasy do typu podstawowego (lub klasy bibliotecznej), wykorzystujemy operator konwersji
- definiując konwersję między dwoma własnymi klasami możemy wybrać, kierując się innymi przesłankami, jak np. wpływem dziedziczenia na konwersje czy nawet kolejnością definicji obu klas w pliku nagłówkowym

***

Zbiorem dobrych rad odnośnie stosowania różnych typów konwersji zakończyliśmy omawianie zaawansowanych aspektów konstruktorów w C++.

Przeciążanie operatorów

W tym podrozdziale przyjrzymy się unikalnej dla C++, a jednocześnie wspaniałej technice przeciążania operatorów. To jedno z największych osiągnięć tego języka w zakresie ułatwiania programowania i uczynienia go przyjemniejszym.

Zanim jednak poznamy tę cudowność, czas na krótką dygresję :) Jak już wielokrotnie wspomniałem, C++ jest członkiem bardzo licznej dzisiaj rodziny języków obiektowych. Takie języki charakteryzuje możliwość tworzenia własnych typów danych - klas - zawierających w sobie (kapsułkujących) pewne dane (pola) oraz pewne działania (metody). Na tym polega OOP.
Żaden język programowania nie może się jednak obyć bez mniej lub bardziej rozbudowanego wachlarza typów podstawowych. W C++ mamy ich mnóstwo, z czego większość jest spadkiem po jego poprzedniku, języku C.

Z jednej strony mamy więc typy wbudowane (w C++: `int`, `float`, `unsigned`, itd.), a drugiej typy definiowane przez użytkownika (struktury, klasy, unie). W jakim stopniu są one do siebie podobne?…
Pomyślisz: „Głupie pytanie! One przecież wcale nie są do siebie podobne. Typów podstawowych używamy przeciez inaczej niż klas, i na odwrót. Nie ma mowy o jakimś większym podobieństwie - może poza tym, że dla wszystkich typów możemy deklarować zmienne i parametry funkcji… No i może jeszcze występują podobne konwersje…” Jeżeli faktycznie tak pomyślałeś, to nie będziesz zdziwiony, że twórcy wielu języków obiektowych także przyjęli taką strategię. W językach Java, Object Pascal (Delphi), Visual Basic, PHP i jeszcze wielu innych, typy definiowane przez użytkownika (klasy) są jakby wydzieloną częścią języka. Mają niewiele punktów wspólnych z typami wbudowanymi, poza tymi naprawdę niezbędnymi, które sam wyliczyłeś.
Jednak wcale nie musi tak być i C++ jest tego najlepszym przykładem. Autorzy tego języka (z Bjarne Stroustrupem na czele) dążyli bowiem do tego, aby definiowane przez programistę typy były funkcjonalnie jak najbardziej zbliżone do typów wbudowanych. Już sam fakt, że możemy tworzyć obiekty na dwa sposoby - jak normalne zmienne oraz poprzez `new` - dobrze o tym świadczy. Możliwość zdefiniowania konstruktorów kopiujących i konwersji świadczy o tym jeszcze bardziej.
Ale ukoronowaniem tych wysiłków jest obecność w C++ mechanizmu **przeciążania operatorów**.

Czy więc jest ten wspaniały mechanizm?

Przeciążanie operatorów (ang. *operator overloading*), zwane też ich **przeładowaniem**, polega na nadawaniu operatorom **nowych znaczeń** - tak, aby mogły być one **wykorzystane w stosunku do obiektów zdefiniowanych klas**.

Polega to więc na napisaniu takiego kodu, który sprawi, że wyrażenia w rodzaju:

```
a = b + c
a /= d
if (b == c) { /* ... */ }
```

będą poprawne nie tylko wtedy, gdy `a`, `b`, `c` i `d` będą zmiennymi, należącymi do typów wbudowanych. Po przeciążeniu operatorów (tutaj: `+`, `=`, `/=` i `==`) dla określonych klas będzie można pisać takie wyrażenia: zawierające operatory i obiekty naszych klas. W ten sposób zdefiniowane przez nas klasy nie będą się różniły praktycznie niczym od typów wbudowanych.

Dlaczego to jest takie cudowne?… By się o tym przekonać, przypomnijmy sobie zdefiniowaną ongiś klasę liczb wymiernych - `CRational`. Napisaliśmy sobie wtedy funkcję, która zajmowała się ich mnożeniem. Używaliśmy jej w ten sposób:

```
CRational Liczba1(1, 2),          // 1/2, czyli pół :)
          Liczba2(5, 1),          // 5
          Wynik;                  // zmienna na wynik

Wynik = Pomnoz(Liczba1, Liczba2;
```

Nie wyglądało to zachwycająco, szczególnie jeśli uświadomimy sobie, że dla typów wbudowanych ostatnia linijka mogłaby prezentować się tak:

```
Wynik = Liczba1 * Liczba2;
```

Nie dość, że krócej, to jeszcze ładniej… Czemu my tak nie możemy?!

Ależ tak, właśnie możemy! Przeciążanie operatorów pozwala nam na to! Znając tę technikę, możemy zdefiniowac nowe znaczenie dla operator mnożenia, czyli `*`. Nauczymy go pracy z liczbami wymiernymi - obiektami naszej klasy `CRational` - i od tego momentu pokazane wyżej mnożenie będzie dla nich **poprawne**! Co więcej, będzie działało zgodnie z naszymi oczekiwaniami: tak, jak funkcja `Pomnoz()`. Czyż to nie piękne?

Na takie wspaniałości pozwala nam przeciążanie operatorów. Na co więc jeszcze czekamy - zobaczmy, jak to się robi!… Hola, nie tak prędko! Jak sama nazwa wskazuje, technika ta dotyczy operatorów, a dokładniej: wyposażania ich w nowe znaczenia. Zanim się za to zabierzemy, warto byłoby znać przedmiot naszych manipulacji. Powinniśmy zatem przyjrzeć się operatorom w C++: ich rodzajom, wbudowanej funkcjonalności oraz innym właściwościom.
I to właśnie zrobimy najpierw. Tylko nie narzekaj :P

Cechy operatorów

Obok słów kluczowych i typów, operatory są podstawowymi elementami każdego języka programowania wysokiego poziomu. Przypomnijmy sobie, czym jest operator.

Operator to jeden lub kilka znaków (zazwyczaj niebędących literami), które mają specjalne znaczenie w języku programowania.

Dotychczas używaliśmy bardzo wielu operatorów - niemal wszystkich, jakie występują w C++ - ale dotąd nie zajęliśmy się nimi całościowo. Poznałeś wprawdzie takie pojęcia jak operatory unarne, binarne, priorytety, jednak teraz będzie zasadne ich powtórzenie.

Zbierzmy więc tutaj wszystkie cechy operatorów występujących w C++.

Liczba argumentów

Operator sam w sobie nie może wykonywać żadnej czynności (to różni go od funkcji), gdyż potrzebuje jakichś „parametrów”. W tym przypadku mówimy zwykle o argumentach operatora - **operandach**.

Operatory dzielą się z grubsza na dwie duże grupy, jeżeli chodzi o liczbę swoich argumentów. Są to operatory jedno- i dwuargumentowe. W C++ mamy jeszcze operator warunkowy `?:`, uznawany za ternarny (trójargumentowy), ale jest on wyjątkiem, którym nie należy zaprzątać sobie głowy.

Operatory jednoargumentowe

Te operatory fachowo nazywa się **unarnymi** (ang. *unary operators*). Stanowią one całkiem liczną rodzinę, która charakteryzuje się jednym: każdy jej członek wymaga do działania **jednego argumentu**. Stąd nazwa tego rodzaju operatorów.

Najbardziej znanym operatorem unarnym (nawet dla tych, którzy nie mają pojęcia o programowaniu!) jest zwykły minus. Formalnie nazywa się go operatorem negacji albo zmiany znaku, a działa on w ten sposób, że zmienia jakaś liczbę na liczbę do niej przeciwną:

```
int nA = 5;
int nB = -nA;         // nB ma wartość -5 (a nA nadal 5)
```

Podobnie działają operatory `!` i `~`, z tym że operują one (odpowiednio): na wyrażeniach logicznych i na ciągach bitów. Istnieją też operatory jednoargumentowane o zupełnie innej funkcjonalności; wszystkie je przypomnimy sobie w następnej sekcji.

Operatory dwuargumentowe

Jak sama nazwa wskazuje, te operatory przyjmują po **dwa argumenty**. Nazywamy je **binarnymi** (ang. *binary operators*). Nie ma to nic wspólnego z binarną reprezentacją danych, lecz po prostu z ilością operandów.

Typowymi operatorami dwuargumentowymi są operatory arytmetyczne, czyli popularne „znaki działań”:

```
int nA = 8, nB = -2, nC;
nC = nA + nB;              // 6
nC = nA - nB;              // 10
nC = nA * nB;              // -16
nC = nA / nB;              // -4
```

Mamy też operatory logiczne oraz bitowe, Warto wspomnieć (o czym będziemy jeszcze bardzo szeroko mówić), że przypisanie (`=`) to także operator dwuargumentowy, dość specyficzny zresztą.

Priorytet

Operatory mogą występować w złożonych wyrażeniach, a ich argumenty mogą pokrywać się. Oto prosty przykład:

```
int nA = nB * 4 + 18 / nC - nD % 3;
```

Zapewne wiesz, że w takiej sytuacji kompilator kieruje się **priorytetami operatorów** (ang. *operators' precedence*), aby rozstrzygnąć problem. Owe priorytety to nic innego, jak swoista „kolejność działań”. Różni się ona od tej znanej z matematyki tylko tym, że w C++ mamy także inne operatory niż arytmetyczne.
Dla znaków `+`, `-`, `*`, `/`, `%` priorytety są aczkolwiek dokładnie takie, jakich nauczyliśmy się w szkole. Wyrażenia zawierające te operatory możemy więc pisać bez pomocy nawiasów. Jeżeli jednak są one skomplikowane, albo używamy w nich także innych rodzajów operatorów, wówczas konieczne należy pomagać sobie nawiasami. Lepiej przecież postawić po kilka znaków więcej niż co chwila sięgać do stosownej tabelki pierwszeństwa.

Łączność

Gdy w wyrażeniu pojawi się obok siebie kilka operatorów tego samego rodzaju, mają one oczywiście ten sam priorytet. Trzeba jednak nadal rozstrzygnąć, w jakiej kolejności działania będą wykonywane.

Tutaj pomaga **łączność operatorów** (ang. *operators' associativity*). Określa ona, od której strony będą obliczane wyrażenia (lub ich fragmenty) z sąsiedztwem operatorów o tych samych priorytetach. Mamy dwa rodzaje łączności:

- **łączność lewostronna** (ang. *left-to-right associativity*), która rozpoczyna obliczenia od lewej strony i wykorzystuje cząstkowe wyniki jako lewostronne argumenty dla kolejnych operatorów
- **łączność prawostronna** (ang. *right-to-left associativity*) - tutaj obliczenia są wykonywane, poczynając od prawej strony. Częściowe wyniki są następnie używane jako prawostronne argumenty kolejnych operatorów

Najlepiej zilustrować to na przykładzie. Jeżeli mamy takie oto wyrażenie:

```
nA + nB + nC + nD + nE + nF + nG + nH
```

to oczywiście priorytety wszystkich operatorów są te same. Zaczyna dominować łączność, która w przypadku operatorów arytmetycznych (oraz im podobnych, jak bitowe, logiczne i relacyjne) jest lewostronna. To naturalne, po przecież takie obliczenia również przeprowadzalibyśmy „od lewej do prawej".
Kompilator będzie więc obliczał powyższe wyrażenie w ten sposób:

```
((((((nA + nB) + nC) + nD) + nE) + nF) + nG) + nH
```

Zauważmy, że akurat w przypadku plusa łączność nie ma znaczenia, bo dodawanie jest przecież przemienne. Gdyby jednak chodziło o odejmowanie czy dzielenie, wówczas byłoby to bardzo ważne.

Łączność prawostronna dotyczy na przykład operatora przypisania:

```
nA = nB = nC = nD = nE = nF
```

Innymi słowy, powyższe wyrażenie zostanie potraktowane tak:

```
nA = (nB = (nC = (nD = (nE = nF))))
```

Oznacza to, że kompilator wykona najpierw skrajnie prawe przypisanie, a zwróconą przez to wyrażenie wartość (równą wartości przypisywanej) wykorzysta w kolejnym przypisaniu, i tak dalej. W sumie więc wszystkie zmienne będą potem równe zmiennej `nF`.

Operatory w C++

Język C++ posiada całe multum różnych operatorów. Pod tym względem jest chyba rekordzistą wśród wszystkich języków programowania. Świadczy to zarówno o jego wielkich możliwościach, jak i sporej elastyczności.
Co ciekawe, dotąd praktycznie nie ma jednoznacznej definicji operatora w tym języku, a w wielu źródłach można znaleźć nieco różniące się między sobą zestawy operatorów. Są to jednak głównie niuanse, których rozstrzyganie dla przeciętnego programisty nie jest wcale istotne.

W tej sekcji powtórzymy sobie i uzupełnimy wiadomości na temat wszystkich operatorów C++ - przynajmniej tych, co do których nie ma wątpliwości, że faktycznie są operatorami. Podzielimy je sobie na kilka kategorii.

Operatory arytmetyczne

Już na samym początku zetknęliśmy się z operatorami arytmetycznymi. Nic dziwnego, to przecież najprostszy i „najbardziej naturalny" rodzaj operatorów. Znają go wszyscy absolwenci przedszkola.

Unarne operatory arytmetyczne

Mamy dwa podstawowe jednoargumentowe operatory arytmetyczne:

- operator **zachowania znaku**, czyli `+`. On praktycznie nie robi nic - zachowuje znak liczby, przy której stoi. Obecny w C++ chyba tylko dla zgodności z zasadami matematyki
- operator **zmiany znaku**, czyli `–`. Zamienia liczbę na przeciwną, zupełnie tak jak w arytmetyce

Trochę przykładów:

```
int nA = 7
int nB = +nA;          // 7
int nB = -NA;          // -7
```

Myślę, że jest to na tyle oczywiste, że nie wymaga dalszych komentarzy.

Inkrementacja i dekrementacja

Specyficzne dla C++ są operatory inkrementacji i dekrementacji. W odróżnieniu od większości operatorów, **modyfikują one swój argument**. Dokładniej mówiąc, dodają one (inkrementacja) lub odejmują (dekrementacja) jedynkę do (od) swego operandu.

Operatorem inkrementacji jest `++`, zaś dekrementacji `––`. Oto przykład:

```
int nX = 9;
++nX;            // teraz nX == 10
--nX;            // teraz znowu nX == 9
```

Powyższy kod można też zapisać jako:

```
nX++;
nY++;
```

Jeżeli ignorujemy wartość zwracaną przez te operatory, to użycie którejkolwiek wersji (zwanej, jak wiesz, prein/dekrementacją oraz postin/dekrementacją) nie sprawa różnicy - przynajmniej dla typów podstawowych.
Gdy natomiast zapisujemy gdzieś zwracaną wartość, to powinniśmy pamiętać o różnicy między znaczeniem operatorów w obu miejscach (na początku i na końcu zmiennej). Mówiliśmy już o tym, ale przypomnę jeszcze raz:

Prein/dekrementacja zwraca **wartość już zwiększoną (zmniejszoną) o 1**.
Postin/dekrementacja zwraca **oryginalną wartość**.

Wariant postfiksowy jest generalnie bardziej kosztowny, ponieważ wymaga przygotowania tymczasowego obiektu, w którym zostanie zachowana pierwotna wartość w celu jej późniejszego zwrotu. Dla typów podstawowych to kwestia kilku bajtów, ale dla klas zdefiniowanych przez użytkownika (które mogą przeciążać oba operatory - czym się rzecz jasna zajmiemy za momencik) może to być spora różnica.

Binarne operatory arytmetyczne

Przypomnijmy, że w C++ mamy pięć takich operatorów, zwanych popularnie „znakami działań”:

- operator **dodawania** - plus (+). Dodaje dwie liczby do siebie
- operator **odejmowania** - minus (-). Zwraca wynik odejmowania drugiego argumentu od pierwszego
- operator **mnożenia** - gwiazdka (*). Mnoży oba argmenty
- operator **dzielenia** - slash (/). W zależności od typu swoich operandów może albo wykonywać dzielenie całkowitoliczbowe (gdy oba argumenty są liczbami całkowitymi), albo zmiennoprzecinkowe
- operator **reszty z dzielenia**, czyli %. Zwraca resztę z dzielenia podanych liczb

Znowu popatrzmy na kilka przykładów:

```
int nA = 9, nB = 4, nX;
float fX;

nX = nA + nB;                          // 13
nX = nA - nB;                          // 5
nX = nA * nB;                          // 36
nX = nA / nB;                          // 2
fX = nA / static_cast<float>(nB);      // 2.25f
nX = nA % nB;                          // 1
```

Ponownie nie ma tu nic nieoczekiwanego.

Operatory bitowe

Przedstawione wyżej operatory arytmetyczne działają na liczbach na zasadach, do jakich przyzwyczaiła nas matematyka. Nie ma w tym przypadku znaczenia, że operacje przeprowadzane są na komputerze. Nie ma też znaczenia wewnętrzna reprezentacja liczb.

Jak wiemy, komputery przechowują dane w postaci ciągów zer i jedynek, zwanych **bitami**. Pojedyncze bity mogą przechowywać tylko elementarną informację - 0 (bit ustawiony) lub 1 (bit nieustawiony). Aby przedstawiać bardziej złożone dane - choćby liczby - należy bity łączyć ze sobą. Powstają w ten sposób **wektory bitowe**, **ciągi bitów** (ang. *bitsets*) lub **słowa** (ang. *words*). Są po prostu sekwencje zer i jedynek.
Do operacji na wektorach bitów C++ posiada sześć operatorów. Obecnie nie są one tak często używane jak na przykład w czasach C, ale nadal są bardzo przydatne. Omówię je tu pokrótce.

O wiele obszerniejsze omówienie tych operatorów, wraz z zastosowaniami, znajdziesz w Dodatku C, *Manipulacje bitami*.

Operacje logiczno-bitowe

Cztery operatory: ~, &, | i ^ wykonują na bitach operacje zbliżone do logicznych, gdzie bit ustawiony (1) odgrywa rolę wyrażenia prawdziwego, zaś nieustawiony (0) - fałszywego. Oto te operatory:

- **negacja bitowa** (operator ~) zmienia w całym ciągu (zwykle liczbie) wszystkie bity na przeciwne. Ustawione zmieniają się na nieustawione i odwrotnie
- **koniunkcja bitowa** (operator &) porównuje ze sobą odpowiadające bity dwóch słów: tam, gdzie napotka na dwie jedynki, wypisuje do wyniku także jedynkę; w przeciwnym wypadku zero

- **alternatywa bitowa** (operator |) również działa na dwóch słowach. Porównując ich kolejne bity, zwraca w bicie wyniku zero, jeżeli stwierdzi w operandach dwa nieustawione bity oraz jedynkę w przeciwnym wypadku
- **bitowa różnica symetryczna** (operator ^) porównuje bity słów i zwraca 1, jeżeli są różne i 0, gdy są sobie równe

Operator ~ jest jednoargumentowy (unarny), zaś pozostałe dwa są binarne - i wcale nie dlatego, że pracują w systemie dwójkowym :)

Przesunięcie bitowe

Mamy też dwa operatory **przesunięcia bitowego** (ang. *bitwise shift*). Jest to:
- **przesunięcie w lewo** (operator <<). Przesuwa on bity w lewą stronę słowa o podaną liczbę miejsc
- **przesunięcie w prawo** (operator >>) działa analogicznie, tylko że przesuwa bity w prawą stronę słowa

Z obu operatorów korzystamy podobnie, tj. w ten sposób:

```
słowo << ile_miejsc
słowo >> ile_miejsc
```

Oto kilka przykładów - dla uproszczenia z liczbami zapisanymi binarnie (niestety, w C++ nie można tego zrobić):

```
00010010 << 3        // 10010000
1111000 >> 4         // 00001111
00111100 << 5        // 10000000
```

Jak widać, bity które „wyjeżdżają” w wyniku przesunięcia poza granicę słowa są tracone. Pustki są natomiast wypełniane zerami.

Operatory strumieniowe

Czytając ten akapit na pewno pomyślałeś: „Jakie operatory bitowe?! Przecież to są ‘strzałki’, których używamy razem ze strumieniami wejścia i wyjścia!” Tak, to również prawda - ale to tylko jedna jej strona.
Faktem jest, że << i >> to przede wszystkim operatory przesunięcia bitowego. Nie przeszkadza to jednak, aby miały one także inne znaczenie - co więcej, mają je one tylko w odniesieniu do strumieni. W sumie więc pełnią one w C++ aż dwie funkcje.

Czy domyślasz się, dlaczego?… Ależ tak, właśnie tak - operatory te zostały **przeciążone** przez twórców Biblioteki Standardowej C++. Posiadają one dodatkową funkcjonalność, która pozwala na ich używanie razem z obiektami `cout` i `cin`[109]. W odniesieniu do samych liczb nadal jednak są one operatorami przesunięcia bitowego.
Nieco więcej informacji o tych operatorach otrzymasz przy okazji omawiania strumieni STL. Tam też nauczysz się przeciążać je dla swoich własnych klas - tak, aby ich obiekty można było zapisywać do strumieni i odczytywać z nich w identyczny sposób, jak typy wbudowane.

Operatory porównania

Bardzo ważnym rodzaje operatorów są operatory porównania, czyli znaki: < (mniejszy), > (większy), <= (mniejszy lub równy), >= (większy lub równy), == (równy) oraz != (różny).

[109] Również `clog`, `cerr` oraz wszystkimi innymi obiektami, wywodzącymi się od klas `istream` i `ostream` oraz ich pochodnych. Po więcej informacji odsyłam do rozdziału o strumieniach Biblioteki Standardowej.

O tych operatorach wiemy w zasadzie wszystko, bo używamy ich nieustannie. O tym, jak działają, powiedzieliśmy sobie zresztą bardzo wcześnie.
Zwrócę jeszcze tylko uwagę, aby nie mylić operatora równości (==) z operatorem przypisania (=). Omyłkowe użycie tego drugiego w miejsce pierwszego nie zostanie bowiem oprotestowane przez kompilator (co najwyżej wygeneruje on ostrzeżenie). Dlaczego tak jest - wyjaśnię przy okazji operatrów przypisania.

Operatory logiczne

Te operatory służą do łączenia wyrażeń logicznych (`true` lub `false`) w złożone warunki. Takie warunki możemy potem wykorzystać z instrukcjach `if` oraz pętlach, co zresztą niejednokrotnie robiliśmy.

W C++ mamy trzy operatory logiczne, będące odpowiednikami pewnych operatorów bitowych. Różnica polega jednak na tym, że operatory logiczne działają na **wartościach liczb** (lub wyrażeń logicznych: fałszywe oznacza 0, zaś prawdziwe - 1) zaś bitowe - na wartościach bitów.
Oto te trzy operatory:
- **negacja** (zaprzeczenie, operator `!`) powoduje zamianę prawdy (1) na fałsz (0)
- **koniunkcja** (iloczyn logiczny, operator `&&`) dwóch wyrażeń zwraca prawdę tylko wówczas, gdy oba jej argumenty są prawdziwe
- **alternatywa** (suma logiczna, operator `||`) jest prawdziwa, gdy choć jeden z jej argumentów jest prawdziwy (różny od zera)

Warto zapamiętać, że w wyrażeniach zawierających operatory `&&` i `||` wykonywanych jest tylko tyle obliczeń, ile jest koniecznych do zdeterminowania wartości warunkowych. Przykładowo, w poniższym kodzie:

```
int nZmienna;
std::cin >> nZmienna;
if (nZmienna >= 1 && nZmienna <= 10)   { /* ... */ }
```

jeżeli stwierdzona zostanie falszywość pierwszej części koniunkcji (`nZmienna >= 1`), to druga nie będzie już sprawdzana i cały warunek uznany zostanie za fałszywy. Podobnie dzieje się przy alternatywie, której pierwszy argument jest prawdziwy - wówczas całe wyrażenie również reprezentuje prawdę.

Argumenty operatorów logicznych są więc zawsze obliczane od lewej do prawej.

Wśród operatorów nie ma różnicy symetrycznej, zwanej alternatywą wykluczającą (ang. *XOR* - *eXclusive OR*). Można ją jednak łatwo uzyskać, wykorzystując tożsamość:

$$a \oplus b \Leftrightarrow \neg(a \Leftrightarrow b)$$

co w przełożeniu na C++ wygląda tak:

```
if (!(a == b)) { /* ... */ }      // a i b to wyrażenia logiczne
```

Operatory przypisania

Kolejną grupę stanowią operatory przypisania. C++ ma ich kilkanaście, choć wiemy, że tak naprawdę tylko jeden jest do szczęścia potrzebny. Pozostałe stworzono dla wygody programisty, jak zresztą wiele mechanizmów w C++.

Popatrzmy więc na operatory przypisania.

Zwykły operator przypisania

Operator przypisania (ang. *assignment operator*) ma postać pojedynczego znaku 'równa się' (=). Doskonale też wiemy, jak się go używa:

```
int nX;
nX = 7;
```

Po wykonaniu tego kodu, zmienna nX będzie miał wartość 7.

L-wartość i r-wartość

Zauważmy, że odwrotne przypisanie:

```
7 = nX;          // źle!
```

jest niepoprawne. Nie możemy nic przypisać do siódemki, bo ona nie zajmuje żadnej komórki w pamięci - w przeciwieństwie do zmiennej, jak np. nX.

Zarówno 7, jak i nX, są jednak poprawnymi wyrażeniami języka C++. Widzimy aczkolwiek, że różnią się pod względem „współpracy z przypisaniem”. nX może być celem przypisania, zaś 7 - nie.
Mówimy, że nX jest **l-wartością**, zaś 7 - **r-wartością** lub **p-wartością**.

L-wartość (ang. *l-value*) jest wyrażeniem **mogącym wystąpić po lewej stronie operatora przypisania** - stąd ich nazwa.

R-wartość (ang. *r-value*), po polsku zwana **p-wartością**, **może wystąpić tylko po prawej stronie operatora przypisania**.

Zauważmy, że nic nie stoi na przeszkodzie, aby nX pojawiło się po prawej stronie operatora przypisania:

```
int nY;
nY = nX;
```

Jest tak, ponieważ:

Każda l-wartość jest **jednocześnie r-wartością** (p-wartością) - lecz nie odwrotnie!

Domyślasz się pewnie, że w C++ **każde wyrażenie jest r-wartością**, ponieważ reprezentuje jakieś dane. L-wartościami są natomiast te wyrażenia, które:
- odpowiadają komórkom pamięci operacyjnej
- nie są oznaczone jako stałe (const)

Najbardziej typowymi rodzajami l-wartości są więc:
- zmienne wszystkich typów niezadeklarowane jako const
- wskaźniki do powyższych zmiennych, wobec których stosujemy operator dereferencji, czyli gwiazdkę (*)
- niestałe referencje do tychże zmiennych
- elementy niestałych tablic
- niestałe pola klas, struktur i unii, które podpadają pod jeden z powyższych punktów i nie występują w ciele stałych metod[110]

[110] Wyjątkiem są pola oznaczone słowem mutable, które zawsze mogą być modyfkowane.

R-wartości to oczywiście te, jak i wszystkie inne wyrażenia.

Rezultat przypisania

Wyrażeniem jest także samo przypisanie, gdyż samo w sobie reprezentuje pewną wartość:

```
std::cout << (nX = 5);
```

Ta linijka kodu wyprodukuje rezultat:

```
5
```

co pozwala nam ugólnić, iż:

Rezultatem przypisania jest **przypisywana wartość**.

Ten fakt powoduje, że w C++ możliwe są, niespotykane w innych językach, wielokrotne przypisania:

```
nA = nB = nC = nD = nE;
```

Ponieważ operator(y) przypisania mają łączność prawostronną, więc ten wiersz zostanie obliczony jako:

```
nA = (nB = (nC = (nD = nE)));
```

Innymi słowy, `nE` zostanie przypisane do `nD`. Następnie rezultat tego przypisania (czyli `nE`, bo to było przypisywane) zostanie przypisany do `nC`. To także wyprodukuje rezultat - i to ten sam, `nE` - który zostanie przypisany `nB`. To przypisanie również zwróci ten sam wynik, który zostanie wreszcie umieszczony w `nA`. W ten więc sposób wszystkie zmienne będą miały ostatecznie tą samą wartość, co `nE`.
Tą techniką możemy wykonać tyle przypisań naraz, ile tylko sobie życzymy.

Uwaga na przypisanie w miejscu równości

Niestety, traktowanie przypisania jako wyrażenia ma też swoją ciemną stronę. Bardzo łatwo jest umieścić je omyłkowo w warunku `if` lub pętli zamiast operatora `==`, np.:

```
while (nA = 5)
    std::cin >> nA;
```

Jeżeli nasz kompilator jest lekkoduchem, to może nas nie ostrzec przed niebezpieczeństwem tej pętli. A zagrożenie jest spore, bo jest nic innego, jak **pętla nieskończona**. Podobno komputer Cray wykonałby ją w dwie sekundy - jeżeli chcesz, możesz sprawdzić, ile zajmie to twojej maszynie ;D Lepiej jednak zaradzić powstałemu problemowi.
Jak on jednak powstaje?… Otóż sprawa jest dość prosta, a wszystkiemu winien warunek pętli. Jest to przecież przypisanie - przypisanie wartości `5` do zmiennej `nA`. Jako test logiczny wykorzystywana jest **wartość tego przypisania** - czyli piątka. Pięć jest oczywiście różne od zera, zatem zostanie uznane za warunek prawdziwy. Tak oto pętla się zapętla i zaciska na szyi biednego programisty.

Możemy się kłocić, że to wina C++, który nie dość, że uznaje liczby całkowite (jak `5`) za wyrażenia logiczne, to jeszcze pozwala na wykonywanie przypisania w warunkach `if`'ów i pętli. Możliwości te zostały jednak dopuszczone z uzasadnionych względów (praca ze wskaźnikami), więc wcale niewykluczone, że kiedyś je docenimy. Niezależnie od tego, czy

będziemy świadomie wykonywać przypisania w podobnych sytuacjach, musimy pamiętać, że:

Należy zwracać baczną uwagę na każde przypisanie występujące w warunku instrukcji `if` lub pętli. Może to być bowiem niedoszłe porównanie.

Zaleca się, aby **opatrywać stosownym komentarzem** każde **zamierzone użycie** przypisania w tych newralgicznych miejscach. Dzięki temu unikniemy nieporozumień z kompilatorem, innymi programistami i… samym sobą!

Złożone operatory przypisania

Dla wygody programisty C++ posiada jeszcze dziesięć innych operatorów przypisania. Są one po prostu krótszym zapisem często stosowanych instrukcji. Ich postać i „rozwinięcia” przedstawia to oto tabelka:

przypisanie	„rozwinięcie”
`a += b`	`a = a + b`
`a -= b`	`a = a - b`
`a *= b`	`a = a * b`
`a /= b`	`a = a / b`
`a %= b`	`a = a % b`
`a &= b`	`a = a & b`
`a |= b`	`a = a | b`
`a ^= b`	`a = a ^ b`
`a <<= b`	`a = a << b`
`a >>= b`	`a = a >> b`

Tabela 17. Złożone operatory przypisania w C++

‘Rozwinięcie’ wziąłem w cudzysłów, ponieważ nie jest tak, że jakiś mechanizm w rodzaju makrodefinicji zamienia te skrócone wyrażenia do ich „pełnych” form. O nie, one są kompilowane w tej postaci. Ma to taki skutek, że wyrażenie po lewej stronie operatora jest obliczane **jeden raz**. W wersji „rozwiniętej” byłoby natomiast obliczane dwa razy.

Podobna zasada obowiązuje też w operatorach pre/postin/dekrementacji.

Jest to też realizacja bardziej fundamentalnej reguły, która mówi, że składniki każdego wyrażenia są obliczane tylko raz.

Operatory wskaźnikowe

Wskaźniki były ongiś kluczową cechą języka C, a i w C++ nie straciły wiele ze swojego znaczenia. Do ich obsługi mamy w naszym ulubionym języku trzy operatory.

Pobranie adresu

Jednoargumentowy operator `&` służy do **pobrania adresu** obiektu, przy którym stoi. Oto przykład:

```
int nZmienna;
int* pnWskaznik = &nZmienna;
```

Argument tego operatora musi być **l-wartością**. To raczej oczywiste, bo przecież musi ona rezydować w jakimś miejscu pamięci. Inaczej niemożliwe byłoby pobranie adresu tego miejsca. Typowo operandem dla `&` jest zmienna lub funkcja.

Dostęp do pamięci poprzez wskaźnik

Do obszaru pamięci, do którego posiadamy wskaźnik, możemy odnieść się na kilka sposobów. Dokładnie: na dwa.

Dereferencja

Najprostszym i najczęściej stosowanym sposobem jest dereferencja:

```
int nZmienna;
int* pnWskaznik = &nZmienna;
*pnWskaznik = 42;
```

Odpowiada za nią jednoargumentowy operator `*`, zwany operatorem **dereferencji** lub adresowania pośredniego. Pozwala on na dostęp do miejsca w pamięci, któremu odpowiada wskaźnik. Operator ten wykorzystuje ponadto typ wskaźnika, co gwarantuje, że odczytana zostanie właściwa ilość bajtów. Dla `int*` będzie to `sizeof(int)`, zatem `*pnWskaznik` reprezetuje u nas liczbę całkowitą.

To, czy `*`*`wskaźnik`* jest l-wartością, czy nie, zależy od stałości wskaźnika. Jeżeli jest to stały wskaźnik (`const` *`typ`*`*`), wówczas nie możemy modyfikować pokazywanej przezeń pamięci. Mamy więc do czynienia z r-wartością. W pozostałych przypadkach mamy l-wartość.

Indeksowanie

Jeżeli wskaźnik pokazuje na tablicę, to możemy dostać się do jej kolejnych elementów za pomocą **operatora indeksowania** (ang. *subscript operator*) - nawiasów kwadratowych `[]`.

Oto zupełnie banalny przykład:

```
std::string aBajka[3];

aBajka[0] = "Dawno, dawno temu, ...";
aBajka[1] = "w odleglej galaktyce...";
aBajka[2] = "zylo sobie siedmiu kransoludkow...";
```

Jeżeli zapytasz „A gdzie tu wskaźnik?”, to najpierw udam, że tego nie słyszałem i pozwolę ci na chwilę zastanowienia. A jeśli nadal będziesz się upierał, że żadnego wskaźnika tu nie ma, to będę zmuszony nałożyć na ciebie wyrok powtórnego przeczytania rozdziału o wskaźnikach. Chyba tego nie chcesz? ;-)

Wskaźnikiem jest tu oczywiście `aBajka` - jaka nazwa tablicy wskazuje na jej pierwszy element. W zasadzie więc można dokonać jego dereferencji i dostać się do tego elementu:

```
*aBajka = "Dawno, dawno temu, ...";
```

Przesuwając wskaźnik przy pomocy dodawania można też dostać się do pozostałej części tablicy:

```
*(aBajka + 1) = "w odleglej galaktyce...";
*(aBajka + 2) = "zylo sobie siedmiu kransoludkow...";
```

Taki zapis jest jednak dość kłopotliwy w interpretacji - choć koniecznie trzeba go znać (przydaje się przy iteratorach STL). C++ ma wygodniejszy sposób dostepu do elementów tablicy o danym indeksie - jest to właśnie operator indeksowania.

Na koniec muszę jeszcze przypomnieć, że wyrażenie:

```
tablica[i]
```

odpowiada (i-1)-emu elementowi *tablicy*. A to dlatego, że:

W C++ elementy tablic (oraz łańcuchów znaków) liczymy **od zera**.

Skoro już tak się powtarzam, to przypomnę jeszcze, że:

W *n*-elementowej tablicy nie istnieje element o indeksie *n*. Próba odwołania się do niego spowoduje błąd ochrony pamięci.

Zasada ta nie dotyczy aczkolwiek łańcuchów znaków, gdzie *n*-ty element to zawsze znak o kodzie 0 ('\0'). Jest to zaszłość zakonserwowana w czasach C, która przetrwała do dziś.

Operatory pamięci

Mamy w C++ kilka operatorów zajmujących się pamięcią. Jedne służą do jej alokacji, drugie do zwalniania, a jeszcze inne do pobierania rozmiaru typów i obiektów.

Alokacja pamięci

Alokacja pamięci to przydzielenie jej określonej ilości dla programu, by ten mógł ją wykorzystać do własnych celów. Pozwala to dynamicznie tworzyć zmienne i tablice.

new

new jest przeznaczony do dynamicznego tworzenia zmiennych. Obiekty stworzone przy pomocy tego operatora są tworzone na stercie, a nie na stosie, zatem nie znikają po opuszczeniu swego zakresu. Tak naprawdę to w ogóle nie stosuje się do nich pojęcie zasięgu.

Tworzenie obiektów poprzez new jest banalnie proste:

```
float pfZmienna = new float;
```

Oczywiście nie ma zbyt wielkiego sensu tworzenie zmiennych typów podstawowych czy nawet prostych klas. Jeżeli jednak mamy do czynienia z dużymi obiektami, które muszą istnieć przez dłuższy czas i być dostępne w wielu miejscach programu, wtedy musimy tworzyć je dynamicznie poprzez new.

W przypadku kreowania obiektów klas, new dba o prawidłowe wywołanie konstrukturów, więc nie trzeba się tym martwić.

new[]

Wersję operatora new, która służy do alokowania tablic, nazywam new[], aby w ten sposób podkreślić jej związek z delete[].

new[] potrafi alokować tablice dynamiczne po podanym rozmiarze. Aby użyć tej możliwości po nazwie docelowego typu określamy wymiary pożądanej tablicy, np.:

```
float** matMacierz4x4 = new float [4][4];
```

W wyniku dostajemy odpowiedni wskaźnik lub ewentualnie wskaźnik do wskaźnika (do wskaźnika do wskaźnika itd. - zależnie od liczby wymiarów), który możemy zachować w zmiennej określonego typu.
Do powstałej tablicy odwołujemy się tak samo, jak do tablic statycznych:

```
for (unsigned i = 0; i < 4; ++i)
   for (unsigned j = 0; j < 4; ++j)
         matMacierz4x4[i][j] = (i == j ? 1.0f : 0.0f);
```

Dynamiczna tablica istnieje jednak na stercie, więc tak samo jak wszystkie obiekty tworzone w czasie działania programu nie podlega regułom zasięgu.

Zwalnianie pamięci

Pamięć zaalokowana przy pomocy `new` i `new[]` musi zostać zwolniona przy pomocy odpowiadających im operatorów `delete` i `delete[]`. Wiesz doskonale, że w przeciwnym razie dojdzie do groźnego błędu wycieku pamięci.

delete

Za pomocą `delete` niszczymy pamięć zaalokowaną przez `new`. Dla operatora tego należy podać wskaźnik na tenże blok pamięci, np.:

```
delete pfZmienna;
```

`delete` zapewnia wywołanie destruktora klasy, jeżeli takowy jest konieczny. Destruktor taki może być wiązany wcześnie (jak zwykła metoda) lub późno (jak metoda wirtualna) - ten drugi sposób jest zalecany, jeżeli chcemy korzystać z dobrodziejstw polimorfizmu.

delete[]

Analogicznie, `delete[]` służy do zwalniania dynamicznych tablic. Nie musimy podawać rozmiaru takiej tablicy, gdy ją niszczymy - wystarczy tylko wskaźnik:

```
delete[] matMacierz4x4;
```

Koniecznie pamiętajmy, aby nie mylić obu postaci operatora `delete[]` - w szczególności nie można stosować `delete` do zwalniania pamięci przydzielonej przez `new[]`.

Operator sizeof

`sizeof` pozwala na pobranie rozmiaru obiektu lub typu:

```
int nZmienna;
if (sizeof(nZmienna) != sizeof(int))
   std::cout << "Chyba mamy zepsuty kompilator :D";
```

Jest to operator **czasu kompilacji**, więc nie może korzystać z informacji uzyskanych w czasie działania programu. W szczególności, nie może pobrać rozmiaru dynamicznej tablicy - nawet mimo takich prob:

```
int* pnTablica = new int [5];

std::cout << sizeof(pnTablica);        // to samo co sizeof(int*)
std::cout << sizeof(*pnTablica);       // to samo co sizeof(int)
```

Taki rozmiar trzeba po prostu zapisać gdzieś po alokacji tablicy.

`sizeof` zwraca wartość należącą do predefiniownego typu `size_t`. Zwykle jest to liczba bez znaku lub bardzo duża liczba ze znakiem.

Ciekawostka: operator `__alignof`

W Visual C++ istnieje jeszcze podobny do `sizeof` operator `__alignof`. Używamy go w ten sam sposób, podając mu zmienną lub typ. W wyniku zwraca on tzw. **wyrównanie** (ang. *alignment*) danego typu danych. Jest to liczba, która określa sposób organizacji pamięci dla danego typu danych. Przykładowo, jeżeli wyrównywanie wynosi 8, to znaczy to, iż obiekty tego typu są wyrównane w pamięci do wielokrotności ośmiu bajtów (ich adresy są wielokrotnocią ośmiu).
Wyrównanie sprawia rzecz jasna, że dane zajmują w pamięci nieco więcej miejsca niż faktycznie mogłyby. Zyskujemy jednak szybciej, ponieważ porcje pamięci wyrównane do całkowitych potęg dwójki (a takie jest zawsze wyrównanie) są przetwarzane szybciej.

Wyrównanie można kontrolować poprzez `__declspec(align(liczba))`. Np. poniższa struktura:

```
__declspec(align(16)) struct FOO { int nA, nB; };
```

będzie tworzyć zmienne zajmujące w pamięci fragmenty po 16 bajtów, choć jej faktyczny rozmiar jest dwa razy mniejszy[111].
Polecając wyrównywanie do 1 bajta określimy praktyczny jego brak:

```
#define PACKED __declspec(align(1))
```

Typy danych opatrzone taką deklaracją będą więc ciasno upakowane w pamięci. Może to dać pewną jej oszczędność, ale zazwyczaj spadek prędkości dostępu do danych nie jest tego wart.

Operatory typów

Istnieją języki programowania, które całkiem dobrze radzą sobie bez posiadania ściśle zarysowanych typów danych. C++ do nich nie należy: w nim typ jest sprawą bardzo ważną, a do pracy z nim oddelegowano kilka specjalnych operatorów.

Operatory rzutowania

Rzutowanie jest zmianą typu wartości, czyli jej konwersją. Mamy parę operatorów, które zajmują się tym zadaniem i robią to w różny sposób.

Wśród nich są tak zwane cztery „nowe” operatory, o składni:

```
określenie_cast<typ_docelowy>(wyrażenie)
```

To właśnie one są zalecane do używania we wszystkich sytuacjach, wymagających rzutowania. C++ zachowuje aczkolwiek także starą formę rzutowania, znaną z C.

`static_cast`

Ten operator może być wykorzystywany do większości konwersji, jakie zdarza się przeprowadzać w C++. Nie oznacza to jednak, że pozwala on na wszystko:

> Poprawność rzutowania `static_cast` jest sprawdzana **w czasie kompilacji** programu.

`static_cast` można używać np. do:
- konwersji między typami numerycznymi
- rzutowania liczby na typ wyliczeniowy (`enum`)

[111] Jeżeli `int` ma 4 bajty długości, a tak jest na każdej platformie 32-bitowej.

- rzutowania wskaźników do klas związanych relacją dziedziczenia

Jeżeli chodzi o ostatnie zastosowanie, to należy pamiętać, że tylko konwersja wskaźnika na obiekt klasy pochodnej do wskaźnika na obiekt klasy bazowej jest zawsze bezpieczna. W odwrotnym przypadku trzeba być pewnym co do wykonalności rzutowania, aby nie narobić sobie kłopotów. Taką pewność można uzyskać na przykład za pomocą sposobu z metodami wirtualnymi, który zaprezentowałem w rozdziale 1.7, lub poprzez operator `typeid`.
Inną możliwością jest też użycie operatora `dynamic_cast`.

dynamic_cast

Przy pomocy `dynamic_cast` można rzutować wskaźniki i referencje do obiektów w dół hierarchii dziedziczenia. Oznacza to, że można zamienić odwołanie do obiektu klasy bazowej na odwołanie do obiektu klasy pochodnej. Wygląda to np. tak:

```
class CFoo                    { /* ... */ };
class CBar : public CFoo      { /* ... */ };

void Funkcja(CFoo* pFoo)
{
   CBar* pBar = dynamic_cast<CBar*>(pFoo);

   // ...
}
```

Taka zamiana nie zawsze jest możliwa, bo przecież dany wskaźnik (referencja) niekoniecznie musi pokazywać na obiekt żądanej klasy pochodnej. Operacja jest jednak bezpieczna, ponieważ:

Poprawność rzutowania `dynamic_cast` jest sprawdzana **w czasie działania** programu.

Wiemy doskonale, w jaki sposób poznać rezultat tego sprawdzania. `dynamic_cast` zwraca po prostu `NULL` (wskaźnik pusty, zero), jeżeli rzutowanie nie mogło zostać wykonane. Należy to zawsze skontrolować:

```
if (!pBar)
{
   // OK - pBar faktycznie pokazuje na obiekt klasy CBar
}
```

Dla skrócenia zapisu można wykorzystać wartość zwracaną operatora przypisania:

```
if (pBar = dynamic_cast<CBar*>(pFoo))
{
   // rzutowanie powiodło się
}
```

Znak `=` jest tu oczywiście zamierzony. Warunek będzie miał bowiem wartość równą rezultatowi rzutowania, zatem będzie prawdziwy tylko wtedy, gdy się ono powiedzie. Zwrócony wskaźnik będzie wtedy różny od zera.

reinterpret_cast

`reinterpret_cast` może służyć do dowolnych konwersji między wskaźnikami, a także do rzutowania wskaźników na typy liczbowe i odwrotnie. Wachlarz możliwości jest więc szeroki, niestety:

Poprawność rzutowania `reinterpret_cast` **nie jest sprawdzana**.

Łatwo więc może dojść do niebezpiecznych konwersji. Ten operator powinien być używany tylko jako ostatnia deska ratunku - jeżeli inne zawiodą, a my jesteśmy przekonani o względnym bezpieczeństwie planowanej zamiany. Wykorzystanie tego operatora generalnie jednak powinno być bardzo rzadkie.

`reintepret_cast` możemy potencjalnie użyć np. do uzyskania dostępu do pojedynczych bitów w zmiennej o większej ich ilości:

```
unsigned __int32 u32Zmienna;     // liczba 32-bitowa
unsigned __int8* pu8Bajty;       // wskaźnik na liczby 8-bitowe (bajty)

// zamieniamy wskaźnik do 4 bajtowej zmiennej na wskaźnik do
// 4-elementowej tablicy bajtów
pu8Bajty = reinterpret_cast<unsigned __int8*>(&u32Zmienna);

// wyświetlamy kolejne bajty zmiennej u32Zmienna
for (unsigned i = 0; i < 4; ++i)
   std::cout << "Bajt nr " << i << ": " << pu8Bajty[i] << std::endl;
```

Widać więc, że najlepiej sprawdza się w operacjach niskopoziomowych. Tutaj możnaby oczywiście użyć przesunięcia bitowego, ale tablica wygląda z pewnością przejrzyściej.

const_cast

Ostatni z „nowych” operatorów rzutowania ma dość ograniczone zastosowanie:

`const_cast` służy do **usuwania przydomków** `const` i `volatile` z opatrzonych nimi wskaźników do zmiennych.

Obecność tego operatora służy chyba tylko temu, aby możliwe było całkowite zastąpienie sposobów rzutowania znanych z C. Jego praktyczne użycie należy do sporadycznych sytuacji.

Rzutowanie w stylu C

C++ zachowuje „stare” sposoby rzutowania typów. Jednym z nich jest rzutowanie nazywane, całkiem adekwatnie, rzutowaniem w stylu C (ang. *C-style cast*):

```
(typ) wyrażenie
```

Ta składnia konwersji jest nadal często używana, gdyż jest po prostu krótsza. Należy jednak wiedzieć, że nie odróżnia ona różnych sposobów rzutowania i w zależności od *typu* i *wyrażenia* może się zachowywać jak `static_cast`, `reinterpret_cast` lub `const_cast`.

Rzutowanie funkcyjne

Inną składnię ma rzutowanie funkcyjne (ang. *function-style cast*):

```
typ(wyrażenie)
```

Przypomina ona wywołanie funkcji, choć oczywiście żadna funkcja nie jest tu wywoływana. Ten rodzaj rzutowania działa tak samo jak rzutowanie w stylu C, aczkolwiek nie można w nim stosować co niektórych nazw typów. Nie można na przykład wykonać:

```
int*(&fZmienna)
```

i to z dość prozaicznego powodu. Po prostu gwiazdka i nawias otwierający występujące obok siebie zostaną potraktowane jako błąd składniowy. W tej sytuacji można sobie ewetualnie pomóc odpowiednim `typedef`'em.

Operator `typeid`

`typeid` służy pobrania informacji o typie podanego wyrażenia podczas działania programu. Jest to tzw. RTTI, czyli **informacja o typie czasu wykonania** (ang. *Run-Time Type Information*).
Przygotowanie do wykorzystania tego operatora objemuje włączenie RTTI (co dla Visual C++ opisałem w rozdziałe 1.7) oraz dołączenie standardowego nagłówka *typeinfo*:

```
#include <typeinfo>
```

Potem możemy już stosować `typeid` np. tak:

```
class CFoo                 { /* ... */ };
class CBar : public CFoo   { /* ... */ };

int nZmienna;
CFoo* pFoo = new CBar;
std::cout << typeid(nZmienna).name();        // int
std::cout << typeid(pFoo).name();            // class CFoo *
std::cout << typeid(*pFoo).name();           // class CBar
```

Jak widać, operator ten jest leniwy i jeśli tylko może, będzie korzystać z informacji dostępnych w czasie kompilacji programu. Ażeby więc poznać np. typ polimorficznego obiektu, na który pokazujemy wskaźnikiem, trzeba użyć derefrencji…

Operatory dostępu do składowych

Pięć kolejnych operatorów służy do wybierania składników klas, struktur, unii, itd. Przy ich pomocy można więc dostać się do zagnieżdżonych składowych. Nie zawsze jest to jednak możliwe - wszystko zależy od ich widoczności, czyli od tego, jakimi specyfikatorami dostępu są one opatrzone (`private`, `protected`, `public`).
O tychże specyfikatorach mówiliśmy już bardzo wiele, więc teraz przypomnijmy sobie tylko same operatory wyłuskania.

Wyłuskanie z obiektu

Mając zmienną obiektową, do jej składników odwołujemy się poprzez operator kropki (`.`), np. tak:

```
struct FOO     { int x; };

FOO Foo;
Foo.x = 10;
```

W podobny działa operator `.*`, który służy aczkolwiek do wyłowienia składnika poprzez wskaźnik do niego:

```
int FOO::*p2mnSkladnik = &FOO::x;
Foo.*p2mnSkladnik = 42;
```

Wskaźniki na składowe są przedmiotem następnego podrozdziału.

Wyłuskanie ze wskaźnika

Gdy mamy wskaźnik na obiekt, wówczas zamiast kropki używamy innego operatora wyłuskania - strzałki (->):

```
FOO* pFoo = new FOO;
pFoo->x = 16;
```

Tutaj także mamy odpowiednik, służący do wybierania składowych za pośrednictwem wskaźnika na nie:

```
pFoo->*p2mnSkladnik += 80;
```

W powyższej linijce mamy dwa wskaźniki, stojące po obydwu stronach operatora `->*`. O pierwszym rodzaju powiedzieliśmy sobie na samym początku programowania obiektowego - to po prostu zwyczajny wskaźnik na obiekt. Drugi to natomiast wskaźnik do składowej klasy - o tym typie wskaźników pisze więcej następny podrozdział.

Operator zasięgu

Ten operator, nazywany też **operatorem rozwikłania zakresu** (ang. *scope resolution operator*) służy w C++ do rozróżniania nazw, które rezydują w różnych zakresach. Znamy dwa podstawowe zastosowania tego operatora:
- dostęp do przesłoniętych zmiennych globalnych
- dostęp do składowych klasy

Ogólnie, operatora tego używamy, aby dostać się do identyfikatora zagnieżdżoneego wewnątrz nazwanych zakresów:

```
zakres_poziom1::[zakres_poziom2::[zakres_poziom3::[...]]]nazwa
```

Nazwy zakresów odpowiadają m.in. strukturom, klasom i uniom. Przykładowo, `FOO` z poprzedniego akapitu było nazwą zakresu - oprócz tego, rzecz jasna, także nazwą struktury. Przy pomocy operatora `::` można odnieść się do jej zawartości.

Zakresy można też tworzyć poprzez tzw. **przestrzenie nazw** (ang. *namespaces*). Jest to bardzo dobre narzędzie, służące organizacji kodu i zapobiegające konfliktom oznaczeń. Opisuje je rozdział *Sztuka organizacji kodu*.
Do tej pory cały czas korzystaliśmy z pewnej szczególnej przestrzeni nazw - `std`. Pamiętasz doskonale, że przy niej także używaliśmy operatora zakresu.

Pozostałe operatory

Ostatnie trzy operatory trudno zakwalifikować do jakiejś konkretnej grupy, więc zebrałem je tutaj.

Nawiasy okrągłe

Nawiasy `()` to dość oczywisty operator. W C++ służy on głównie do:
- grupowania wyrażeń w celu ich obliczania w pierwszej kolejności
- deklarowania funkcji i wskaźników na nie
- wywoływania funkcji
- rzutowania

Brak nawiasów może być przyczyną błędnego (innego niż przewidywane) obliczania wyrażeń, a także nieprawidłowej interpretacji niektórych deklaracji (np. funkcji i wskaźników na nie). Obfite stawianie nawiasów jest szczególnie ważne w makrodefinicjach.

Z kolei nadmiar nawiasów jeszcze nikomu nie zaszkodził :)

Operator warunkowy

Operator `?:` jest nazywamy ternarnym, czyli trójargumentowym. Jako jedyny bierze bowiem trzy dane:

```
warunek ? wynik_dla_prawdy : wynik_dla_fałszu
```

Umiejętne użycie tego operatora skraca kod i pozwala uniknąć niepotrzebnych instrukcji `if`. Co ciekawe, może on być także użyty w deklaracjach, np. pól w klasach. Wtedy jednak wszystkie jego operandy muszą być stałymi.

Przecinek

Przecinek (ang. *comma*) to operator o najniższym priorytecie. Oprócz tego, że oddziela on argumenty funkcji, może też występować samodzielnie, np.:

```
(nX + 17, 26, rand() % 5, nY)
```

W takim wyrażeniu operandy są obliczane od lewej do prawej, natomiast wynikiem jest wartość ostatniego wyrażenia. Tutaj więc będzie to `nY`.

Przecinek przydaje się, gdy chcemy wykonać pewną dodatkową czynność w trakcie wyliczania jakiejś wartości. Przykładowo, spójrzmy na taką pętlę odczytującą znaki:

```
char chZnak;
while (chZnak = ReadChar(), chZnak != ' ')
{
   // zrób coś ze znakiem, który nie jest spacją
}
```

`ReadChar()` jest funkcją, która pobiera następny znak (np. z pliku). Sama pętla ma zaś wykonywać się aż do napotkania spacji. Zanim jednak można sprawdzić, czy dany znak jest spacją, trzeba go odczytać. Robimy to w warunku pętli, posługując się przecinkiem. Bez niego trzebaby najprawdopodobniej zmienić całą pętlę na `do`, co spowodowałoby konieczność powtórzenia kodu wywołującego `ReadChar()`. Inne wyjście to użycie pętli nieskończonej. C++ pozwala jednak osiągnąć ten sam efekt na kilka sposobów, spośród których wybieramy ten najbardziej nam pasujący.

Nowe znaczenia dla operatorów

Przypomnieliśmy sobie wszystkie operatory C++ i ich domyślne znaczenia. Nam to jednak nie wystarcza - chcemy przecież zdefiniować dla nich całkiem nowe funkcje. Zobaczmy zatem, jak możemy to uczynić.

Funkcje operatorowe

Pomyślmy: co właściwie robi kompilator, gdy natrafi w wyrażeniu na jakiś operator? Czy tylko sobie znanymi sposobami oblicza on docelową wartość, czy może jednak jest w tym jakaś zasada?…

Otóż tak. Działanie operatora definiuje pewna funkcja, zwana **funkcją operatorową** (ang. *operator function*). Istnieje wiele takich funkcji, które są wbudowane w kompilator i działają na typach podstawowych. Dodawanie, odejmowanie i inne predefiniowane działania na liczbach są dostępne bez żadnych starań z naszej strony.
Kiedy natomiast chcemy przeciążyć jakiś operatory, to oznacza to konieczność napisania własnej funkcji dla nich. Zwyczajnie, trzeba podać jej argumenty oraz wartość zwracaną i

wypełnić kodem. Nie ma w tym żadnej „magii”. Za chwilę zresztą przekonasz się, jak to działa.

Kilka uwag wstępnych

Zobaczmy więc, jak można zdefiniować dodatkowe znaczenia dla operatorów w C++.

Ogólna składnia funkcji operatorowej

Przeciążenie operatora oznacza napisanie dla niego funkcji, odpowiedzialnej za jego nowe działanie. Oto najbardziej ogólna składnia takiej funkcji:

```
zwracany_typ operator symbol([parametry])
{
   treść_funkcji
}
```

Zamiast nazwy mamy tu słowo kluczowe `operator`, za którym należy podać symbol przeciążanego operatora (można go oddzielić od spacją, lecz nie jest to wymagane). Jeżeli więc chcemy np. zdefiniowiać nowe znaczenie dla plusa (+), to piszemy funkcję `operator+()`.

Jak każda funkcja, także i ta przyjmuje pewne parametry. Ich liczba zależy ściśle od tego, jaki operator chcemy przeładować. Jeśli jest to operator binarny, to siłą rzeczy konieczne będą dwa parametry; dla jednoargumentowych operatorów wystarczy jeden parametr.
Ale uwaga - `parametry` podane w nawiasie niekoniecznie są jedynymi, które funkcja otrzymuje. Pamiętasz zapewne, że metody klas mają ukryty parametr - obiekt, na rzecz którego metoda została wywołana, dostępny poprzez wskaźnik `this`. Otóż ten parametr jest **brany pod uwagę** w tym przypadku. Pamiętaj więc, że:

Funkcja operatorowa przyjmuje tyle argumentów, ile ma przeciążany przy jej pomocy operator. Do tych argumentów **zalicza się wskaźnik `this`**, jeżeli jest to metoda klasy.

Od tej zasady istnieje tylko jeden wyjątek (a w zasadzie dwa). Stanowią go operatory postinkrementacji i postdekrementacji: wprowadzono do nich dodatkowy parametr typu `int`, który należy zignorować. Dzięki temu możliwe jest odróżnienie tych operatorów od wariantów prefiksowych.

Operatory, które możemy przeciążać

Możemy przeciążać bardzo wiele operatorów - zarówno takich, dla których natychmiast znajdziemy praktyczne zastosowanie, jak i tych, których przeciążanie wydawałoby się dziwaczne. Oto kompletna lista przeciążalnych operatorów:

```
+    -    *    /    %    &    |    ^     <<     <<
~    &&   ||   !    ==   !=   <    <=    >      >=
+=   -=   *=   /=   %=   &=   |=   ^=    <<=    >>=
++   --   =    ->   ->*  ()   []   new   delete  ,
```

Tabela 18. Przeciążalne operatory C++

Przeładowywać możemy **te i tylko te** operatory. W większości książek i kursów za chwilę nastąpiłaby podobna (acz znacznie krótsza) lista operatorów, których przeciążać nie można. Z doświadczenia wiem jednak, że rodzi to niewyobrażalną ilośc nieporozumień, spowodowaną nieprecyzyjnym określeniem, co jest operatorem, a co nie. Dlatego też nie podaję żadnej takiej tabelki - zapamiętaj po prostu, że przeciążać można **wyłącznie te** operatory, które wymieniłem wyżej.

Muszę jednak podać kilka wyjaśnień odnośnie tej tabelki:
- operatory: +, –, *, & można przeciążać zarówno w wersji jedno-, jak i dwuargumentowej
- operatory inkrementacji (++) i dekrementacji (––) przeciążamy oddzielnie dla wersji prefiksowej i postfiksowej
- przeciążenie `new` i `delete` powoduje także zdefiniowanie ich działania dla wersji tablicowych (`new[]` i `delete[]`)
- operatory () i [] to nawiasy: okrągłe (grupowanie wyrażeń) i kwadratowe (indeksowanie, wybór elementów tablicy)
- operatory –> i –>* mają predefiniowane działanie dla wskaźników na obiekty - jego nie możemy zmienić. Możemy natomiast zdefiniowiać ich działanie dla samych obiektów

Czego nie możemy zmienić

Przeciążając operatory możemy zdefiniować dla nich dodatkowe znaczenie. Nie możemy jednak:
- tworzyć własnych operatorów, jak np. @, ?, === czy \
- zmienić liczby argumentów, na których pracują przeciążane operatory. Przykładowo, nie stworzymy dwuargumentowego operatora ! czy jednoargumentowego ||
- zmodyfikować priorytetu operatora
- zmienić łączności przeładowanego operatora

Dla każdego typu C++ automatycznie generuje też pięć niezbędnych operatorów, których nie musimy przeciążać, aby działały poprawnie, Są to:
- zwykły operator przypisania (=). Dokonuje on dosłownego kopiowania obiektu („pole po polu”)
- operator pobrania adresu (jednoargumentowy &). Zwraca on adres obiektu w pamięci
- `new` dokonuje alokacji pamięci dla obiektu
- `delete` niszczy i usuwa obiekt z pamięci
- przecinek (,) - jego znaczenie jest takie same, jak dla typów wbudowanych

Możliwe jest aczkolwiek przeciążenie tych pięciu symboli, aby działały inaczej dla naszych klas. Nie można jednak unieważnić ich domyślnej funkcjonalności, jaką dostarcza kompilator **dla każdego typu**. Mówiąc potocznie, nie można ich „rozdefiniować”.

Pozostałe sprawy

Warto jeszcze powiedzieć o pewnych „naturalnych” sprawach:
- przynajmniej jeden argument przeciążanego operatora musi być innego typu niż wbudowane. To naturalne: operatory przeciążamy na rzecz własnych typów (klas), bo działania na typach podstawowych są wyłączną domeną kompilatora. Nie wtrącamy się w nie
- funkcja operatorowa nie może posiadać parametrów domyślnych
- przeciążenia nie kumulują się, tzn. jeżeli na przykład przeciążymy operatory + oraz =, nie będzie to oznaczało automatycznego zdefiniowania operatora +=. Każde nowe znaczenie dla operatora musimy podać sami

Definiowanie przeciążonych wersji operatorów

Operator możemy przeciążyć na kilka sposobów, w zależności od tego, gdzie umieścimy funkcję operatorową. Może być ona bowiem zarówno składnikiem (metodą) klasy, na rzecz której działa, jak i funkcją globalną.

Na te dwa przypadki popatrzymy sobie, definiując operator mnożenia (dwuargumentowy `*`) dla klasy `CRational`, znanej z poprzednich podrozdziałów. Chcemy sprawić, aby jej obiekty można było mnożyć przez inne liczby wymierne, np. tak:

```
CRational JednaPiata(1, 5), TrzyCzwarte(3, 4);
CRational Wynik = JednaPiata * TrzyCzwarte;
```

To będzie spore udogodnienie, więc zobaczmy, jak mozna to zrobić.

Operator jako funkcja składowa klasy

Wpierw spróbujmy zdefiniować `operator*()` jako funkcję składową klasy. Wiemy, że nasz operator jest dwuargumentowy; wiemy także, że każda metoda klasy przyjmuje jeden ukryty parametr - wskaźnik `this`. Wynika stąd, że funkcja operatorowa będzie u nas miał tylko jeden „prawdziwy" parametr i wyglądała na przykład tak:

```
CRational CRational::operator*(const CRational& Liczba) const
{
   return CRational(m_nLicznik * Liczba.m_nLicznik,
                    m_nMianownik * Liczba.m_nMianownik);
}
```

To wystarczy - po tym zabiegu możemy bez problemu mnożyć przez siebie zarówno dwa obiekty klasy `CRational`:

```
CRational DwieTrzecie(2, 3), TrzySiodme(3, 7);
CRational Wynik = DwieTrzecie * TrzySiodme;
```

jak i jeden obiekt przez liczbę całkowitą:

```
CRational Polowa(1, 2);
CRational Calosc = Polowa * 2;
```

Jak to działa?… Najlepiej prześledzić funkcjonowanie operatora, jeżeli wyrażenia zawierające go:

```
DwieTrzecie * TrzySiodme
Polowa * 2
```

zapiszemy z **jawnie wywołaną** funkcją operatorową:

```
DwieTrzecie.operator*(TrzySiodme)
Polowa.operator*(2)
```

Widać wyraźnie, że pierwszy argument operatora jest przekazywany jako wskaźnik `this`. Drugi jest natomiast normalnym parametrem funkcji `operator*()`.
A jakim sposobem zyskaliśmy od razu możliwość mnożenia także przez liczby całkowite? Myślę, że to nietrudne. Zadziałała tu po prostu niejawna konwersja, zrealizowana przy pomocy konstruktora klasy `CRational`. Drugie wyrażenie jest więc w rzeczywistości wywołaniem:

```
Polowa.operator*(CRational(2))
```

Mimochodem uzyskaliśmy zatem dodatkową funkcjonalność. A wszystko za pomocą jednej funkcji operatorowej (no i jednego konstruktora).

Problem przemienności

Nasz entuzjazm szybko może jednak osłabnąć. jeżeli zechcemy wypróbować przemienność tak zdefiniowanego mnożenia. Nie będzie przeszkód dla dwóch liczb wymiernych:

```
CRational Wynik = TrzySiodme * DwieTrzecie;
```

albo dla pary całkowita-wymierna kompilator zaprotestuje:

```
CRational Calosc = 2 * Polowa;          // błąd!
```

Dlaczego tak się dzieje? Ponowny rzut oka na jawne wywołanie `operator*()` pomoże rozwikłać problem:

```
TrzySiodme.operator*(DwieTrzecie)       // OK
2.operator*(Polowa)                     // ???
```

Wyraźnie widać przyczynę. Dla dwójki nie można wywołać funkcji `operator*()`, bo taka funkcja nie istnieje dla typu `int` - on przecież nie jest nawet klasą. Nic więc dziwnego, że użycie operatora zdefiniowanego jako metoda nie powiedzie się.
„Zaraz - a co z niejawną konwersją? Dlaczego ona nie zadziałała?" Faktycznie, możnaby przypuszczać, że konstruktor konwertujący może zamienić `2` na obiekt klasy `CRational` i uczynić wyrażenie poprawnym:

```
CRational(2).operator*(Polowa)          // OK
```

To jest nieprawda. Powodem jest to, iż:

Niejawne konwersje **nie działają** przy **wyłuskiwaniu składników** obiektu.

Kompilator nie rozwinie więc problematycznego wyrażenia do powyższej postaci i zgłosi błąd.

Operator jako zwykła funkcja globalna

Wynika z tego prosty wniosek: Houston, mamy problem :) Nie rozwiążemy go na pewno, definiując `operator*()` jako funkcję składową klasy. Trzebaby bowiem dostać się do definicji klasy `int` i dodać do niej odpowiednią metodę. Szkoda tylko, że nie mamy dostępu do tej definicji, co zresztą nie zaskakuje, bo `int` nie jest przecież żadną klasą. Gdyby jednak załoga Apollo 13 załamywała się po napotkaniu tak prostych problemów, nie wróciłaby na Ziemię cała i zdrowa. Nasza sytuacja nie jest aż tak dramatyczna, chociaż „częściowo przemienny" operator mnożenia też nie jest szczytem komfortu. Trzeba coś na to poradzić.

Rozwiązanie oczywiście istnieje: należy uczynić `operator*()` funkcją globalną:

```
CRational operator*(const CRational& Liczba1, const CRational& Liczba2)
{
   return CRational(Liczba1.Licznik() * Liczba2.Licznik(),
                    Liczba1.Mianownik() * Liczba2.Mianownik());
}
```

Zmieni to bardzo wiele. Odtąd dwa rozważane wyrażenia będą rozwijane do postaci:

```
operator*(TrzySiodme, DwieTrzecie)      // OK
operator*(2, Polowa)                    // też OK!
```

W tej formie oba argumenty operatora są normalnymi parametrami funkcji `operator*()`. Ma więc ona teraz dwa wyraźne parametry, wobec których może zajść niejawna konwersja. W tym przypadku `2` faktycznie będzie więc interpretowane jako `CRational(2)`, zatem mnożenie powiedzie się bez przeszkód.
To spostrzeżenie można uogólnić:

Globalna funkcja operatorowa pozwala kompilatorowi na dokonywanie **niejawnych konwersji** wobec **wszystkich argumentów** operatora.

Jest to prosty sposób na definiowanie przemiennych działań na obiektach różnych typów, między którymi istnieją określenia konwersji.

Operator jako zaprzyjaźniona funkcja globalna

Porównajmy jeszcze treść obu wariantów funkcji `operator*()`: jako metody klasy `CRational` i jako funkcji globalnej. Widzimy, że w pierwszym przypadku operowała ona bezpośrednio na prywatnych polach `m_nLicznik` i `m_nMianownik`. Jako funkcja globalna musiała z kolei posiłkować się metodami dostępowymi - `Licznik()` i `Mianownik()`.
Nie powinno cię to dziwić. `operator*()` jako zwykła funkcja globalna jest właśnie - zwykłą funkcją globalną, zatem nie ma żadnych specjalnych uprawnień w stosunku do klasy `CRational`. Jest tak nawet pomimo faktu, że definiuje dlań operację mnożenia.

Żadne specjalne uprawnienia nie są potrzebne, bo funkcja doskonale radzi sobie bez nich. Czasem jednak operator potrzebuje dostępu do niepublicznych składowych klasy, których nie uzyska za pomocą publicznego interfejsu. W takiej sytuacji konieczne staje się uczynienie funkcji operatorowej **zaprzyjaźnioną**.
Podkreślmy jeszcze raz:

Globalna funkcja operatorowa nie musi być zaprzyjaźniona z klasą, na rzecz której definiuje znaczenie operatora.

Ten fakt pozwala na przeciążanie operatorów także dla nieswoich klas. Jak bardzo może to być przydatne, zobaczymy przy okazji omawiania strumieni STL z Biblioteki Standardowej.

Sposoby przeciążania operatorów

Po generalnym zapoznaniu się z przeciążaniem operatorów, czas na konkretne przykłady. Dowiedzmy się więc, jak przeciążać poszczególne typy operatorów.

Najczęściej stosowane przeciążenia

Najpierw poznamy takie rodzaje przeciążonych operatorów, które stosuje się najczęściej. Pomocą będzie nam tu głównie służyć klasa `CVector2D`, którą jakiś czas temu pokazałem:

```
class CVector2D
{
   private:
         float m_fX, m_fY;

   public:
         explicit CVector2D(float fX = 0.0f, float fY = 0.0f)
               { m_fX = fX; m_fY = fY; }
};
```

Nie jest to przypadek. Operatory przeciążamy bowiem najcześciej dla tego typu klas, zwanych narzędziowymi. Wektory, macierze i inne przydatne „obiekty matematyczne" są właśnie idealnymi kandydatami na klasy z przeładowanymi operatorami.

Pokazane tu przeciążenia nie będą jednak tylko sztuką dla samej sztuki. Wspomniane obiekty będą nam bowiem niezbędne z programowaniu grafiki przy użyciu DirectX. A że przy okazji ilustrują tę ciekawą technikę programistyczną, jaką jest przeciążanie operatorów, tym lepiej dla nas :)

Spójrzmy zatem, jakie ciekawe operatory możemy przedefiniować na potrzeby tego typu klas.

Typowe operatory jednoargumentowe

Operatory unarne, jak sama nazwa wskazuje, przyjmują jeden argument. Chcąc dokonać ich przeciążenia, mamy do wyboru:

- zdefiniowanie odpowiedniej metody w klasie, na rzecz której dokonujemy redefinicji:

```
klasa klasa::operator symbol() const;
```

- napisanie globalnej funkcji operatorowej:

```
klasa operator symbol(const klasa&);
```

Zauważyłeś zapewne, że w obu wzorach podaję parametry oraz typ zwracanej wartości[112]. Przestrzeganie tego schematu nie jest jednak wymogiem języka, lecz raczej powszechnie przyjętej konwencji dotyczącej przeciążania operatorów. Mówi ona, że:

Działanie operatorów wobec typów zdefiniowanych przez programistę powinno w miarę możliwości pokrywać się z ich funkcjonalnością dla typów wbudowanych.

Co to znaczy?… Otóż większość operatorów jednoargumentowych (poza in/dekrementacją) nie modyfikuje w żaden sposób przekazanych im obiektów. Przykładowo, operator jednoargumentowego minusa - zastosowany wobec liczby zwraca po prostu liczbę przeciwną.
Chcąc zachować tę konwencję, należy umieścić w odpowiednich miejscach deklaracje stałości `const`. Naturalnie nie trzeba tego bezwarunkowo robić - pamiętajmy jednak, że przestrzeganie szeroko przyjętych (i rozsądnych!) zwyczajów jest zawsze w interesie programisty. Dotyczy to zarówno piszącego, jak i czytającego i konserwującego kod.

No, ale dość tych tyrad. Pora na zastosowanie zdobytej wiedzy w praktyce. Zastanówmy się, jakie operatory możemy logicznie przeciążyć dla naszej klasy `CVector2D`. Nie jest ich wiele - w zasadzie tylko plus (`+`) oraz minus (`-`). Pierwszy nie powinien w ogóle zmieniać obiektu wektora i zwrócić go w nienaruszonym stanie, zaś drugi musi oddać wektor o przeciwnym zwrocie.
Sądzę, że bez problemu napisałbyś takie funkcje. Są one przecież niezwykle proste:

```
class CVector2D
{
   // (pomijamy szczegóły)

   public:
         // (tu też)
```

[112] Nie dotyczy to operatorów inkrementacji i dekrementacji, których omówienie znajduje się dalej.

```
            CVector2D operator+() const
                  { return CVector2D(+m_fX, +m_fY); }
            CVector2D operator-() const
                  { return CVector2D(-m_fY, -m_fY); }
   };
```

Co do drugiego operatora, to chyba nie ma żadnych wątpliwości. Natomiast przeładowywanie plusa może wydawać się wręcz śmieszne. To jednak całkowicie uzasadniona praktyka: jeśli operator ten działa dla typów wbudowanych, to powinien także funkcjononować dla naszego wektora. Aczkolwiek treść metody `operator+()` to faktycznie przykład-analogia do `operator-()`: rozsądniej byłoby po prostu zwrócić `*this` (czyli kopię wektora) niż tworzyć nowy obiekt.

Obie metody umieszczamy bezpośrednio w definicji klasy, bo są one na tyle krótkie, żeby zasługiwać na atrybut *inline*.

Inkrementacja i dekrementacja

To, co przed chwilą powiedziałem o operatorach jednoargumentowych, nie stosuje się do operatorów inkrementacji (++) i dekrementacji (--). Ściśle mówiąc, nie stosuje się w całości. Mamy tu bowiem dwie odmienne kwestie.

Pierwszą z nich jest to, iż oba te operatory nie są już tak „grzeczne” i nie pozostawiają swojego argumentu w stanie nienaruszonym. Potrzebny jest im więc dostęp do obiektu, który zezwalałby na jego modyfikację. Trudno oczekiwać, aby wszystkie funkcje miały do tego prawo, zatem `operator++()` i `operator--()` powinny być co najmniej zaprzyjaźnione z klasą. A najlepiej, żeby były po prostu jej metodami:

```
   klasa klasa::operator++();       // lub operator--()
```

Druga sprawa jest nieco innej natury. Wiemy bowiem, że inkrementacja i dekrementacja występuje w dwóch wersjach: przedrostkowej i przyrostkowej. Z zaprezentowanej wyżej składni wynika jednak, że możemy przeładować tylko jedną z nich. Czy tak?...

Bynajmniej. Powyższa forma jest prototypem funkcji operatorowej dla **preinkrementacji**, czyli dla przedrostkowego wariantu operatora. Nie znaczy to jednak, że wersji postfiksowej nie można przeciążyć. Przeciwnie, jest to jak najbardziej możliwe w ten oto sposób:

```
   klasa klasa::operator++(int);    // lub operator--(int)
```

Nie jest on zbyt elegancki i ma wszelkie znamiona „triku”, ale na coś trzeba było się zdecydować... Dodatkowy argument typu `int` jest tu niczym innym, jak **środkiem do rozróżnienia** obu typów in/dekrementacji. Nie pełni on poza tym żadnej roli, a już na pewno nie trzeba go podawać podczas stosowania postfiksowego operatora `++` (`--`). Jest on nadal jednoargumentowy, a dodatkowy parametr jest tylko mało satysfakcjonującym wyjściem z sytuacji.

> W początakach C++ tego nie było, gdyż po prostu niemożliwe było przeciążanie przyrostkowych operatorów inkrementacji (dekrementacji). Później jednak stało się to dopuszczalne - opuścimy już jednak zasłonę milczenia na sposób, w jaki to zrealizowano.

Tak samo jak w przypadku wszystkich operatorów zaleca się, aby zachowanie obu wersji `++` i `--` było spójne z typami podstawowymi. Jeśli więc przeciążamy prefiksowy `operator++()` lub (i) `operator--()`, to w wyniku powinien on zwracać obiekt już po dokonaniu założonej operacji zwiększenia o 1.
Dla spokoju sumienia lepiej też przeciążyć obie wersje tych operatorów. Nie jest to uciążliwe, bo możemy korzystać z już napisanych funkcji. Oto przykład dla `CVector2D`:

```
// preinkrementacja
CVector2D CVector2D::operator++()      { ++m_fX; ++m_fY; return *this; }

// postinkrementacja
CVector2D CVector2D::operator++(int)
{
   CVector2D vWynik = *this;
   ++(*this);
   return vWynik;
}

// (dekrementacja przebiega analogicznie)
```

Spostrzeżmy, że nic nie stoi na przeszkodzie, aby w postinkrementacji użyć operatora preinkrementacji:

```
++(*this);
```

Przy okazji można dostrzec wyraźnie, dlaczego wariant prefiskowy jest wydajniejszy. W odmianie przyrostkowej trzeba przecież ponieść koszt stworzenia tymczasowego obiektu, aby go potem zwrócić jako rezultat.

Typowe operatory dwuargumentowe

Operatory dwuargumentowe, czyli binarne, przyjmują po argumenty. Powiedzmy sobie od razu, że **nie muszą** być to operandy tych samych typów. Wobec tego nie ma czegoś takiego, jak ogólna składnia prototypu funkcji operatora binarnego.

Ponownie jednak możemy mieć do czynienia z dwoma drogami implementacji takiej funkcji:

- jako metody jednej z klas, na obiektach której pracuje operator. Jego jawne wywołanie wygląda wówczas tak:

  ```
  operand1.operator symbol(operand2)
  ```

- jako funkcji globalnej - zaprzyjaźnionej bądź nie:

  ```
  operator symbol(operand1, operand2)
  ```

Obie linijki zastępują normalne użycie operatora w formie:

```
operand1 symbol operand2
```

O tym, która możliwość przeciążania jest lepsza, wspominałem już na początku. Przy wyborze największą rolę odgrywają ewentualne niejawne konwersje - jeżeli chcemy, by kompilator takowych dokonywał.
W bardzo uproszczonej formie można powiedzieć, że jeśli jednym z argumentów ma być typ wbudowany, to funkcja operatorowa jest dobrym kandydatem na globalną (z przyjaźnią bądź nie, zależnie od potrzeb). W innym przypadku możemy pozostać przy metodzie klasy - lub kierować się innymi przesłankami, jak w poniższych przykładach…

Celem ujrzenia tych przykładów wróćmy do naszego wektora. Jak wiemy, na wektorach w matematyce możemy dokonywać mnóstwa operacji. Nie wszystkie nas interesują, więc tutaj zaimplementujemy sobie tylko:

- dodawanie i odejmowanie wektorów
- mnożenie i dzielenie wektora przez liczbę
- iloczyn skalarny

Czy będzie to trudne? Myślę, że ani trochę. Zacznijmy od dodawania i odejmowania:

```
class CVector2D
{
   // (pomijamy szczegóły)

   // dodawanie
   friend CVector2D operator+(const CVector2D& vWektor1,
                              const CVector2D& vWektor2)
   {
         return CVector2D(vWektor1.m_fX + vWektor2.m_fX,
                          vWektor1.m_fY + vWektor2.m_fY);
   }

   // (analogicznie definiujemy odejmowanie: operator-())
};
```

Zastosowałem tu funkcję zaprzyjaźnioną - przypominam przy okazji, że nie jest to metoda klasy `CVector2D`, choć pewnie na to wygląda. Umieszczenie jej wewnątrz bloku klasy to po prostu zaakcentowanie faktu, że funkcja niejako należy do „definicji” wektora - nie tej *stricte* programistycznej, ale matematycznej. Oprócz tego pozwala nam to na zgrupowanie wszystkich funkcji związanych z wektorem w jednym miejscu, no i na czerpanie zalet wydajnościowych, bo przecież `operator+()` jest tu funkcją *inline*.

Kolejny punkt programu to mnożenie i dzielenie przez liczbę. Tutaj opłaca się zdefiniować je jako metody klasy:

```
class CVector2D
{
   // (pomijamy szczegóły)

   public:
         // (tu też)

         // mnożenie wektor * liczba
         CVector2D operator*(float fLiczba) const
               { return CVector2D(m_fX * fLiczba, m_fY * fLiczba); }

         // (analogicznie definiujemy dzielenie: operator/())
};
```

Dlaczego? Ano dlatego, że pierwszy argument ma być naszym wektorem, zatem odpowiada nam fakt, iż będzie to `this`. Drugi operand deklarujemy jako liczbę typu `float`.

Ale chwileczkę… Przecież mnożenie jest przemienne! W naszej wersji operatora * liczba może jednak stać tylko po prawej stronie!

„Ha, a nie mówiłem! `operator*()` jako metoda jest niepoprawny - trzeba zdefiniować go jako funkcję globalną!” Hola, nie tak szybko. Faktycznie, powyższa funkcja nie wystarczy, ale to nie znaczy, że mamy ją od razu wyrzucać. Przy zastosowaniu funkcji globalnych musielibyśmy przecież także napisać ich dwie sztuki:

```
CVector2D operator*(const CVector2D& vWektor, float fLiczba);
CVector2D operator*(float fLiczba, const CVector2D& vWektor);
```

W każdym więc przypadku jeden `operator*()` nie wystarczy[113]. Musimy dodać jego kolejną wersję:

```
class CVector2D
{
   // (pomijamy szczegóły)

   // mnożenie liczba * wektor
   friend CVector2D operator*(float fLiczba, const CVector2D& vWektor)
         { return vWektor * fLiczba; }
};
```

Korzystamy w niej z uprzednio zdefiniowanej. Kwestia, czy należy poprzednią wersję operatora także zamienić na zwykłą funkcję zaprzyjaźnioną, jest otwarta. Jeżeli razi cię niekonsekwencja (jeden wariant jako metoda, drugi jako zwykła funkcja), możesz to zrobić.

Na koniec dokonamy… trzeciej definicji `operator*()`. Tym razem jednak będzie to operator mnożenia dwóch wektorów - czyli iloczynu skalarnego (ang. *dot product*). Przypomnijmy, że takie działanie jest po prostu sumą iloczynów odpowiadających sobie współrzędnych wektora. Jego wynikiem jest więc pojedyncza liczba.
Ponieważ operator będzie działał na dwóch obiektach `CVector2D`, decyzja co do sposobu jego zapisania nie ma znaczenia. Aby pozostać w zgodzie z tym ustalonym dla operatorów dodawania i mnożenia, niech będzie to funkcja zaprzyjaźniona:

```
class CVector2D
{
   // (pomijamy szczegóły)

   // iloczyn skalarny
   friend float operator*(const CVector2D& vWektor1,
                          const CVector2D& vWektor2)
   {
         return vWektor1.m_fX * vWektor2.m_fX,
                + vWektor1.m_fY * vWektor2.m_fY;
   }
};
```

Definiowanie operatorów binarnych jest więc bardzo proste, czyż nie? :D

Operatory przypisania

Teraz porozmawiamy sobie o pewnym wyjątkowym operatorze. Jest on unikalny pod wieloma względami; mowa o **operatorze przypisania** (ang. *assignment operator*) tudzież podstawienia.

Dość często nie potrzebujemy nawet jego wyraźnego zdefiniowania. Kompilator dla każdej klasy generuje bowiem taki operator, o domyślnym działaniu. Taki automatyczny operator dokonuje przypisania „składnik po składniku" - tak więc po jego zastosowaniu przypisywane obiekty są sobie równe na poziomie wartości pól[114]. Taka sytuacja nam często odpowiada - przykładowo, dla naszej klasy `CVector2D` będzie to idealne rozwiązanie. Niekiedy jednak nie jest to dobre wyjście - za chwilę zobaczymy, dlaczego. Powiedzmy jeszcze tylko, że domyślny operator przypisania **nie jest** tworzony przez kompilator, jeżeli klasa:

[113] Pomijam tu zupełnie fakt, że za chwilę funkcję tę zdefiniujemy po raz trzeci - tym razem jako iloczyn skalarny dwóch wektorów.
[114] W tym kopiowanie „pole po polu" wykorzystywane są aczkolwiek indywidualne operatory przypisania od klas, które instancjujemy w postaci pól. Nie zawsze więc obiekty takie faktycznie są sobie doskonale równe.

- ma składnik będący stałą (`const typ`) lub stałym wskaźnikiem (`typ* const`)
- posiada składnik będący referencją
- istnieje prywatny (`private`) operator przypisania:
  - ✓ w klasie bazowej
  - ✓ w klasie, której obiekt jest składnikiem naszej klasy

Nawet jeśli żaden z powyższych punktów nie dotyczy naszej klasy, domyślne działanie operatora przypisania może nam nie odpowiadać. Wtedy należy go zdefiniować samemu w ten oto sposób:

```
klasa& klasa::operator=(const klasa&);
```

Jest to najczęstsza forma występowania tego operatora, umożliwiająca kontrolę przypisywania obiektów tego samego typu co macierzysta klasa. Możliwe jest aczkolwiek **przypisywanie dowolnego typu** - czasami jest to przydatne.
Jest jednak coś, na co musimy zwrócić uwagę w pierwszej kolejności:

Operatory przypisania (zarówno prosty, jak i te złożone) muszą być zdefiniowane jako **niestatyczna funkcja składowa** klasy, na której pracują.

Widać to z zaprezentowanej deklaracji. Nie widać z niej jednak, że:

Przeciążony operator przypisania **nie jest dziedziczony**.

Dlaczego - o tym mówiłem przy okazji wprowadzania samego dziedziczenia.

OK, wystarczy tej teorii. Czas zobaczyć definiowanie tego opratora w praktyce.
Wspomniałem już, że dla klasy `CVector2D` w zupełności wystarczy operator tworzony przez kompilator. Mamy jednak inną klasę, dla której jest to wręcz niedopuszczalne rozwiązanie. To `CIntArray`, nasza tablica liczb.
Dlaczego nie możemy skorzystać dla z niej z przypisania „składnik po składniku”? Z bardzo prostego powodu: spowoduje to przecież skopiowanie wskaźników na tablice, a nie samych tablic.
Zauważmy, że z tego samego powodu napisaliśmy dla `CIntArray` konstruktor kopiujący. To nie przypadek.

Jeżeli klasa musi mieć konstruktor kopiujący, to najprawdopodobniej potrzebuje także własnego operatora przypisania (i na odwrót).

Zajmijmy się więc napisaniem tego operatora. Aby to uczynić, pomyślmy, co powinno się stać w takim przypisaniu:

```
CIntArray aTablica1(7), aTablica2(8);
aTablica1 = aTablica2;
```

Po jego dokonaniu obie tablice musza zawierać te same elementy, lecz jednocześnie być **niezależne** - modyfikacja jednej nie może pociągać za sobą zmiany zawartości drugiej.
Operator przypisania musi więc:
- zniszczyć tablicę w obiekcie `aTablica1`
- zaalokować w tym obiekcie tyle pamięci, aby pomieścić zawartość `aTablica2`
- skopiować ją tam

Te trzy kroki są charakterystyczne dla większości implementacji operatora przypisania.
Dzielą one kod funkcji operatorowej na dwie części:
- część „destruktorową”, odpowiedzialną za zniszczenie zawartości obiektu, który jest celem przypisania

- część „konstruktorową”, zajmującą się kopiowaniem

Nie można jednak ograniczyć go do prostego wywołania destruktora, a potem konstruktora kopiującego - choćby z tego względu, że tego drugiego nie da się tak po prostu wywołać.

Dobrze, teraz to już naprawdę zaczniemy coś kodować :) Napiszemy operator przypisania dla klasy `CIntArray`:

```
CIntArray& CIntArray::operator=(const CIntArray& aTablica)
{
   // usuwamy naszą tablicę
   delete[] m_pnTablica;

   // alokujemy tyle pamięci, aby pomieścić przypisywaną tablicę
   m_uRozmiar = aTablica.m_uRozmiar;
   m_pnTablica = new int [m_uRozmiar];

   // kopiujemy tablicę
   memcpy (m_pnTablica, aTablica.m_pnTablica, m_uRozmiar * sizeof(int));

   // zwracamy wynik
   return *this;
}
```

Nie jest on chyba niespodzianką - mamy tu wszystko, o czym mówiliśmy wcześniej. Tak więc na początku zwalniamy tablicę w obiekcie, będącym celem przypisania. Później alokujemy nową - na tyle dużą, aby zmieścić przypisywany obiekt. Wreszcie dokonujemy kopiowania.

I pewnie jeszcze tylko jedna sprawa zaprząta twoją uwagę: dlaczego funkcja zwraca w wyniku `*this`?...
Nie jest trudno odpowiedzieć na to pytanie. Po prostu realizujemy tutaj konwencję znaną z typów podstawowych, mówiącą o rezultacie przypisania, Pozwala to też na dokonywanie wielokrotnych przypisać, np. takich:

```
CIntArray aTablica1(4), aTablica2(5), aTablica3(6);
aTablica1 = aTablica2 = aTablica3;
```

Powyższy kod bedzie działał identycznie, jak dla typów podstawowych. Wszystkie tablice staną się więc kopiami obiektu `aTablica3`.
Aby to osiągnąć, wystarczy trzymać się prostej zasady:

Operator przypisania powinien zwracać **referencję do `*this`**.

Wydawałoby się, że teraz wszystko jest już absolutnie w porządku, jeżeli chodzi o przypisywanie obiektów klasy `CIntArray`. Niestety, znowu zawodzi nas czujność. Popatrzmy na taki oto kod:

```
CIntArray aTablica;
aTablica = aTablica;       // co się stanie z tablicą?
```

Być może przypisywanie obiektu do niego samego jest dziwne, ale jednak kompilator dopuszcza je dla typów podstawowych, gdyż jest dla nich nieszkodliwe. Nie można tego samego powiedzieć o naszej klasie i jej operatorze przypisania.
Wywołanie funkcji `operator=()` spowoduje bowiem **usunięcie wewnętrznej tablicy** w obu obiektach (bo są one przecież jednym i tym samym bytem), a następnie próbę

skopiowania tej **usuniętej tablicy** do nowej! Będziemy mogli mówić o szczęściu, jeśli spowoduje to „tylko” błąd *access violation* i awaryjne zakończenie programu…

Przed taką ewentualnością musimy się więc zabezpieczyć. Nie jest to trudne i ogranicza się do prostego sprawdzenia, czy nie mamy do czynienia z przypisywaniem obiektu do jego samego. Robimy to tak:

```
klasa& klasa::operator=(const klasa& obiekt)
{
   if (&obiekt == this)    return *this;

   // (reszta instrukcji)
}
```

albo tak:

```
klasa& klasa::operator=(const klasa& obiekt)
{
   if (&obiekt != this)
   {
         // (reszta instrukcji)
   }
}
```

W instrukcji `if` porównujemy wskaźniki: adres przypisywanego obiektu oraz `this`. W ten wyłapujemy ich ewentualną identyczność i zapobiegamy katastrofie.

Operator indeksowania

Skoro jesteśmy już przy naszej tablicy, warto zająć się operatorem o wybitnie tablicowym charakterze. Mówię oczywiście o nawiasach kwadratowych `[]`, czyli **operatorze indeksowania** (ang. *subscript operator*).

Operator ten definiujemy zwykle w taki oto sposób:

```
typ_wartości& klasa::operator[](typ_klucza);
```

Znowu widzimy, że jest to metoda klasy i po raz kolejny nie jest to przypadkiem:

Operator indeksowania musi być zdefiniowany jako **niestatyczna metoda klasy**.

To już drugi operator, którego dotyczy taki wymóg. Podpada pod niego jeszcze następna dwójka, której przeciążanie omówimy za chwilę. Najpierw zajmijmy się operatorem indeksowania.

Przede wszystkim chciałbyś pewnie wiedzieć, jak on działa. Nie jest to trudne; jeżeli przeciążymy ten operator, to wyrażenie w formie:

```
obiekt[klucz]
```

zostanie przez kompilator zinterpretowane jako wywołanie w postaci:

```
obiekt.operator[](klucz)
```

Do funkcji operatorowej poprzez parametr trafia więc `klucz`, czyli wartość, jaką podajemy w nawiasach kwadratowych. Co ciekawe, nie musi to być wcale wartość typu `int`, ani nawet wartość liczbowa - równie dobrze sprawdza się tu całkiem **dowolny typ**

danych, nawet napisy. Pozwala to tworzyć klasy tzw. tablic asocjacyjnych, znanych na przykład z języka PHP[115].

Ponieważ wspomniałem już o tablicach, zajmijmy się tą, która sami kiedyś napisaliśmy i ciągle udoskonalamy. Nie da się ukryć, że `CIntArray` wiele zyska na przeciążeniu operatora `[]`. Jeżeli zrobimy to umiejętnie, będzie można używac go tak samo, jak czynimy to w stosunku do zwykłych tablic języka C++.
Aby jednak to zrobić, musimy zwrócić uwagę na pewien szczególny fakt. W stosunku do typów wbudowanych operator `[]` jest mianowicie bardzo elastyczny: w szczególności pozwala on zarówno na odczyt, jak i modyfikację elementów tablicy:

```
int aTablica[10]
aTablica[7] = 100;          // zapis
std::cout << aTablica[7];  // odczyt
```

Wyrażenie z operatorem `[]` może stać zarówno po lewej, jak i po prawej stronie znaku przypisania. Tę cechę wypadałoby zachować we własnej jego wersji - znaczy to, że:

Operator indeksowania powinien **w wyniku zwracać l-wartość**.

Gwarantuje to, że jego użycie będzie zgodne z tym dla typów podstawowych. Zaakcentowałem ten wymóg, pisząc w składni operatora referencję jako typ zwracanej wartości. To właśnie spowoduje pożądane zachowanie.

Jeżeli nie możemy sobie pozwolić sobie na zwracanie l-wartości, to powinniśmy raczej całkowicie zrezygnować z przeładowania operatora `[]` i poprzestać na metodach dostępowych - takich jak `Pobierz()` i `Ustaw()` w klasie `CIntArray`.

Zabierzmy się teraz do pracy: napiszemy przeciążoną wersję operatora indeksowania dla klasy `CIntArray`. Dzięki temu będziemy mogli manipulować elementami tablicy w taki sam sposób, jaki znamy dla normalnych tablic. To będzie całkiem spory krok naprzód. Osiągnięcie tego nie jest przy tym trudne - wręcz przeciwnie, u nas będzie niezwykle proste:

```
int& CIntArray::operator[](unsigned uIndeks)
   { return m_pnTablica[uIndeks]; }
```

To wszystko! Zwrócenie referencji do elementu w prawidziwej, wewnętrznej tablicy pozwoli na niczym nieskrępowany dostęp do jej zawartości. Teraz możemy w wygodny sposób odczytywać i zapisywać liczby w naszej tablicy:

```
CIntArray aTablica(4);

aTablica[0] = 1;
aTablica[1] = 4;
aTablica[2] = 9;
aTablica[3] = 16;

for (unsigned i = 0; i < aTablica.Rozmiar(); ++i)
   std::cout << aTablica[i] << ", ";
```

Obecnie jest już ona funkcjonalnie identyczna z tablicą typu `int[]`. Możemy jednak zacząć czerpać także pewne korzyści z napisania tej klasy. Skoro juz przeciążamy

[115] Zazwyczaj lepszym rozwiązaniem jest skorzystanie z mapy STL, czyli klasy `std::map`. Omówimy ją, kiedy przejdziemy do opisu klas pojemnikowych Biblioteki Standardowej.

operator `[]`, to zadbajmy, aby wykonywał po drodze jakąś pożyteczną czynność - na przykład sprawdzał poprawność żądanego indeksu:

```
int& CIntArray::operator[](unsigned uIndeks)
   { return m_pnTablica[uIndeks < m_uRozmiar ? uIndeks : m_uRozmiar-1];
}
```

Przy takiej wersji funkcji nie grozi nam już błąd przekroczenia zakresu (ang. *subscript out of range*). W razie podania nieprawidłowego numeru elementu, funkcja zwróci po prostu odwołanie do ostatniej liczby w tablicy. Nie jest to najlepsze rozwiązanie, ale przynajmniej zabezpiecza przed błędem czasu wykonania.

Znacznie lepszym wyjściem jest rzucenie wyjątku, który poinformuje wywołującego o zainstaniałym problemie. O wyjątkach porozmawiamy sobie w następnym rozdziale.

Operatory wyłuskania

C++ pozwala na przeładowanie dwóch operatorów wyłuskania: `->` oraz `->*`. Nie jest to częsta praktyka, a jeśli nawet jest stosowana, to przeciążaniu podlega zwykle tylko pierwszy z tych operatorów. Możesz więc pominąć ten akapit, jeżeli nie wydaje ci się konieczna znajomość sposobu przeładowywania operatorów wyłuskania.

Operator ->

Operator `->` kojarzy nam się z wybieraniem składnika poprzez wskaźnik do obiektu. Wygląda to np. tak:

```
CFoo* pFoo = new CFoo;
pFoo->Metoda();
delete pFoo;
```

Jeżeli jednak spróbowaliśmy użyć tego operatora w stosunku do samego obiektu (lub referencji do niego):

```
CFoo Foo;
Foo->Metoda();        // !!!
```

to bez wątpienia otrzymalibyśmy komunikat o błędzie. Domyślnie nie jest bowiem możliwe użycie operatora `->` w stosunku do samych obiektów. Jest on aplikowalny tylko do wskaźników.

Ale w C++ nawet ta żelazna może zostać nagięta. Możliwe jest bowiem nadanie operatorowi `->` znaczenia i dopuszczenie do jego używania razem ze zmiennymi obiektowymi. Aby to uczynić, trzeba oczywiście przeciążyć ten operator.
Czynimy to taką oto funkcją:

```
jakaś_klasa* klasa::operator->();
```

Nie wygląda ona na skomplikowaną… ale znowu jest to metoda klasy! Tak więc:

Operator wyłuskania `->` musi być **niestatyczną funkcją składową** klasy.

Powiedzmy sobie teraz, jak on działa. Nie jest przecież wcale takie oczywiste - choćby z tego względu, że z niewiadomych na razie powodów operator zadowala się zaledwie jednym argumentem… (Jest on rzecz jasna przekazywany poprzez wskaźnik `this`)
A oto i odpowiedź. Kiedy przeciążymy operator `->`, wyrażenie w formie:

```
obiekt->składnik
```

zostanie zmienione na:

```
(obiekt.operator->())->składnik
```

Mamy tu już jawne wywołanie `operator->()`, ale nadal pojawia się strzałka w swej normalnej postaci. Otóż jest to konieczne; w tym kodzie `->` stojący tuż przy składniku jest już bowiem **zwykłym operatorem** wyłuskania `->`. Zwykłym - to znaczy takim, który oczekuje wskaźnika po swojej lewej stronie - a nie obiektu, jak operator przeciążony. Wynika z tego wyrażenie:

```
obiekt.operator->()
```

musi reprezentować wskaźnik, aby całość działała poprawnie. Dlatego też funkcja `operator->()` zwraca w wyniku typ wskaźnikowy. Jednocześnie nie interesuje się ona tym, co stoi po prawej stronie strzałki - to jest już bowiem sprawą tego normalnego, wbudowanego w kompilator operatora `->`.
Podsumowując, można powiedzieć, że:

Funkcja `operator->()` dokonuje raczej **zamiany obiektu na wskaźnik** niż faktycznego przedefiniowania znaczenia operatora `->`.

Godne uwagi jest to, że wskaźnik zwracany przez tę funkcje wcale nie musi być wskaźnikiem na obiekt jej macierzystej klasy. Może to być wskaźnik na **dowolną klasę**, co zresztą obrazuje składnia funkcji.

Zastanawiasz się pewnie: „A po co mi przeciążanie tego operatora? Może po to, aby do składników obiektu odnosić się nie tylko kropką (.), ale i strzałką (->)?" Odradzam przeciążanie operatora w tym celu, bo to raczej ukryje błędy w kodzie niż ułatwi programowanie.
Operator -> możemy jednak przeciążyć i będzie to przydatne przy pisaniu klas tzw. **inteligentnych wskaźników**.

Inteligentny wskaźnik (ang. *smart pointer*) to klasa będąca opakowaniem dla normalnych wskaźników i zapewniająca wobec nich dodatkowe, „inteligentne" zachowanie.

Rodzajów tych inteligentnych zachowań jest doprawdy mnóstwo. Może to być kontrola odwołań do wskaźnika - zarówno w prostej formie zliczania, jak i zaawansowanej komunikacji z mechanizmem zajmującym się usuwaniem nieużywanych obiektów (odśmiecaczem, ang. *garbage collector*). Innym zastosowaniem może być ochrona przed wyciekami pamięci spowodowanymi nagłym opuszczeniem zakresu.

My napiszemy sobie najprostszą wersję takiego wskaźnika. Będzie on przechowywał odwołanie do obiektu `CFoo`, które przekażemy mu w konstruktorze, i zwalniał je w swoim destruktorze. Oto kod klasy wskaźnika:

```
class CFooSmartPtr
{
   private:
         // opakowywany, właściwy wskaźnik
         CFoo* m_pWskaznik;

   public:
         // konstruktor i destruktor
         CFooSmartPtr(CFoo* pFoo) : m_pWskaznik(pFoo)    { }
         ~CFooSmartPtr()   { if (m_pWskaznik) delete m_pWskaznik; }
```

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

        // operator dereferencji
        CFoo operator*()        { return *m_pWskaznik; }

        // operator wyłuskania
        CFoo* operator->()      { return m_pWskaznik }
};
```

Ta klasa jest uboższą wersją `std::auto_ptr` z Biblioteki Standardowej. Służy ona do bezpiecznego obchodzenia się z pamięcią w sytuacjach związanych z wyjątkami. Omówimy ją sobie w następnym rozdziale (wrócimy tam zresztą także i do powyższej klasy).

Co nam daje taki wskaźnik? Jeżeli go użyjemy, to zapobiegnie on wyciekowi pamięci, który może zostać spowodowany przez nagłe opuszczenie zakresu (np. w wyniku wyjątku - patrz następny rozdział). Jednocześnie nie umniejszamy sobie w żaden sposób wygody kodowania - nadal możemy korzystać ze składni, do której się przyzwyczailiśmy:

```
CFooSmartPtr pFoo = new CFoo;

// wywołanie metody na dwa sposoby
pFoo->Metoda();            // naprawdę: (pFoo.operator->())->Metoda()
(*pFoo).Metoda();          // naprawdę: (pFoo.operator*()).Metoda()
```

Proszę tylko nie sądzić, że odtąd powinniśmy używać tylko takich sprytnych wskaźników. O nie, one nie są panaceum na wszystko i mają całkiem konkretne zastosowania. Nie należy ich traktowac jako złoty środek - szczególnie jako środek przeciwko zapomnialskiemu niezwalnianiu zaalokowanej pamięci.

Ciekawostka: operator `->*`

Drugi z operatorów wyłuskania, `->*`, jest bardzo rzadko używany. Nie dziwi więc, że sytuacje, w których jest on przeciążany, są wręcz sporadyczne. Niemniej, skoro już mówimy o przeciążaniu, to możemy wspomnieć także o nim.

Wpierw przydałoby się aczkolwiek, abyś znał mechanizm wskaźników na składowe klasy, opisany w następnym podrozdziale.

`->*` jest używany do wybierania składników obiektu poprzez wskaźniki do składowych. Podobnie jak `->`, nie ma on predefiniowanego znaczenia dla zmiennych obiektowych, a jedynie dla wskaźników na obiekty. Na tym jednak podobieństwa się kończą.

`->*` jest przeciążany jako **operator binarny** dla **konkretnego zestawu** dwóch danych, które stanowią:

- referencja do obiektu (argument lewostronny)
- wskaźnik do składowej klasy (argument prawostronny)

Nie ma też wymogu, aby funkcja `operator->*()` była funkcją składową klasy. Może być równie dobrze funkcją globalną.

Jak więc przeciążyć ten operator? Ponieważ, jak mówiłem, definiujemy go dla konkretnego typu składnika, postać prototypu funkcji `operator->*()` różni się dla wskaźników do pól oraz do metod klasy.
W pierwszym przypadku składnia przeciążenia wygląda mniej więcej tak:

```
typ_pola& klasa::operator->*(typ_pola klasa::*);
typ_pola& operator->*(klasa&, typ_pola klasa::*);
```

Jest chyba dość logiczne, że typ docelowego pola oraz typ zwracany przez funkcję operatorową musi się zgadzać. Dość podobnie jest dla metod:

```
zwracany_typ klasa::operator->*(zwracany_typ (klasa::*)([parametry]));
zwracany_typ operator->*(klasa&, zwracany_typ (klasa::*)([parametry]));
```

Tutaj funkcja musi zwracać ten sam typ, co metoda klasy, na której wskaźnik przyjmujemy.

Jak wygląda przeciążanie w praktyce? Spójrzmy na przykład na taką oto klasę:

```
class CFoo
{
   public:
         int nPole1, nPole2;

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

         // operator ->*
         int& operator->*(int CFoo::*)       { return nPole1; }
};
```

Po takim redefiniowaniu operatora, wszystkie wskaźniki na składowe typu `int` w klasie `CFoo` będą „prowadziły” tylko i wyłącznie do pola `nPole1`.

Operator wywołania funkcji

Czas na kolejny operator, chyba jeden z bardziej interesujących. To **operator wywołania funkcji** (ang. *function-call operator*), czyli nawiasy okrągłe `()`.

Nawiasy mają jeszcze dwa znaczenia w C++: grupują one wyrażenia oraz pozwalają wykonywać rzutowanie (w stylu C lub funkcyjnym). Żadnego z tych pozostałych znaczeń nie możemy jednak zmieniać. Przeciążeniu może ulec tylko operator wywołania funkcji.

Tak jest, on także może być przeciążony. O czym w tym przypadku należy pamiętać?… Otóż:

Operator wywołania funkcji może być zdefiniowany tylko jako **niestatyczna funkcja składowa** klasy.

Jest to ostatni rodzaj operator, którego dotyczy to ograniczenie. Przypominam, że pozostałymi są: operatory przypisania, indeksowania oraz wyłuskania (`->`).

Na tym zastrzeżeniu kończą się jednak jakiegolwiek obostrzeżenia nakładane na to przeciążenie. `operator()()` (tak, dwie pary nawiasów) może być bowiem funkcją przyjmującą **dowolne argumenty** i zwracającą **dowolny typ** wartości:

```
zwracany_typ klasa::operator()([parametry]);
```

To jedyny operator, który może przyjmować każdą ilość argumentów. To zresztą zrozumiałe: skoro normalnie służy on do wywoływania funkcji, mogących mieć przecież dowolną liczbę parametrów, to i jego przeciążona wersja nie powinna nakładać ograniczeń w tym zakresie. Podobnie dzieje się, jeżeli chodzi o typ zwracanej wartości. Oznacza to również, że możliwe jest zdefiniowanie wielu wersji przeciążonego operatora `()`. Muszą one jednak być rozróżnialne w tym sam sposób, jak przeładowane funkcje. Powinny więc posiadać inną liczbę, kolejność i/lub typy parametrów.

Do czego może nam przydać się taka potęga i elastyczność? Możliwości jest bardzo wiele, może do nich należeć np. wybór elementu tablicy wielowymiarowej. Do ciekawszych zastosowań należy jednak tworzenie tzw. **obiektów funkcyjnych** (ang. *function objects*) - **funktorów**.
Funktory są to obiekty przypominające zwykłe funkcje, jednak różnią się tym, iż mogą posiadać stan. Mają go, ponieważ w rzeczywistości są to klasy, które zawierają jakieś publiczne pola, zaś składnię wywołania funkcji uzyskują za pomocą przeciążenia operatora `()`.
Oto prosty przykład - funktor obliczający średnią arytmetyczną z podanych liczb i aktualizujący wynik z każdym kolejnym wywołaniem:

```
class CAverageFunctor
{
   private:
         // aktualny wynik
         double m_fSrednia;

         // ilość wywołań
         unsigned m_uIloscLiczb;

   public:
         // konstruktor
         CAverageFunctor() : m_fSrednia(0.0), m_uIloscLiczb(0) { }

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

         // funkcja resetująca stan funktora
         void Reset()      { m_fSrednia = m_uIloscLiczb = 0; }

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

         // operator wywołania funkcji - oblicza średnią
         double operator()(double fLiczba)
         {
               // liczymy nową średnią, uwzględniającą dodaną liczbę
               // oraz aktualizujemy zmienną przechowują ilość liczb
               // wszystko w jednym wyrażeniu - za to kochamy C++ ;D
               m_fSrednia = ((m_fSrednia * m_uIloscLiczb) + fLiczba)
                              / m_uIloscLiczb++);

               // zwracamy nową średnią
               return m_fSrednia;
         }
};
```

Użycie tego obiektu wygląda tak:

```
CAverageFunctor Srednia;

Srednia(4);                 // średnia z 4
Srednia(18.5);              // średnia z 4 i 18.5
Srednia(-6);                // średnia z 4, 18.5 i -6
Srednia(42);                // średnia z 4, 18.5, -6 i 42
Srednia.Reset();            // zresetowanie funktora, wartość przepada

Srednia(56);                // średnia z 56
Srednia(90);                // średnia z 56 i 90
Srednia(4 * atan(1));       // średnia z 56, 90 i pi
std::cout << Srednia(13);   // wyświetlenie średniej z 56, 90, pi i 13
```

Naturalnie, matematycy złapaliby się za głowę widząc taki algorytm obliczania średniej. Bardzo skutecznie prowadzi on bowiem to kumulowania błędów związanych z niedokładnym zapisem liczb w komputerze. Jest to jednak całkiem dobra ilustracja koncepcji funkctora.
W Bibliotece Standardowej mamy całkiem sporo klas funktorów, z którymi będziesz mógł się wkrótce zapoznać.

Operatory zarządzania pamięcią

Oto kolejne dwa wyjątkowe operatory: `new` i `delete`. Jak doskonale wiemy, służą one do dynamicznego tworzenia w pamięci operacyjnej (a dokładniej na stercie) zmiennych, tablic i obiektów. To może wydawać się niemal niesamowite, ale je także możemy przeładować!

Wpierw jednak muszę przypomnieć, że praca tych operatorów nie ogranicza się w rzeczywistości tylko do przydzielenia pamięci (`new`) i jej zwolnienia (`delete`). Jesteśmy świadomi, że może za tym iść także zainicjowanie lub sprzątniecie alokowanego obszaru pamięci. Oznacza to na przykład wywołanie konstruktora (`new`) i destruktora (`delete`) klasy, której obiekt tworzymy.
Widzimy więc, że oba operatory wykonują więcej niż jedną czynność. Zmodyfikować możemy jednak tylko jedną z nich:

Przeciążone operatory `new` i `delete` mogą jedynie **zmienić sposób alokowania i zwalniania pamięci**. Nie można ingerować w inicjalizację (wywołanie konstruktorów) i sprzątanie (przywołanie destruktorów), które temu towarzyszą.

Zauważmy, że fakt ten niweluje dla nas różnice między operatorem `new` a `new[]` oraz `delete` i `delete[]`. Na poziomie alokacji (zwalniania) pamięci niczym się one bowiem nie różnią. Dlatego też dla potrzeb przeciążania mówimy tylko o operatorach `new` i `delete`, mając jednak w pamięci tę uwagę.

Czy to, że kontrolujemy jedynie zarządzanie pamięcią znaczy, że przeciążanie tych operatorów nie jest interesujące?… Przeciwnie - alokacja i zwalnianie pamięci to są właśnie te czynności, które najbardziej nas interesują. Napisanie własnego algorytmu ich wykonywania, albo chociaż śledzenia tych standardowych, jest podstawą działania tak zwanych **menedżerów pamięci** (ang. *memory managers*). Są to mechanizmy zajmujące się kontrolą wykorzystania pamięci operacyjnej, zapobiegające zwykle jej wyciekom i często optymalizujące program.

Stworzenie dobrego menedżera pamięci nie jest oczywiście proste, jednak przeciążenie `new` i `delete` to bardzo łatwa czynność. Aby ją wykonać, spójrzmy na prototypy obu funkcji - `operator new()` i `operator delete()`:

```
void* [klasa::]operator new(size_t);
void [klasa::]operator delete(void*);
```

To nie pomyłka: funkcje te mają ściśle określone listy parametrów oraz typy zwracanych wartości. W tym względzie jest to wyjątek wśród wszystkich operatorów.
`operator new()` przyjmuje jeden parametr typu `size_t` - jest to ilość bajtów, jaka ma być zaalokowana. W zamian powinien on zwrócić `void*` - jak można się domyślać: wskaźnik do przydzielonego obszaru pamięci o żądanym rozmiarze.
Z kolei funkcja dla operatora `delete` potrzebuje tylko parametru, będącego wskaźnikiem. Jest to rzecz jasna wskaźnik do obszaru pamięci, który ma być zwolniony. W zamian funkcja zwraca `void`, czyli nic. Oczywiste.

Mniej oczywista jest opcjonalna fraza `klasa::`. Owszem, sugeruje ona, że obie funkcje mogą być metodami klasy lub funkcjami globalnymi. W przeciwieństwie do pozostałych operatorów ma to jednak znaczenie: `new` i `delete` jako metody mają bowiem inne znaczenie niż `new` i `delete` - funkcje globalne. Mamy mianowicie możliwość lokalnego przeciążenia obydwu operatorów, jak również zdefiniowania ich nowych, globalnych wersji. Omówimy sobie oba te przypadki.

Lokalne wersje operatorów

Operatory `new` i `delete` możemy przeciążyć w stosunku do pojedynczej klasy. W takiej sytuacji będą one używane do alokowania i (lub) zwalniania pamięci dla obiektów **wyłącznie tej** klasy.
Może to się przydać np. do zapobiegania fragmentacji pamięci, spowodowanej częstym tworzeniem i zwalnianiem małych obiektów. W takim przypadku operator `new` może zarządzać większym kawałkiem pamięci i wirtualnie „odcinać" z niego mniejsze fragmenty dla kolejnych obiektów. `delete` dokonywałby wtedy tylko pozornej dealokacji pamięci.

Zobaczmy zatem, jak odbywa się przeładowanie lokalnych operatorów `new` i `delete`. Oto prosty przykład, korzystający w zasadzie ze standardowych sposób przydzielania i oddawania pamięci, ale jednocześnie wypisujący informacje o tych czynnościach:

```
class CFoo
{
   public:
         // new
         void* operator new(size_t cbRozmiar)
         {
               // informacja na konsoli
               std::cout << "Alokujemy " << cbRozmiar << " bajtow";

               // alokujemy pamięć i zwracamy wskaźnik
               return ::new char [cbRozmiar];
         }

         // delete
         void operator delete(void* pWskaznik)
         {
               // informacja
               std::cout << "Zwalniamy wskaznik " << pWskaznik;

               // usuwamy pamięć
               ::delete pWskaznik;
         }
};
```

Kiedy teraz spróbujemy stworzyć dynamicznie obiekt klasy `CFoo`:

```
CFoo* pFoo = new CFoo;
```

to odbędzie się to z jednoczesnym powiadomieniem o tym fakcie przy pomocy strumienia wyjścia. Analogicznie będzie w przypadku usunięcia:

```
delete pFoo;
```

Nadal jednak możemy skorzystać z normalnych wersji `new` i `delete` - wystarczy poprzedzić ich nazwy operatorem zakresu:

```
CFoo* pFoo = ::new CFoo;
```

```
// ...
::delete pFoo;
```

Tak też robimy w ciele naszych funkcji operatorowych. Mamy dzięki temu pewność, że wywołujemy standardowe operatory i nie wpadamy w pułapkę nieskończonej rekurencji. W przypadku lokalnych operatorów nie jest to bynajmniej konieczne, ale warto tak czynić dla zaznaczenia faktu korzystania z wbudowanych ich wersji.

Globalna redefinicja

new i delete możemy też przeładować w sposób całościowy i globalny. Zastąpimy w ten sposób wbudowane sposoby alokacji pamięci dla każdego użycia tych operatorów. Wyjątkiem będzie tylko jawne poprzedzenie ich operatorem zakresu, `::`.

Jak dokonać takiego fundamentalnego przeciążenia? Bardzo podobnie, jak to robiliśmy w „trybie lokalnym”. Tym razem nasze funkcje `operator new()` i `operator delete()` będa po prostu funkcjami globalnymi:

```
// new
void* operator new(size_t cbRozmiar)
{
   // informacja na konsoli
   std::cout << "Alokujemy " << cbRozmiar << " bajtow";

   // alokujemy pamięć i zwracamy wskaźnik
   return ::new char [cbRozmiar];
}

// delete
void operator delete(void* pWskaznik)
{
   // informacja
   std::cout << "Zwalniamy wskaznik " << pWskaznik;

   // usuwamy pamięć
   ::delete pWskaznik;
}
```

Ponownie pełnią one u nas wyłącznie funkcję monitorującą, ale to oczywiście nie jest jedyna możliwość. Wszystko zależy od potrzeb i fantazji.
Koniecznie zwróćmy jeszcze uwagę na sposób, w jaki w tych przeciążanych funkcjach odwołujemy się do oryginalnych operatorów new i delete. Używamy ich w formie `::new` i `::delete`, aby omyłkowo nie użyć własnych wersji… które przecież właśnie piszemy! Gdybyśmy tak nie robili, spowodowałoby to wpadnięcie w niekończący się ciąg wywołań rekurencyjnych. Pamiętajmy zatem, że:

Jeśli w treści przeciążonych, globalnych operatorów `new` i `delete` musimy skorzystać z ich standardowej wersji, **koniecznie** należy użyć formy `::new` i `::delete`.

Z domyślnych wersji operatorów pamięci możemy też korzystać świadomie nawet po ich przeciążeniu:

```
int* pnZmienna1 = new int;       // przeciażona wersja
int* pnZmienna2 = ::new int;     // oryginalna wersja
```

Naturalnie, trzeba wtedy zdawać sobie sprawę z tego przeciążenia i na własne życzenie użyć operatora `::`. To gwarantuje nam, że nikt inny, jak tylko kompilator będzie zajmował się zarządzaniem pamięci.

Nie wpadajmy jednak w paranoję. Jeżeli korzystamy z kodu, w którym zaimplementowano inny sposób nadzorowania pamięci, to nie należy bez wyraźnego powodu z niego rezygnować. W końcu po to ktoś (może ty?) pisał ów mechanizm, żeby był on wykorzystywany w praktyce, a nie z premedytacją omijany.

Cały czas mniej lub bardziej subtelnie sugeruję, że operatory `new` i `delete` należy przeciążać razem. Nie jest to jednak formalny wymóg języka C++ i jego kompilatorów. Zwykle jednak tak właśnie trzeba czynić, aby wszystko działało poprawnie - zwłaszcza, jeśli stosujemy inny niż domyślny sposób alokacji pamięci.

Operatory konwersji

Na koniec przypomnę jeszcze o pewnym mechanizmie, który w zasadzie nie zalicza się do operatorów, ale używa podobnej składni i dlatego także nazywamy go operatorami. Rzecz jasna są to **operatory konwersji**.

Składnia takich operatorów to po prostu:

```
klasa::operator typ();
```

Jak doskonale pamiętamy, celem funkcji tego typu jest zmiana obiektu klasy do danego typu. Przy jej pomocy kompilator może dokonywać niejawnych konwersji.
Innym (lecz nie zawsze stosowalnym) sposobem na osiągnięcie podobnych efektów jest konstruktor konwertujący. O obu tych drogach mówiliśmy sobie wcześniej.

Wskazówki dla początkującego przeciążacza

Przeciążanie operatorów jest wspaniałą możliwością języka C++. Nie ma jednak żadnego przymusu stosowania jej - dość powiedzieć, że do tej pory świetnie radziliśmy sobie bez niej. Nie ma aczkolwiek powodu, aby ją całkiem odrzucać - trzeba tylko nauczyć się ją właściwie wykorzystywać. Temu właśnie służy ten paragraf.

Zachowujmy sens, logikę i konwencję

Jakkolwiek język C++ jest znany ze swej elastyczności, przez lata jego użytkowania wypracowano wiele reguł, żądzących między innymi działaniem operatorów. Chcąc przeciążać operatory dla własnych klas, należałoby ich w miarę możliwości przestrzegać - zwłaszcza, że często są one zbieżne ze zdrowym rozsądkiem.

Podczas przeładowania operatorów trzeba po prostu zachować ich pierwotny sens. Jak to zrobić?…

Symbole operatorów powinny odpowiadać ich znaczeniom

W pierwszej kolejności należy powstrzymać się od radosnej twórczości, sprzecznej z wszelką logiką. Może i zabawne będzie użycie operatora `==` jako symbolu dodawania, `^` w charakterze operatora mnożenia i `&` jako znaku odejmowania. Pomyśl jednak, co w takiej sytuacji oznaczać będzie zapis:

```
if (Foo ^ Bar & (Baz  == Qux) == Thud)
```

Łagodnie mówiąc: nie jest to zbyt oczywiste, prawda? Pamiętaj zatem, żeby symbole operatorów odpowiadały ich naturalnym znaczeniom, a nie tworzyły uciążliwe dla programisty rebusy.

Zapewnijmy analogiczne zachowania jak dla typów wbudowanych

Wszystkie operatory posiadają już jakieś zdefiniowane działanie dla typów wbudowanych. Dla naszych klas może ono całkiem różnić się od tego początkowego, ale dobrze byłoby, aby przynajmniej zależności między poszczególnymi operatorami zostały zachowane.

Co to znaczy? Zauważmy na przykład, że trzy poniższe instrukcje:

```
int nA;

// o te
nA = nA + 1;
nA += 1;
nA++;
```

dla typu `int` (i dla wszystkich podstawowych typów) są w przybliżeniu równoważne. Dobrze byłoby, ale dla naszych przeładowanych operatorów te „tożsamości” zostały zachowane.
Podobnie jest dla typów wskaźnikowych:

```
CFoo* pFoo = new CFoo;

// instrukcje robiące to samo
pFoo->Metoda();
(*pFoo).Metoda();
// ewentualnie jeszcze pFoo[0].Metoda()

delete pFoo;
```

Jeśli tworzymy klasy inteligentnych wskaźników, należałoby wobec tego przeciążyć dla nich operatory `->`, `*` i ewentualnie `[]` (a także `operator bool()`, aby można je było stosować w wyrażeniach warunkowych).

Nie przeciążajmy wszystkiego

Na koniec jeszcze jedna, „oczywista” uwaga: nie ma sensu przeciążać wszystkich operatorów - przynajmniej do chwili, gdy nie piszemy klasy symulującej wszystkie typy w C++. Jeżeli mimo wszystko wykonamy tę niepotrzebną zwykle pracę i udostępnimy naszą pięknie opakowaną klasę innym programistom, najprawdopodobniej zignorują oni te przeciążenia, które nie będą miały dla nich sensu. A jeśli sami używać będziemy takiej klasy, to zapewne szybko sami przekonamy się, że uporczywe używanie operatorów nie ma zbytniego sensu. Drogą naturalnej selekcji w obu przypadkach zostaną więc w użyciu tylko te operatory, które są naprawdę potrzebne.
Nie powinniśmy jednak czekać, aż życie zweryfikuje nasze przypuszczenia, bo przeciążając niepotrzebnie operatory, stracimy mnóstwo czasu. Lepiej więc od razu zastanowić się, co warto przeładować, a czego nie. Kierujmy się w tym jedną, prostą zasadą:

Symbol operatora powinien kojarzyć się z czynnością przez niego wykonywaną.

Zastosowanie się do tej reguły likwiduje zazywczaj większość niepewności.

***

Zakończyliśmy w ten sposób poznawanie przydatnej techniki programowania, jaką jest przeciążanie operatorów dla naszych własnych klas.

W następnym podrozdziale, dla odmiany, zapoznamy się ze znacznie mniej przydatną techniką ;)) Chodzi o wskaźniki do składników klasy. Mimo tej mało zachęcającej zapowiedzi, zapraszam do przeczytania tego podrozdziału.

Wskaźniki do składowych klasy

W ostatnim rozdziale części pierwszej poznaliśmy zwykłe wskaźniki języka C: pokazujące na zmienne oraz na funkcje. Tutaj zajmiemy się pewną nowością, jaką do wskaźników wprowadziło programowanie obiektowe: **wskaźnikami do składowych** (ang. *pointers-to-members*).

Ten podrozdział nie jest niezbędny do kontynuowania nauki języka C++. Jeżeli stwierdzisz, że jest ci na razie niepotrzebny lub za trudny, możesz go opuścić. Zalecam to szczególnie przy pierwszym czytaniu kursu.

Podobnie jak dla normalnych wskaźników, wskaźniki na składowe także mogą odnosić się do danych (pól) oraz do kodu (metod). Omówimy sobie osobno każdy z tych rodzajów wskaźników.

Wskaźnik na pole klasy

Wskaźniki na pola klas są obiektowym odpowiednikiem zwykłych wskaźników na zmienne, jakie doskonale znamy. Funkcjonują one jednak nieco inaczej. Jak? O tym traktuje niniejsza sekcja.

Wskaźnik do pola wewnątrz obiektu

Przypomnijmy, jak wygląda zwykły wskaźnik - na przykład na typ `int`:

```
int nZmienna;
int* pnZmienna = &nZmienna;
```

Zadeklarowany tu wskaźnik `pnZmienna` został ustawiony na adres zmiennej `nZmienna`. Wobec tego poniższa linijka:

```
*pnZmienna = 14;
```

spowoduje przypisanie liczby `14` do `nZmienna`. Stanie się to za pośrednictwem wskaźnika.

Wskaźnik na obiekt

To już znamy. Wiemy też, że możemy tworzyć także wskaźniki do obiektów swoich własnych klas:

```
class CFoo
{
   public:
         int nSkladnik;
};

CFoo Foo;
CFoo* pFoo = &Foo;
```

Przy pomocy takich wskaźników możemy odnosić się do składników obiektu. W tym przypadku możemy na przykład zmodyfikować pole `nSkladnik`:

```
pFoo->nSkladnik = 76;
```

Sprawi to rzecz jasna, że zmieni się pole `nSkladnik` w obiekcie `Foo` - jego adres ma bowiem wskaźnik `pFoo`. Wypisanie wartości pola tego obiektu:

```
std::cout << Foo.nSkladnik;
```

uświadomi więc nam, że ma ono wartość 76. Ustawiliśmy ją bowiem za pośrednictwem wskaźnika. To też już znamy dobrze.

Pokazujemy na składnik obiektu

Czas więc na nowość. Pytanie brzmi: czy zwykłym wskaźnikiem można odnieść się do pola we wnętrzu obiektu?…

A owszem. Wystarczy pomyśleć, że wyrażenie:

```
Foo.nSkladnik
```

jest l-wartością typu `int`, zatem można pobrać jej adres zapisać we wskaźniku typu `int*`:

```
int* pnSkladnikFoo = &(Foo.nSkladnik);
```

Powiedzmy jeszcze wyraźnie, co tu zrobiliśmy. Otóż pobraliśmy adres konkretnego pola (`nSkladnik`) w konkretnym obiekcie (`Foo`). Jest to najzupełniej możliwe, bo przecież obiekt reprezentują w pamięci jego pola. Skoro zaś możemy odnieść się do obiektu jako całości, to możemy także pobrać adres jego pól.
Jeśli teraz wypiszemy wartośc pola przy pomocy tego wskażnika:

```
std::cout << *pnSkladnikFoo;
```

to zobaczymy oczywiście 76, jako że nic nie zmieniliśmy od poprzedniego akapitu.

Muszę jeszcze powiedzieć, że manewr z pobraniem adresu pola w obiekcie powiedzie się tylko wtedy, jeżeli to pole jest publiczne. W innej sytuacji wyrażenie `Foo.nSkladnik` zostanie odrzucone przez kompilator.
Zawsze można aczkolwiek pobierać adresy pól wewnątrz klasy (np. w jej metodach) oraz w funkcjach i klasach zaprzyjaźnionych. Te obszary kodu mają bowiem dostęp do wszystkich składników - także niepublicznych i mogą z nimi robić cokolwiek: na przykład pobierać ich adresy w pamięci.

Wskaźnik do pola wewnątrz klasy

Kontynuujemy naszą zabawę. Teraz weźmy pod lupę trochę inną klasę, z którą już mnóstwo razy się spotykaliśmy - wektor:

```
struct VECTOR3 { float x, y, z; };
```

Formalnie jest to struktura, ale jak wiemy, w C++ różnica między strukturą a klasą jest drobnostką i sprowadza się do domyślnej widoczności składników. Dla słówka `struct` jest to `public`, więc nasze trzy pola są tu publiczne bez konieczności jawnego określania tego faktu.

Mając klasę (albo strukturę - jak kto woli) z trzema polami możemy ją naturalnie instancjować (czyli stworzyć jej obiekt):

```
VECTOR3 Wektor;
```

Następnie możemy też pobrać adres jej pola - którejś ze współrzędnych:

```
float* pfX = &Wektor.x;
```

Miejsce pola w definicji klasy

Przyjrzyjmy się jednak definicji klasy. Mamy w niej trzy takie same pola, następujące jedno po drugim. Pierwsze (`x`), drugie (`y`) i trzecie (`z`)… Jeżeli ci to pomoże, możesz nawet wyobrazić sobie nasz wektor jako trójelementową tablicę, w której nazwaliśmy poszczególne elementy (pola). Zamiast odwoływać się do nich poprzez indeksy, potrafimy posłużyć się ich nazwami (`x`, `y`, `z`).
Porównanie z tablicą jest jednak całkiem trafne - choćby dlatego, że nasze pola są ułożone w pamięci w kolejności występowania w definicji klasy. Najpierw mamy więc `x`, potem `y`, a dalej `z`. Polu `x` możemy więc przypisać „indeks" 0, `y` - 1, a dla `z` „indeks" 2.

Słowo 'indeks' biorę tu w cudzysłów, bo jest to tylko takie pojęcie pomocnicze. Wiesz, że w przypadku tablic indeksy są ostatecznie zamieniane na wskaźniki w ten sposób, że do adresu całej tablicy (czyli jej pierwszego elementu) dodawany jest indeks:

```
int* aTablica[5];

// te dwie linijki są równoważne
aTablica[3] = 12;
*(aTablica + 3) = 12;
```

Dodawanie, jakie występuje w ostatnim wierwszu, nie jest dosłownym dodaniem trzech bajtów do wskaźnika `aTablica`, jest przesunięciem się o trzy elementy. Właściwie więc kompilator zamienia to na:

```
aTablica + 3 * sizeof(int)
```

i tak oto uzyskuje adres czwartego elementu tablicy (o indeksie 3). Spójrzmy na dodawane wyrażenie:

```
3 * sizeof(int)
```

Określa ono **przesunięcie** (ang. *offset*) elementu tablicy o indeksie 3 względem jej początku. Znając tę wartość kompilator oraz adres pierwszego elementu tablicy, kompilator może wyliczyć pozycję w pamięci dla elementu numer 3.

Dlaczego jednak o tym mówię?… Otóż bardzo podobna operacja zachodzi przy odwoływaniu się do pola w obiekcie klasy (struktury). Kiedy bowiem odnosimy się jakiegoś pola w ten oto sposób:

```
Wektor.y
```

to po pierwsze, kompilator zamienia to wyrażenie tak, aby posługiwać się wskaźnikami, bo to jest jego „mową ojczystą":

```
(&Wektor)->y
```

Następnie stosuje on ten sam mechanizm, co dla elementów tablic. Oblicza więc adres pola (tutaj `y`) według schematu:

&Wektor + *offset_pola_y*

W tym przypadku sprawa nie jest aczkolwiek taka prosta, bo definicja klasy może zawierać pola wielu różnych typów o róznych rozmiarach. Offset nie będzie więc mógł być wyliczany tak, jak to się dzieje dla elementu tablicy. On musi być znany już wcześniej… Skąd?

Z definicji klasy! Określając naszą klasę w ten sposób:

```
struct VECTOR3 { float x, y, z; };
```

zdefiniowaliśmy nie tylko jej składniki, ale też kolejność pól w pamięci. Oczywiście nie musimy podawać dokładnych liczb, precyzujących położenie np. pola z względem obiektu klasy VECTOR3. Tym zajmie się już sam kompilator: przeanalizuje całą definicję i dla każdego pola wyliczy sobie oraz zapisze gdzieś odpowiednie przesunięcie.

I tę właśnie liczbę nazywamy **wskaźnikiem na pole klasy**:

Wskaźnik na pole klasy jest określeniem **miejsca w pamięci**, jakie zajmuje pole danej klasy, **względem początku obiektu w pamięci**.

W przeciwnieństwie do zwykłego wskaźnika **nie jest to więc liczba bezwzględna.** Nie mówi nam, że tu-i-tu znajduje się takie-a-takie pole. Ona tylko informuje, o ile bajtów należy się przesunąć, poczynając od adresu obiektu, a znaleźć w pamięci konkretne pole w tym obiekcie.

Może jeszcze lepiej zrozumiesz to na przykładzie kodu. Jeżeli stworzymy sobie obiekt (statycznie, dynamicznie - nieważne) - na przykład obiekt naszego wektora:

```
VECTOR3* pWektor = new VECTOR3;
```

i pobierzemy adres jego pola - na przykład adres pola y **w tym obiekcie**:

```
int* pnY = &pWektor->y;
```

to różnica wartości obu wskaźników (adresów) - na obiekt i na jego pole:

pnY - pWektor

bedzie niczym innym, jak właśnie **offsetem** tegoż pola, czyli jego **miejscem w definicji klasy**! To jest ten rodzaj wskaźników C++, jakim się chcemy tutaj zająć.

Pobieranie wskaźnika

Zauważmy, że offset pola jest wartością globalną dla całej klasy. Każdy bowiem obiekt ma tak samo rozmieszczone w pamięci pola. Nie jest tak, że wśród kilku obiektów naszej klasy VECTOR3 jeden ma pola ułożone w kolejności x, y, z, drugi - y, z, x, trzeci - z, y, x, itp. O nie, tak nie jest: wszystkie pola są poukładane dokładnie w **takiej kolejności**, jaką ustaliliśmy w **definicji klasy**, a ich umiejscowienie jest **dla każdego obiektu identyczne**.

Uzyskanie offsetu danego pola, czyli wskaźnika na pole klasy, może więc odbywać się bez konieczności posiadania obiektu. Wystarczy tylko podać, o jaką klasę i o jakie pole nam chodzi, np.:

```
&VECTOR3::y
```

Powyższe wyrażenie zwróci nam wskaźnik na pole y w klasie VECTOR3. Powtarzam jeszcze raz (abyś dobrze to zrozumiał), iż będzie to ilość bajtów, o jaką należy się przesunąć poczynając od adresu jakiegoś obiektu klasy VECTOR3, aby natrafić na pole y tegoż obiektu. Jeżeli jest to dla ciebie zbyt trudne, to możesz mysleć o tym wskaźniku jako o „indeksie” pola y w klasie VECTOR3.

Deklaracja wskaźnika na pole klasy

No dobrze, pobranie wskaźnika to jedno, ale jego zapisanie i wykorzystanie to zupełnie coś innego. Najpierw więc dowiedzmy się, jak można zachować wartość uzyskaną wyrażeniem `&VECTOR3::y` do późniejszego wykorzystania.

Być może domyślasz się, że będzie potrzebowali specjalnej zmiennej typu wskaźnikowego - czyli wskaźnika na pole klasy. Aby go zadeklarować, musimy przypomnieć sobie, czym charakteryzują się wskaźniki w C++.
Nie jest to trudne. Każdy wskaźnik ma swój **typ**: w przypadku wskaźników na zmienne był to po prostu typ docelowej zmiennej. Dla wskaźników na funkcje sprawa była bardziej skomplikowana, niemniej też miały one swoje typy.

Podobnie jest ze wskaźnikami na składowe klasy. Każdy z nich ma przypisaną **klasę**, na które składniki pokazuje - dotyczy to zarówno odniesień do pól, którymi zajmujemy się teraz, jak i do metod, które poznamy za chwilę.
Oprócz tego wskaźnik na pole klasy musi też znać **typ docelowego pola**, czyli wiedzieć, jaki rodzaj danych jest w nim przechowywany.

Czy wiemy to wszystko? Tak. Wiemy, że naszą klasą jest `VECTOR3`. Pamiętamy też, że jej wszystkie pola zadeklarowaliśmy jako `float`. Korzystając z tej informacji, możemy zadeklarować **wskaźnik na pola typu `float` w klasie `VECTOR3`**:

```
float VECTOR3::*p2mfWspolrzedna;
```

Huh, co za zakręcona deklaracja… Gdzie tu jest w ogóle nazwa tej zmiennej?…
Spokojnie, nie jest to aż takie straszne - to tylko tak wygląda :) Nasz wskaźnik nazywa się oczywiście `p2mfWspolrzedna`[116], zaś niezbyt przyjazna forma deklaracji stanie się jaśniejsza, jeżeli popatrzymy na jej ogólną składnię:

```
typ klasa::*wskaźnik;
```

Co to jest? Otóż jest to deklaracja *`wskaźnika`*, pokazującego na *`pola`* podanego *`typu`*, znajdujące się we wnętrzu określonej *`klasy`*. Nic prostrzego, prawda? ;-)

Teraz, kiedy mamy już zmienną odpowiedniego typu wskaźnikowego, możemy przypisać jej względny adres pola `y` w klasie `VECTOR3`:

```
p2mfWspolrzedna = &VECTOR3::y;
```

Pamiętajmy, że w ten sposób nie pokazujemy na konkretną współrzędną Y (pole `y`) w konkretnym wektorze (obiekcie `VECTOR3`), lecz na miejsce pola w definicji klasy. Pojedynczo taki wskaźnik nie jest więc użyteczny, bo jego wartośc nabiera znaczenia dopiero w momencie zastosowania jej dla konkretnego obiektu. Jak to zrobić - zobaczymy w następnym akapicie.

Zwróćmy jeszcze uwage, że `y` nie jest jedynym polem typu `float` w klasie `VECTOR3`. Z równym powodzeniem możemy pokazywać naszym wskaźnikiem także na pozostałe:

```
p2mfWspolrzedna = &VECTOR3::x;
p2mfWspolrzedna = &VECTOR3::z;
```

[116] `p2mf` to skrót od 'pointer-to-member float'.

Warunkiem jest jednak, aby **pole było publiczne**. W przeciwnym wypadku wyrażenie `klasa::pole` byłoby nielegalne (poza klasą) i nie możnaby zastosować wobec niego operatora `&`.

Użycie wskaźnika

Wskaźnik na pole klasy jest **adresem względnym**, offsetem. Aby skorzystać z niego praktycznie, musimy posiadać jakiś obiekt; kompilator będzie dzięki temu wiedział, gdzie się dany obiekt zaczyna w pamięci. Posiadając dodatkowo offset pola w definicj klasy, będziemy mogli odwoływać się do tego pola **w tym konkretnym obiekcie**.

A zatem do dzieła. Stwórzmy sobie obiekt naszej klasy:

```
VECTOR3 Wektor;
```

Potem zadeklarujmy wskaźnik na i ustawmy go na jedno z trzech pól klasy `VECTOR3`:

```
float VECTOR3::*p2mfPole = &VECTOR3::x;
```

Teraz przy pomocy tego wskaźnika możemy odwołac się do tego pola w naszym obiekcie. Jak? O tak:

```
Wektor.*p2mfPole = 12;     // wpisanie liczby do pola obiektu Wektor,
                           // na które pokazuje wskaźnik p2mfPole
```

Cała zabawa polega tu na tym, że `p2mfPole` może pokazywać na dowolne z trzech pól klasy `VECTOR3` - `x`, `y` lub `z`. Przy pomocy wskaźnika możemy jednak do każdego z nich odwoływać się w ten sam sposób.

Co nam to daje? Mniej więcej to samo, co w przypadku zwykłych wskaźników. Wskaźnik na pole klasy możemy przekazać i wykorzystać gdzie indziej. W tym przypadku potrzebujemy aczkolwiek jeszcze jednej danej: obiektu naszej klasy, w kontekście którego użyjemy wskaźnika.
Może czas na jakiś konkretny przykład. Wyobraźmy sobie funkcję, która zeruje jedną współrzędną tablicy wektorów. Teraz możemy ją napisać:

```
void WyzerujWspolrzedna(VECTOR3 aTablica[], unsigned uRozmiar,
                        float VECTOR3::*p2mfWspolrzedna)
{
   for (unsigned i = 0; i < uRozmiar; ++i)
         aTablica[i].*p2mfWspolrzedna = 0;
}
```

W zależności od tego, jak ją wywołamy:

```
VECTOR3 aWektory[50];

WyzerujWspolrzedna (aWektory, 50, &VECTOR3::x);
WyzerujWspolrzedna (aWektory, 50, &VECTOR3::y);
WyzerujWspolrzedna (aWektory, 50, &VECTOR3::z);
```

spowoduje ona wyzerowanie różnych współrzędnych wektorów w podanej tablicy.

Wskaźnik na pole klasy możemy też wykorzystać, gdy na samym obiekcie operujemy także przy pomocy wskaźnika (tym razem zwykłego, na obiekt). Stosujemy wtedy aczkolwiek inną składnię:

```
// deklaracja i inicjalizacja obu wskaźników - na obiekt i pole klasy
```

```
VECTOR3* pWektor = new VECTOR3;
float VECTOR3::p2mfPole = &VECTOR3::z;

// zapisanie wartości do pola z obiektu *pWektor przy pomocy wskaźników
pWektor->*p2mfPole = 42;
```

Jak widać, w kontekście wskaźników na składowe operatory `.*` i `->*` są dokładnymi odpowiednikami operatorów wyłuskania `.` i `->`. Tych drugim używamy jednak wtedy, gdy odwołujemy się do składników obiektu poprzez ich nazwy, natomiast tych pierwszych - jeśli posługujemy się wskaźnikami do składowych.

Operator `->*`, podobnie jak `->`, może być przeciążony. Z kolei `.*`, tak samo jak `.` - nie.

Wskaźnik na metodę klasy

Normalne wskaźniki mogą też pokazywać na kod, czyli funkcje. Obiektowym odpowiednikiem tego faktu są wskaźniki do metod klasy. Zajmiemy się nimi w tej sekcji.

Wskaźnik do statycznej metody klasy

Zwyczajny wskaźnik do funkcji globalnej deklarujemy np. tak:

```
int (*pfnFunkcja)(float);
```

Przypominam, że aby odczytać deklarację funkcji pasujących do tego wskaźnika, wystarczy usunąć gwiazdkę oraz nawiasy otaczające jego nazwę. Tutaj więc możemy do wskaźnika `pfnFunkcja` przypisać adresy wszystkich funkcji globalnych, które przyjmują jeden parametr typu `float` i zwracają liczbę typu `int`:

```
int Foo(float)        { /* ... */ }

// ...

pfnFunkcja = Foo;     // albo pfnFunkcja = &Foo;
```

Jednak nie tylko funkcje globalne mogą być wskazywane przez takie wskaźniki.

Wskaźniki do zwykłych funkcji potrafią też pokazywać na **statyczne metody klas**.

Nietrudno to wyjaśnić. Takie metody to tak naprawdę funkcje globalne o nieco zmienionym zasięgu i notacji wywołania. Najważniejsze, że nie posiadają one ukrytego parametru - wskaźnika `this` - ponieważ ich wywołanie nie wymaga obecności żadnego obiektu klasy. Nie korzystają one więc z konwencji wywołania *thiscall* (właściwej metodom niestatycznym), a zatem możemy zadeklarować zwykłe wskaźniki, które będą nań pokazywać.
Warunkiem jest jednak to, aby metoda statyczna była zadeklarowana jako `public`. W przeciwnym razie wyrażenie *nazwa_klasy*`::`*nazwa_metody* nie będzie legalne.

Podobne uwagi można poczynić dla statycznych pól, na które można pokazywać przy pomocy zwykłych wskaźników na zmienne.

Wskaźnik do niestatycznej metody klasy

A jak jest z metodami niestatycznymi? Czy na nie też możemy pokazywać zwykłymi wskaźnikami?…

Niestety nie. Fakt ten może się wydać zaskakujący, ale można go wyjaśnić nawet na kilka sposobów.
Po pierwsze: wspomniałem już, że metody niestatyczne korzystają ze specjalnej konwencji *thiscall*. Oprócz normalnych parametrów musza one bowiem dostać obiekt, który w ich wnętrzu będzie reprezentowany przez wskaźnik `this`. C++ nie pozwala na zadeklarowanie funkcji używających konwencji *thiscall* - nie bardzo wiadomo, jak taka deklaracja miałaby wyglądać[117].
Po drugie: metody niestatyczne potrzebują wskaźnika `this`. Gdyby dopuścić do sytuacji, w której wskaźniki na funkcje mogą pokazywać na metody, wówczas trzebaby było zapewnić jakoś dostarczenie tego wskaźnika `this` (czyli obiektu, na rzecz którego metoda jest wywoływana). Jak? Poprzez dodatkowy parametr?… Wtedy mielibyśmy koszmarną nieścisłość składni: deklaracje wskaźników do funkcji **nie zgadzałyby się** z prototypami pasujących do nich metod.

Nawet jeśli nie bardzo zrozumiałeś te argumenty, musisz przyjąć, że na niestatyczne metody klasy nie pokazujemy zwykłymi wskaźnikami do funkcji. Zamiast tego wykorzystujemy drugi rodzaj wskaźników na składowe klasy.

Wykorzystanie wskaźników na metody

Mam tu na myśli **wskaźniki na metody klas**.

Wskaźnik do metody w klasie (ang. *pointer-to-member function*) określa **miejsce deklaracji tej metody** w **definicji klasy**.

Widać tu analogie ze wskaźnikami do pól klasy. Tutaj także określamy umiejscowienie danej metody względem…

No właśnie - względem czego?! W przypadku pól mogliśmy jeszcze mówić, że wskaźnik jest określeniem przesunięcia (offsetu), który pozwala znaleźć pole danego obiektu, gdy mamy adres początku tegoż obiektu. Ale przecież metody nie podlegają tym zasadom. Dla **wszystkich obiektów** mamy przecież **jeden zestaw metod**. Jak więc można mówić o tym, że wskaźniki na nie działają w ten sam sposób?…
Ekhm, tego raczej nie powiedziałem. Wskaźniki te **mogą** działać ten sam sposób, czyli być adresami względnymi. Mogą one także być adresami bezwzględnymi (w sumie - dlaczego nie? Przecież metody to też funkcje), a nawet indeksami jakiejś wewnętrznej tablicy czy jeszcze dziwniejszymi liczbami z gatunku identyfikatorów-uchwytów. Tak naprawdę **nie powinno nas to interesować**, gdyż jest to wewnętrzna sprawa kompilatora. Dla nas wskaźniki te pokazują po prostu na jakąś metodę wewnątrz danej klasy. Jak to robią - to już nie nasze zmartwienie.

Deklaracja wskaźnika

Spójrzmy lepiej na jakiś przykład. Weźmy taką oto klasę:

```
class CNumber
{
   private:
         float m_fLiczba;

   public:
         // konstruktor
         CNumber(float m_fLiczba = 0.0f) : m_fLiczba(fLiczba) { }
```

[117] Zauważmy, że deklaracja metody „wyjęta” z klasy i umieszczona poza nią automatycznie stanie się funkcją globalną. Nie trzeba dokonywać żadnych zmian w jej prototypie, polegających np. na usunięciu słowa `thiscall`. Takiego słowa kluczowego po prostu nie ma: C++ odróżnia metody od zwykłych funkcji **wyłącznie** po miejscu ich zadeklarowania.

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

            // kilka metod
            float Dodaj(float x)    { return (m_fLiczba += x); }
            float Odejmij(float x)  { return (m_fLiczba -= x); }
            float Pomnoz(float x)   { return (m_fLiczba *= x); }
            float Podziel(float x)  { return (m_fLiczba /= x); }
};
```

Nie jest ona może zbyt mądra - nie ma przeciążonych operatorów i w ogóle wykonuje dość dziwną czynność enkapsulacji typu podstawowego - ale dla naszych celów będzie wystarczająca. Zwróćmy uwagę na jej cztery metody: wszystkie biorą argument typu `float` i takąż liczbę zwracają. Jeżeli chcielibyśmy zadeklarować wskaźnik, mogący pokazywać na te metody, to robimy to w ten sposób[118]:

```
float (CNumber::*p2mfnMetoda)(float);
```

Wskaźnik `p2mfnMetoda` może pokazywać na każdą z tych czterech metod, tj.:

```
float CNumber::Dodaj(float x);
float CNumber::Odejmij(float x);
float CNumber::Pomnoz(float x);
float CNumber::Podziel(float x);
```

Można stąd całkiem łatwo wywnioskować ogólną składnię deklaracji takiego wskaźnika. A więc, dla metody klasy o nagłówku:

```
zwracany_typ nazwa_klasy::nazwa_metody([parametry])
```

deklaracja odpowiadającego jej wskaźnika wygląda tak:

```
zwracany_typ (nazwa_klasy::*nazwa_wskaźnika)([parametry]);
```

Deklaracja wskaźnika na metodę klasy wygląda tak, jak nagłówek tej metody, w którym fraza `nazwa_klasy::nazwa_metody` została zastąpiona przez sekwencję `(nazwa_klasy::*nazwa_wskaźnika)`. Na końcu deklaracji stawiamy oczywiście średnik.

Sposób jest więc bardzo podobny jak przy zwykłych wskaźnikach na funkcje. Ponownie też istotne stają się nawiasy. Gdybyśmy bowiem je opuścili w deklaracji `p2mfnMetoda`, otrzymalibyśmy:

```
float CNumber::*p2mfnMetoda(float);
```

co zostanie zinterpretowane jako:

```
float CNumber::* p2mfnMetoda(float);
```

czyli funkcja biorąca jeden argument `float` i zwracająca wskaźnik do pól typu `float` w klasie `CNumber`. Zatem znowu - zamiast wskaźnika na funkcję otrzymujemy funkcję zwracającą wskaźnik.

Dla wskaźników na metody klas nie ma problemu z umieszczenia słowa kluczowego konwencji wywołania, bo wszystkie metody klas używają domyślnej i jedynie słusznej w

[118] `p2mfn` to skrót od 'pointer-to-member function'.

ich przypadku konwencji *thiscall*. Nie ma możliwości jej zmiany (mam nadzieję, że jest oczywiste, dlaczego…).

Pobranie wskaźnika na metodę klasy

Kiedy mamy już zadeklarowany właściwy wskaźnik, powiążmy go z którąś z metod klasy CNumber. Robimy to w prosty i raczej przewidywalny sposób:

```
p2mfnMetoda = &CNumber::Dodaj;
```

Podobnie jak dla zwykłych funkcji, także i tutaj operator `&` nie jest niezbędny:

```
p2mfnMetoda = CNumber::Odejmij;
```

Znowu też stosuje się tu zasada o publiczności składowych. Jeżeli spróbujemy pobrać wskaźnik na metodę prywatną lub chronioną, to kompilator oczywiście zaprotestuje.

Użycie wskaźnika

Czas wreszcie na akcję. Zobaczmy, jak można wywołać metodę pokazywaną przez wskaźnik:

```
CNumber Liczba = 42;
std::cout << (Liczba.*p2mfnMetoda)(2);
```

Potrzebujemy naturalnie jakiegoś obiektu klasy `CNumber`, aby na jego rzecz wywołać metodę. Tworzymy go więc; dalej znowu korzystamy z operatora `.*`, wywołując przy jego pomocy metodę klasy `CNumber` dla naszego obiektu - przekazujemy jej jednocześnie parametr `2`. Ponieważ po naszej zabawie z przypisywaniem `p2mfnMetoda` pokazywał na metodę `Odejmij()`, na ekranie zobaczylibyśmy:

```
40
```

Zwracam jeszcze uwagę na nawiasy w wywołaniu metody. Tutaj są one **konieczne** (w przeciwieństwie do zwykłych wskaźników na funkcje) - bez nich kompilator uzna linijkę za błędną.

Domyślasz się, że jeśli posiadalibyśmy tylko wskaźnik na obiekt, to do wywołania jego metody posłużylibyśmy się operatorem `->*`. Identycznie jak przy wskaźnikach na pola klasy.

Ciekawostka: wskaźnik do metody obiektu

Zatrzymajmy się na chwilkę… Jeżeli przebrnąłeś przed ten rozdział od początku aż dotąd, to szczerze ci gratuluję. Wskaźniki na składowe nie są bynajmniej łatwą częścią języka - choćby dlatego, że operują dość dziwnymi koncepcjami („miejsce w definicji klasy”…). Co gorsza, czytając o nich jakoś trudno od razu wpaść na sensowne zastosowanie tego mechanizmu.
Wiem, że podobne odczucia mogły ci towarzyszyć przy lekturze opisów wielu innych elementów języka. Później jednak nieczęsto widziałeś zastosowania omawianych wcześniej rzeczy w dalszej częściu kursu, a pewnie sam znalajdowałeś niektóre z nich po odpoczynku od lektury i dłuższym zastanowieniu.

Tutaj muszę cię nieco zmartwić. Wskaźniki na składowe klasy są w praktyce bardzo rzadko używane, bo w zasadzie trudno znaleźć dla nich jakieś użyteczne zastosowanie. To chyba najdobitniejszy przykład językowego wodotrysku - na szczęście C++ nie posiada zbyt wiele takich nadmiarowych udziwnień.

Spróbujemy jednak znaleźć dla nich jakieś zastosowanie… Okazuje się, że jest to możliwe. Wskaźników tych możemy bowiem użyć do symulowania innego rodzaju wskaźników - nieobecnych niestety w C++, ale za to bardzo przydatnych.
Jakie to wskaźniki? Spójrz na poniższą tabelę. Grupuje ona wszystkie znane (i nieznane ;D) w programowaniu strukturalnym i obiektowym rodzaje wskaźników, wraz z ich nazwami w C++:

rodzaj wskaźnika → / ***cel wskaźnika ↓***	***strukturalne***	***obiektowe***		
		na składowe statyczne	***na składowe niestatyczne***	
			w klasach	***w obiektach***
dane	wskaźniki do zmiennych		wskaźniki do pól klasy	wskaźniki do zmiennych
kod	wskaźniki do funkcji		wskaźniki do metod klasy	BRAK

Tabela 19. Różne rodzaje wskaźników

Wynika z niej, że znamy już wszystkie rodzaje wskaźników, jakie posiada w swoim arsenale C++. A co z tymi brakującymi?…

Czym one są?… Otóż są to takie wskaźniki, które potrafią pokazywać **na konkretną metodę w konkretnym obiekcie**. Podobnie jak wskaźniki do pól obiektu, są one samodzielne. Ich użycie nie wymaga więc żadnych dodatkowych informacji: dokonując zwyczajnej dereferencji takiego wskaźnika, wywoływalibyśmy **określoną metodę w odniesieniu do określonego obiektu**. Zupełnie tak, jak dla zwykłych wskaźników do funkcji - tyle tylko, że tutaj nie wywołujemy funkcji globalną, lecz metodę obiektu.

„No dobrze, nie mamy tego rodzaju wskaźników… Ale co z tego? Na pewno są one równie „użyteczne”, jak te co poznaliśmy niedawno!” Otóż wręcz przeciwnie! Tego rodzaju wskaźniki są niezwykle przydatne! Pozwalają one bowiem na implementację **funkcji zwrotnych** (ang. *callback functions*) z zachowaniem pełnej obiektowości programu.

Cóż to są - te funkcje *callback*? Są to takie funkcje, których adresy przekazujemy komuś, aby ten ktoś mógł je dla nas wywołać w odpowiednim momencie. Ten odpowiedni moment to na przykład zajście jakiegoś zdarzenia, na które oczekujemy (wciśnięcie klawisza, wybicie północy na zegarze, itp.) albo chociażby wystąpienie błędu. W każdej tego typu sytuacji nasz program może być o tym natychmiast poinformowany. Bez funkcji zwrotnych musiałby zwykle dokonywać mozolnego odpytywania „ktosia”, aby dowiedzieć się, czy dana okoliczność wystąpiła. To mało efektywne rozwiązanie.
Funkcje *callback* są lepsze. Jednak w C++ tylko funkcje globalne lub statyczne metody klas mogą być takimi funkcjami. Powód jest prosty: jedynie na takie metody możemy pokazywać samodzielnymi wskaźnikami.

A to jest zupełnie niezadowolające w programowaniu obiektowym. Zmusza to przecież do pisania kodu poza klasami programu. W dodatku trzeba jakoś zapewnić sensowną komunikację między tym kodem-*outsiderem*, a obiektową resztą programu. W sumie mamy mnóstwo kłopotów.
Wymyślono rzecz jasna pewien sposób na obejście tego problemu, polegający na wykorzystaniu metod wirtualnych, dziedziczenia i polimorfizmu. Nie jest to jednak idealne rozwiązanie - przynajmniej nie w C++.

Powiedziałem jednak, że nasze świeżo poznane wskaźniki mogą pomóc w poradzeniu sobie z tym problemem. Zobaczmy jak to zrobić.

Bardzo, ale to bardzo odradzam czytanie tych dwóch punktów przy pierwszym kontakcie z tekstem (to zresztą dotyczy prawie wszystkich *Ciekawostek*). Sprawa jest wprawdzie

bardzo ciekawa i niezwykle przydatna, lecz jej zawiłość może cię szybko odstręczyć od wskaźników klasowych - albo nawet od programowania obiektowego, co by było znacznie gorszą katastrofą.

Wskaźnik na metodę obiektu konkretnej klasy

Najpierw zajmijmy się prostszym przypadkiem. Znajdźmy sposób na symulację wskaźnika, za pośrednictwem którego możnaby wywoływać metodę:
- o określonej sygnaturze (nagłówku)
- na rzecz określonego obiektu
- należącego do określonej klasy

Dosyć dużo tych „określeń”… Najlepiej będzie, jeśli popatrzysz na działanie tego wskaźnika. Przypomnij sobie klasę `CNumber`; stwórzmy obiekt tej klasy:

```
CNumber Liczba;
```

Teraz wyobraźmy sobie, że w języku C++ pojawiła się możliwość zadeklarowania wskaźników, o jakie nam chodzi. Niech `p2ofnMetoda` będzie tym pożądanym wskaźnikiem[119]. Wówczas można z nim zrobić coś takiego:

```
// przypisanie wskaźnikowi "adresu metody" Dodaj w obiekcie Liczba
p2ofnMetoda = Liczba.Dodaj;

// wywołanie metody Dodaj() dla obiektu Liczba()
(*p2ofnMetoda)(10);
```

Jak widać, dokonujemy tu zwykłej dereferencji - zupełnie tak, jak w przypadku wskaźników na funkcje globalne. Tym sposobem wywołujemy jednak metodę klasy dla konkretnego obiektu. Ostatnia linijka jest więc równoważna tej:

```
Liczba.Dodaj(10);
```

Zamiast wywołania `obiekt.metoda()` mamy więc `(*wskaźnik_do_metody_obiektu)()`. I o to nam chodzi.

Wróćmy teraz do rzeczywistości. Niestety C++, nie posiada wskaźników na metody obiektów, lecz chcemy przynajmniej częściowo uzupełnić ten brak. Jak to zrobić?… Przyjrzyjmy się temu, co chcemy osiągnąć. Chcemy mianowicie, aby nasz wskaźnik zastępował wywołanie:

```
obiekt.metoda([parametry])
```

w ten sposób:

```
(*wskaźnik)([parametry])
```

Wskaźnik musi więc zawierać informacje zarówno o obiekcie, którego dotyczy metoda, jak i samej metodzie. Jeden wskaźnik?… Nie - dwa:
- pierwszy to wskaźnik na obiekt, na rzecz którego metoda będzie wywoływana
- drugi to wskaźnik na metodę klasy, która ma być wywoływana

Chcąc stworzyć nasz wskaźnik, musimy więc połączyć te dwie dane. Zróbmy to! Najpierw zdefiniujmy sobie jakąś klasę, na której metody będziemy pokazywać:

---

[119] `p2ofn` to skrót od 'pointer to object-function'.

```
class CFoo
{
   public:
         void Metoda(int nParam)
               { std::cout << "Wywolano z " << nParam; }
};
```

Dalej - dodajmy obiekt, który będzie brał udział w wywołaniu:

```
CFoo Foo;
```

Przypomnijmy wreszcie, że chcemy zrobić taki wskaźnik, którego użycie zastapi nam wywołanie:

```
Foo.Metoda();
```

Potrzebujemy do tego wspomnianych dwóch rodzajów wskaźników:
- wskaźnika na obiekty klasy `CFoo`
- wskaźnika na metody klasy `CFoo` biorące `int` i niezwracające wartości

Połączymy oba te wskaźniki w jedną strukturę, dodając przy okazji pomocnicze funkcje - jak konstruktor oraz `operator()`:

```
struct METHODPOINTER
{
   // rzeczone oba wskaźniki
   CFoo* pObject;                        // wskaźnik na obiekt
   void (CFoo::*p2mfnMethod)(int);       // wskaźnik na metodę

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

   // konstruktor
   METHODPOINTER(CFoo* pObj, void (CFoo::*p2mfn)(int))
         : pObject(pObj), p2mfnMethod(p2mfn)            { }

   // operator wywołania funkcji
   void operator() (int nParam)
         { (pObject->*p2mfnMethod(nParam); }
};
```

Teraz możemy już pokazać takim wskaźnikiem na metodę naszego obiektu. Podajemy po prostu zarówno wskaźnik na obiekt, jak i na metodę klasy:

```
METHODPOINTER p2ofnMetoda(&Foo, &CFoo::Metoda);
```

To wprawdzie pewna niedogodność (nie możemy podać po prostu Foo.Metoda, lecz musimy pamiętać nazwę klasy), ale i tak jest to całkiem dobre rozwiązanie. Naszą metodę możemy bowiem wywołać w najprostszy możliwy sposób:

```
p2ofnMetoda (69);    // to samo co Foo.Liczba (69);
```

To właśnie chcieliśmy osiągnać.

Jest to aczkolwiek rozwiązanie dla szczególnego przypadku. A jak wygląda to w przypadku ogólnym?... Mniej więcej w ten sposób:

```
struct WSKAŹNIK
{
   // wskaźniki
```

```
    klasa* pObject;
    zwracany_typ (klasa::*p2mfnMethod)([parametry_formalne]);

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

    // konstruktor
    WSKAŹNIK(klasa* pObj,
             zwracany_typ (klasa::*p2mfn)([parametry_formalne]))
          : pObject(pObj), p2mfnMethod(p2mfn)            { }

    // operator wywołania funkcji
    zwracany_typ operator() ([parametry_formalne])
          { [return] (pObject->*p2mfnMethod([parametry_aktualne]); }
};
```

Niestety, preprocesor na niewiele nam się przyda w tym przypadku. Tego rodzaju struktury musiałbyś wpisywać do kodu samodzielnie.

Wskaźnik na metodę obiektu dowolnej klasy

Nasz *callback* wydaje się działać (bo i działa), ale jego przydatność jest niestety niewielka. Wskaźnik potrafi bowiem pokazywać tylko na metodę w konkretnej klasie, natomiast do zastosowań praktycznych (jak informowanie o zdarzeniach czy błędach) powinien on umieć wskazać na zgodną ustalonym prototypem metodę obiektu **w dowolnej klasie**.
Tak więc niezależnie od tego, czy nasz obiekt byłby klasy `CFoo`, `CVector2D`, `CEllipticTable` czy `CBrokenWindow`, jeśli tylko klasa ta posiada metodę o określonej sygnaturze, to powinno dać się na nią wskazać w konkretnym obiekcie. Dopiero wtedy dostaniemy do ręki wartościowy mechanizm.

Ten mechanizm ma nazwę: *closure*. Trudno to przetłumaczyć na polski (dosłownie jest to 'przymknięcie', 'domknięcie', itp.), więc będziemy posługiwać się dotychczasową nazwą 'wskaźnik na metodę obiektu'.

Czy można go osiągnąć w C++?... Owszem. Wymaga to jednak dość daleko idącego kroku: otóż musimy sobie zdefiniować **uniwersalną klasę bazową**. Z takiej klasy będą dziedziczyć wszystkie inne klasy, których obiekty i ich metody mają być celami tworzonych wskaźników. Taka klasa może być bardzo prosta, nawet pusta:

```
class IObject  { };
```

Można do niej dodać wirtualny destruktor czy inne wspólne dla wszystkich klas składowe, jednak to nie jest tutaj ważne. Grunt, żeby taka klasa była obecna.

Teraz sprecyzujmy problem. Załóżmy, że mamy kilka innych klas, zawierających metody o właściwej dla nas sygnaturze:

```
class CFoo : public IObject
{
   public:
         float Funkcja(int x)    { return x * 0.75f; }
};

class CBar : public IObject
{
   public:
         float Funkcja(int x)    { return x * 1.42f; }
};
```

Zauważmy z IObject. Czego chcemy? Otóż poszukujemy sposobu na zaimplementowanie wskaźnika, który będzie pokazywał na metodę Funkcja() zarówno w obiektach klasy CFoo, jak i CBar. Nawet więcej - chcemy takiego wskaźnika, który pokaże nam na dowolną metodę biorącą int i zwracają float **w dowolnym obiekcie dowolnej klasy** w naszym programie. Mówiłem już, że w praktyce ta „dowolna klasa" musi dziedziczyć po IObject.

Cóż więc zrobić? „Może znowu sięgniemy po dwa wskaźniki - jeden na obiekt, a drugi na metodę klasy…?" Punkt dla ciebie. Faktycznie, tak właśnie zrobimy. Postać naszego wskaźnika nie różni się więc zbytnio od tej z poprzedniego punktu:

```
struct METHODPOINTER
{
   // rzeczone oba wskaźniki
   IObject* pObject;                     // wskaźnik na obiekt
   float (IObject::*p2mfnMethod)(int);   // wskaźnik na metodę

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

   // konstruktor
   METHODPOINTER(IObject* pObj, float (IObject::*p2mfn)(int))
         : pObject(pObj), p2mfnMethod(p2mfn)            { }

   // operator wywołania funkcji
   float operator() (int x)
         { return (pObject->*p2mfnMethod(x); }
};
```

„Chwileczkę… Deklarujemy tutaj wskaźnik na metody klasy IObject, biorące int i zwracające float… Ale przecież IObject nie ma takich metod - ba, u nas nie ma nawet żadnych metod! Takim wskaźnikiem nie pokażemy więc na żadną metodę!"
Bingo, kolejny punkt za uważną lekturę :) Rzeczywiście, taki wskaźnik wydaje się bezużyteczny. Pamiętajmy jednak, że w sumie chcemy pokazywać na **metodę obiektu**, a nie na metodę klasy. Zaś nasze obiekty będą pochodzić od klasy IObject, bo ich własne klasy po IObject dziedziczą. W sumie więc wskaźnikiem na metodę klasy bazowej będziemy pokazywać na metodę klasy pochodnej. To jest poprawne - za chwilę wyjaśnię bliżej, dlaczego.

Najpierw spróbujmy użyć naszego wskaźnika. Stwórzmy więc obiekt którejś z klas:

```
CBar* pBar = new CBar;
```

i ustawmy nasz wskaźnik na metodę Funkcja() w tym obiekcie - tak, jak to robiliśmy dotąd:

```
METHODPOINTER p2ofnMetoda(pBar, &CBar::Funkcja);
```

I jak?… Mamy przykrą niespodziankę. Każdy szanujący się kompilator C++ najpewniej odrzuci tę linijkę, widząc niezgodność typów. Jaką niezgodność?
Pierwszy parametr jest absolutnie w porządku. To znana i lubiana konwersja wskaźnika na obiekt klasy pochodnej (CBar*) do wskaźnika na obiekt klasy bazowej (IObject*). Brak zastrzeżeń nikogo nie dziwi - przecież na tym opiera się cały polimorfizm.
To drugi parametr sprawia problem. Kompilator nie zezwala na zamianę typu:

```
float (CBar::*)(int)
```

na typ:

```
float (IObject::*)(int)
```

Innymi słowy, nie pozwala na konwersję wskaźnik na metodę klasy pochodnej do wskaźnika na metodę klasy bazowej. Jest to uzasadnione: wskaźnik na metodę (ogólnie: na składową) może być bowiem poprawny w klasie pochodnej, natomiast nie zawsze będzie poprawny w klasie bazowej. Obiekt klasy bazowej może być przecież mniejszy, nie zawierać pewnych elementów, wprowadzonych w młodszym pokoleniu. W takim wypadku wskaźnik będzie „strzelał w próżnię", co skończy się błędem ochrony pamięci[120].
Tak mogłoby być, jednak u nas tak nie będzie. Naszego wskaźnika na metodę użyjemy przecież tylko i wyłacznie do wywołania metody obiektu pBar. Klasa obiektu oraz klasa wskaźnika w tym przypadku zgadzają się, są identyczne - to CBar. Nie ma żadnego ryzyka.

Kompilator bynajmniej o tym nie wie i nie należy go wcale za to winić. Musimy sobie po prostu pomóc rzutowaniem:

```
METHODPOINTER p2ofnMetoda(pBar,
                          static_cast<float (IObject::*)(int)>
                          (&CBar::Funkcja));
```

Wiem, że wygląda to okropnie, ale przecież nic nie stoi na przeszkodzie, aby pomóc sobie odpowiednim makrem.

Zresztą, liczy się efekt. Teraz możemy wywołać metodę pBar->Funkcja() w ten prosty sposób:

```
p2ofnMetoda (42);           // to samo co pBar->Funkcja (42);
```

Jest też zupełnie możliwe, aby pokazać naszym wskaźnikiem na analogiczną metodę w obiekcie klasy CFoo:

```
CFoo Foo;
p2ofnMetoda.pObject = &Foo;
p2ofnMetoda.p2mfnMethod = static_cast<float (IObject::*)(int)>
                          (&CFoo::Funkcja));

p2ofnMetoda (14);          // to samo co Foo.Funkcja (14)
```

Zmieniając ustawienie wskaźnika musimy jednak pamiętać, by:

Klasy **docelowego obiektu** oraz **docelowej metody** muszą być identyczne. Inaczej ryzykujemy bład ochrony pamięci.

Zaprezentowane rozwiązanie może nie jest szczególnie eleganckie, ale wystarczające. Nie zmienia to jednak faktu, że wbudowana obsługa wskaźników na metody obiektów w C++ byłaby wielce pożądana.

Nieco lepszą implementację wskaźników tego rodzaju, korzystającą m.in. z szablonów, możesz znaleźć w moim artykule *Wskaźnik na metodę obiektu*.

***

[120] Konwersja w drugą stronę (ze wskaźnika na składową klasy bazowej do wskaźnika na składową klasy pochodnej) jest z kolei zawsze możliwa. Jest tak dlatego, że klasa pochodna nie może usunąć żadnego składnika klasy bazowej, lecz co najwyżej rozszerzyć ich zbiór. Wskaźnik będzie więc zawsze poprawny.

Czy masz już dość? :) Myślę, że tak. Wskaźniki na składowe klas (czy też obiektów) to nie jest najłatwiejsza część OOPu w C++ - śmiem twierdzić, że wręcz przeciwnie. Mamy ją już jednak za sobą.

Jeżeli aczkolwiek chciałbyś się dowiedzieć na ten temat nieco więcej (także o zwykłych wskaźnikach na funkcje), to polecam świetną witrynę The Function Pointer Tutorials.

W ten sposób poznaliśmy też całą ofertę narzędzi języka C++ w zakresie programowania obiektowego. Możemy sobie pogratulować.

Podsumowanie

Ten długi rozdział był poświęcony kilku specyficznym dla C++ zagadnieniom programowania obiektowego. Zdecydowana większość z nich ma na celu poprawienie wygody, czasem efektywności i „naturalności" kodowania.
Cóż więc zdążyliśmy omówić?…

Na początek poznaliśmy zagadnienie przyjaźni między klasami a funkcjami i innymi klasami. Zobaczyłeś, że jest to prosty sposób na zezwolenie pewnym ściśle określonym fragmentom kodu na dostęp do niepublicznych składowych jakiejś klasy.
Dalej przyjrzeliśmy się bliżej konstruktorom klas. Poznaliśmy ich listy inicjalizacyjne, rolę w kopiowaniu obiektów oraz niejawnych konwersjach między typami.
Następnie dowiedzieliśmy się (prawie) wszystkiego na temat bardzo przydatnego udogodnienia programistycznego: przeciążania operatorów. Przy okazji powtórzyliśmy sobie wiadomości na temat wszystkich operatorów języka C++.
Wreszcie, odważniejsi spośród czytelników zapoznali się także ze specyficznym rodzajem wskaźników: wskaźnikami na składniki klasy.

Następny rozdział będzie natomiast poświęcony niezwykle istotnemu mechanizmowi wyjątków.

Pytania i zadania

Być może zaprezentowane w tym rozdziale techniki służą tylko wygodzie programisty, ale nie zwalnia to kodera z ich dokładnej znajomości. Odpowiedz więc na powyższe pytania i wykonaj ćwiczenia.

Pytania

1. Jakie specjalne uprawnienia ma przyjaciel klasy? Co może być takim przyjacielem?
2. W jaki sposób deklarujemy zaprzyjaźnioną funkcję?
3. Co oznacza deklaracja przyjaźni z klasą?
4. Jak można sprawić, aby dwie klasy przyjaźniły się z wzajemnością?
5. Co to jest konstruktor domyślny? Jakie są korzyści klasy z jego posiadania?
6. Czym jest inicjalizacja? Kiedy i jak przebiega?
7. Do czego służy lista inicjalizacyjna konstruktora?
8. Kiedy konieczny jest konstruktor kopiujący?
9. W jaki sposób możemy definiować niejawne konwersje?
10. Co powoduje słowo kluczowe `explicit` w deklaracji konstruktora?
11. Kiedy konstruktor konwertujący jest jednocześnie domyślnym?
12. Wymień podstawowe cechy operatorów w języku programowania.
13. Jakie rodzaje operatorów posiada C++?
14. Na czym polega przeciążenie operatora?
15. Jaki status mogą posiadać funkcje operatorowe? Czym się one różnią?

16. Jak można skorzystać z niejawnych konwersji, pisząc przeciążone wersje operatorów binarnych?
17. Które operatory mogą być przeciążane wyłącznie jako niestatyczne metody klas?
18. Kiedy konieczne jest zdefiniowanie własnego operatora przypisania?
19. Ile argumentów ma operator wywołania funkcji?
20. O czym należy pamiętać, przeciążając operatory?
21. O czym informuje wskaźnik do składowej klasy?
22. Jakim wskaźnikiem pokazujemy na pole w obiekcie, a jakim na pole w klasie?
23. Czy zwykłym wskaźnikiem do funkcji możemy pokazać na metodę obiektu?

Ćwiczenia

1. Zdefiniuj dwie klasy, które będą ze sobą wzajemnie zaprzyjaźnione.
2. Przejrzyj definicje klas z poprzednich rozdziałów i popatrz na ich konstruktory. W których przypadkach możnaby użyć w nich list inicjalizacyjnych?
3. Do klas `CRational` i `CComplex` dodaj operatory niejawnych konwersji na typ `bool`. Co dzięki temu zyskałeś?
4. (**Trudniejsze**) Wzbogać wspomniane klasy także o operatory dodawania, odejmowania i dzielenia (tylko `CRational`) oraz o odpowiadające im operatory złożonego przypisania i in/dekrementacji.
5. Napisz funktor obliczający największą z podawanych mu liczb typu `float`. Niech stosuje on ten sam interfejs i sposób działania, co klasa `CAverageFunctor`.