Wstęp do programowania obiektowego w języku C++

Wstęp

Programowanie obiektowe (ang. object-oriented programming [skrót OOP]) znacząco różni się od programowania proceduralnego, gdzie dane i procedury nie są ze sobą formalnie powiązane, a programista sam musi dbać o zachowaniu porządku. Programowanie obiektowe znacząco ułatwia to zadanie pozwalając twórcy na zdefiniowanie obiektów łączących dane (nazywane polami) z pewnymi interakcjami i zachowaniami (zwanymi metodami).

Program komputerowy zazwyczaj jest zbiorem wielu obiektów będących w interakcji między sobą. Takie podejście jest bardziej intuicyjne dla człowieka, gdyż w rzeczywistości otaczający nas świat jest de facto zbiorem obiektów posiadających pewne swoje cechy (pola) potrafiących wykonywać pewne zachowania (metody).

Intuicyjnie wiemy czym jest obiekt w rzeczywistości. Jest nim praktycznie wszystko co nas otacza - stół, lampa, budynek, samochód. Każdy z obiektów posiada swoje cechy - lampa może być włączona bądź wyłączona, samochód jest określonego koloru, stół może mieć jedną nogę, cztery albo osiem, a budynek jest czyjąś własnością, zatem posiada właściciela.

Obiekty mogą posiadać pewne zachowania. Przełącznik obiektu lampa świata rzeczywistego w świecie wirtualnym będzie miał dwie odpowiadające funkcje - włącz() i wyłącz(). Choć na pierwszy rzut oka stół może wydawać się mało interaktywny, przenosząc sklep meblarski do świata wirtualnego w C++ możemy nadać mu metodę przesuń(), pozwalającą przesunąć ten konkretny obiekt o zadane wartości przemieszczenia. Tym sposobem naszym obiektom możemy przypisać związane z nimi interakcje.

Można by więc pomyśleć, że programowanie obiektowe służy do łatwiejszego przenoszenia otaczającego nas świata do rzeczywistości wirtualnej. Jednak jest to dużo mocniejsze narzędzie. Czy obiekt w C++ musi być odzwierciedleniem materialnego obiektu? Nie!

Wyobraź sobie, że tworzysz grę komputerową. Gra posiada stan, czyli przykładowo menu, rozgrywka, pauza, ustawienia oraz aktualny licznik punktów. Będąc w menu, punkty wynoszą zero i zwiększają się podczas rozgrywki. Co stoi na przeszkodzie, by utworzyć obiekt klasy gra posiadający pola stan_gry oraz licznik_punktow? Absolutnie nie ma przeszkód. Możesz również utworzyć metody przypisane do obiektu takie jak zmien_stan(), wlacz_pauze(), zwieksz_liczbe_punktow(). Zauważ, że w rzeczywistości nie istnieje fizyczny przedmiot będący odpowiednikiem naszego obiektu. Oznacza to, że obiekty w programowaniu obiektowym możesz wykorzystywać do tworzenia wirtualnych rzeczy w języku C++. Gdy nabierzesz już doświadczenia, przekonasz się, że programy bardzo często składają się w głównej mierze ze stricte wirtualnych obiektów.

Myślę, że ten wstęp rozjaśnił Ci czym są obiekty w C++. Z tematyką obiektów ściśle związane są klasy, ale czym są klasy i jak tworzy się obiekty w C++ przeczytasz już w kolejnej części.

Klasy i obiekty

Skoro wiemy już czym jest obiekt, chcielibyśmy stworzyć jakiś w C++. Jednak jak opisać dany obiekt? Czy tworząc kolejny obiekt tego samego typu, musimy definiować go na nowo?

Dotarliśmy właśnie do kwintesencji klasy. Klasa jest wzorcem, według którego tworzone będą nowe obiekty, które nazywane są instancjami tejże klasy. Programując zgodnie z paradygmatem programowania obiektowego, będziesz pracował na definicjach klas. Same obiekty będą pamiętały swoje dane oraz wykonywały swoje metody.

Podsumowując, każdy obiekt należy do pewnej klasy. Obiekty tej samej klasy posiadają ten sam zestaw metod i pól, jednak ich wartości mogą być zupełnie różne. Myślę, że ten wstęp teorytyczny rozjaśnił Ci mniej więcej z czym mamy tutaj do czynienia i jesteś gotowy zobaczyć programowanie obiektowe w akcji. Przejdźmy w końcu do kodu! :)

```
class NazwaKlasy {

public:
    int pole;

    void metoda();
};
```

Widzimy tutaj słowo kluczowe class, po którym następuje nazwa klasy. W naszej definicji znajduje się jedno pole typu int oraz jedna metoda nie zwracająca żadnej wartości. Nasze pola i metody są publiczne, co pozwala na dostęp do nich z zewnątrz klasy, jednak ten temat będzie dalej rozwinięty przy okazji kontroli dostępu. Zauważ bardzo istotny drobiazg powodujący powstawanie częstego błędu - po klamrze zamykającej definicję klasy następuje średnik!

Zdefiniowaliśmy zatem klasę, ale to nie spowodowało utworzenia żadnego obiektu. Jak to zatem zrobić?

```
NazwaKlasy obiekt;
```

Bardzo intuicyjne, prawda? Zupełnie jakbyśmy tworzyli nową zmienną. W rzeczywistości praktycznie tak właśnie się dzieje - nowa klasa staje się nowym typem danych. Możemy teraz dla przykładu tworzyć funkcje zwracające obiekt:
`NazwaKlasy funkcja_zwracajaca_obiekt(NazwaKlasy obiekt_argument);`

Mamy zatem obiekt, ale jak dostać się do jego pól i metod? Służy do tego operator wyłuskania i zapisujemy go jako kropka (.).

```
obiekt.pole = 2;
obiekt.metoda();
```

Możemy również utworzyć wskaźnik na nasz obiekt, a także dynamicznie alokować pamięć dla nowej instancji. Robimy to w następujący sposób:

```
NazwaKlasy *wsk_na_obiekt = new NazwaKlasy;
```

W przypadku wskaźnika, dostęp do pól i metod otrzymujemy przez strzałkę (->).

```
wks_na_obiekt->pole = 0;
wks_na_obiekt->metoda();
```

Wiemy już jak wywołać metody i uzyskać dostęp do pól. Jednak w naszym przypadku metoda metoda() nie jest zdefiniowana. Gdzie zatem to robić i jak? Rozważmy poniższy przykład:

```
#include <iostream>

using namespace std;

class Kwadrat {
public:
    int dlugosc_boku;

    int licz_pole();
};

int Kwadrat::licz_pole()
{
    return dlugosc_boku * dlugosc_boku;
}

int main()
{
    Kwadrat moj_kwadrat;
    moj_kwadrat.dlugosc_boku = 5;
    cout << "Pole kwadratu wynosi: " << moj_kwadrat.licz_pole();
    return 0;
}
```

Widzimy zatem jak należy definiować metodę danej klasy. W ciele klasy mamy jedynie deklarację funkcji licz_pole(), natomiast poza definicją klasy następuje definicja metody. W praktyce przy podejściu programowaniu obiektowego tworzymy parę plików dla każdej klasy i stosujemy kompilację wielomodułową. W pliku nagłówkowym tworzymy definicję klasy wraz z deklaracjami metod, natomiast ich właściwe definicje umieszczamy w pliku cpp zgodnie ze wzorem

```
void NazwaKlasy::metoda()
{
    //ciało funkcji
}
```

Zauważmy również, że w definicji funkcji nie wywołujemy operatora wyłuskania (.) aby dostać się do pól klasy. Jest to jednak możliwe za pomocą wskaźnika this.

```
void NazwaKlasy::metoda()
{
    this->pole = 4;
}
```

Na pewno zastanawiasz się czym jest magiczny operator public przewijający się już od pierwszego przykładu. Jest to specyfikator dostępu. W C++ wyróżniamy trzy rodzaje:

- Private (prywatny): do tych elementów klasy mają dostęp jedynie metody tej klasy i jej funkcje zaprzyjaźnione.

- Protected (chroniony): dostępne z wewnątrz klasy oraz klas pochodnych i funkcji zaprzyjaźnionych. Klasy pochodne wyjaśnimy w następnym rozdziale dotyczącym dziedziczenia.

- Public (publiczne): dostępne dla każdego w całym programie.

Przykład kodu:

```
class KlasaPrywatna {
private:
    int pole_prywatne;
protected:
    int pole_chronione;
public:
    int pole_publiczne;

private:
    void metoda_prywatna();

public:
    void metoda_publiczna();
};
```

Klasa ta posiada pola i metody prywatne oraz chronione.

```
#include <iostream>

using namespace std;

int main()
{
    KlasaPrywatna moj_obiekt;

    //BŁĄD:
    moj_obiekt.pole_prywatne = 3;
    moj_obiekt.pole_chronione = 0;

    //POPRAWNE:
    moj_obiekt.pole_publiczne = 0;
}
```

Podobnie z metodami. Poniższy przykład kodu nie zadziała:

```
#include <iostream>

using namespace std;

int main()
{
    KlasaPrywatna moj_obiekt;

    moj_obiekt.metoda_prywatna();
}
```

Metodę prywatną możemy wywołać jedynie ze środka klasy, czyli z innej metody tej klasy.

```
void KlasaPrywatna::metoda_publiczna()
{
    metoda_prywatna();
}
```

Takie użycie jest prawidłowe i kompilator nie powinien zgłaszać błędu.

Możesz zadać pytanie - po co ukrywać pewne pola i metody? Jest to jedne z zagadnień programowania obiektowego i nosi nazwę hermetyzacja lub enkpasulacja (ang. encapsulation). Jej głównym celem jest ochrona i walidacja pewnych danych.

```
class Pracownik {
public:
    string nazwisko;
};
```

Dla przykładu, tworząc klasę Pracownik, mającą pole nazwisko, w ten sposób możemy edytować pola obiektów tej klasy dowolnie, pod warunkiem, że nazwisko będzie dowolnym łańcuchem znaków. Wiemy jednak, że formalnie nazwisko może składać się jedynie z liter i myślnika, więc ktoś, kto kiedyś będzie używał naszej klasy może nie zadbać o poprawną walidację danych wejściowych i przypisać tej zmiennej ciąg odpowiadający za numer telefonu. Z punktu widzenia C++ będzie to poprawne, jednak dla systemu informatycznego może okazać się fatalne w skutkach. Aby temu zapobiec, należałoby pole nazwisko ukryć i dodać dwie funkcje - jedna zwracająca wartość danego pola (get_nazwisko()) oraz drugą, pozwalającą na ustawienie wartości zmiennej po wcześniejszym sprawdzeniu poprawności danych.

```
class Pracownik {
private:
    string nazwisko;
public:
    string get_nazwisko();
    void set_nazwisko(string new_nazwisko);
};

string Pracownik::get_nazwisko()
{
    return nazwisko;
}

void Pracownik::set_nazwisko(string new_nazwisko)
{
    if (validate_name((new_nazwisko))
    {
        nazwisko = new_nazwisko;
    }
}
```

Korzystamy tutaj z funkcji pomocniczej validate_name() zdefiniowanej gdzie indziej, nie jest to teraz istotne. Dzięki hermetyzacji jesteśmy w stanie zweryfikować dane i nie dopuścić, by ktoś przypisał je niepoprawnie.

Konstruktory i destruktory

Tworząc klasy w C++ na pewno przyjdzie Ci kiedyś na myśl, czy da się utworzyć metodę wywoływaną przy każdym utworzeniu nowego obiektu. Oczywiście, jest taka możliwość. Metoda ta nazywana jest konstruktorem. Jej funkcją jest zainicjowanie instancji klasy.

Konstruktor jest funkcją posiadającą tą samą nazwę co klasa oraz niezwracająca żadnej wartości. Gdy sam nie zdefiniujesz konstruktora, kompilator zrobi to za Ciebie.

```
class Pracownik {
public:
    string nazwisko;
    string imie;

    Pracownik(); // deklaracja konstruktora
};

Pracownik::Pracownik() // definicja konstruktora
{
    // instrukcje
    cout << "Stworzono nowy obiekt";
}
```

Po utworzeniu nowego obiektu klasy Pracownik na ekranie wyświetli się tekst. Funkcja została wywołana automatycznie podczas tworzenia nowej instancji.

Konstruktor może również przyjmować argumenty.

```
#include <iostream>
#include <string>

using namespace std;

class Pracownik {
public:
    string nazwisko;
    string imie;

    Pracownik();
    Pracownik(string nowe_nazwisko);
    Pracownik(string nowe_nazwisko, string nowe_imie);
};

Pracownik::Pracownik()
{
    // konstruktor bez parametrów
}

Pracownik::Pracownik(string nowe_nazwisko)
{
    // konstruktor z jednym parametrem

    nazwisko = nowe_nazwisko;
}
```

```
Pracownik::Pracownik(string nowe_nazwisko, string nowe_imie)
{
    // konstruktor z dwoma parametrami

    nazwisko = nowe_nazwisko;
    imie = nowe_imie;
}

int main()
{
    Pracownik pracownik1;
    Pracownik pracownik2 ("Kowalski");
    Pracownik pracownik3 ("Kowalski", "Jan");
}
```

Dla obiektów pracownik1, pracownik2, pracownik3 zostaną uruchomione odpowiednie konstruktory w zależności od ilości podanych parametrów.

Istnieje też funkcja przeciwna, a mianowicie destruktor, który uruchamiany jest podczas niszczenia obiektu. W praktyce niszczenie zmiennej obiektowej następuje w chwili, gdy program wychodzi poza zasięg tej zmiennej. Destruktor jest funkcją o tej samej nazwie, co konstruktor, jednak poprzedzony jest znakiem tyldy (~).

```
class Pracownik {
public:
    string nazwisko;

    Pracownik(); // konstruktor
    ~Pracownik(); // destruktor
};
```

Destruktor nie posiada argumentów. Jedna klasa może mieć tylko jeden destruktor.

Dziedziczenie

Tworząc różne klasy, możesz kiedyś dojść do momentu, w którym dwie klasy będą miały część wspólnych pól i metod. Na przykład motocykl posiada cenę, moc silnika i dwa koła. Samochód również posiada cenę, moc silnika, lecz ma cztery koła. Czy dałoby się zdefiniować klasę pojazd, łączącą cechy obu obiektów, a następnie na jej podstawie zbudować dwie klasy pochodne? Tak! Tym w C++ właśnie jest dziedziczenie.

Dziedziczenie umożliwia nam konstruowanie szeregu klas wywodzących się z jednej klasy i korzystanie w przejrzysty i wspólny sposób z ich możliwości.

Klasę, która służy za bazę naszej relacji dziedziczenia nazywamy klasą bazową, natomiast klasa, która jest tworzona, jest klasą pochodną. Klasa pochodna może mieć dostęp do elementów klasy bazowej, może ją rozszerzyć o nowe pola i metody, ale także zmodyfikować metody klasy bazowej.

Jak to wygląda w praktyce?

```
class KlasaBazowa {

};

class KlasaPochodna : public KlasaBazowa {

};
```

Składnia jest bardzo prosta i intuicyjna. Klasa pochodna została utworzona na podstawie klasy bazowej. Słowo kluczowe public możemy zastąpić wybranym przez nas typem dziedziczenia. Podobnie jak przy kontroli dostępu, wyróżniamy trzy typy dziedziczenia:

Dziedziczenie publiczne (public) - Składowe publiczne klasy bazowej są odziedziczone jako publiczne, a składowe chronione jako chronione. Jest to najczęściej stosowany typ dziedziczenia.

Dziedziczenie chronione (protected) - Składowe publiczne są dziedziczone jako chronione, a składowe chronione jako chronione.

Dziedziczenie prywatne (private) - Składowe publiczne są dziedziczone jako prywatne, a chronione jako prywatne. Jeśli nie podamy jawnie sposobu dziedziczenia, ten typ zostanie wybrany domyślnie.

Załóżmy, że tworzymy program symulujący życie zwierząt. Każde zwierzę je, ale ssak oddycha inaczej niż ryba. Stwórzmy zatem klasę bazową, z której zbudujemy klasy pochodne.

```
class Zwierze {
public:
    Zwierze(); // konstruktor
    void Jedz(); // bo każde zwierzę przyjmuje pokarm
};
```

Na jej podstawie zdefiniujmy nowe klasy - Ssak oraz Ryba

```
class Ssak : public Zwierze {
public:
    Ssak();
    void OddychajPlucami();
```

```
};

class Ryba : public Zwierze {
public:
    Ryba();
    void OddychajSkrzelami();
};
```

Możemy zatem korzystać z funkcji Jedz() zarówno w klasie Ssak jak i Ryba:

```
Ssak pies;
pies.OddychajPlucami();
pies.Jedz();

Ryba okon;
okon.Jedz();
okon.OddychajSzkrzelami();
```

Możemy zechcieć też zablokować możliwość tworzenia klas pochodnych z naszej klasy. W C++ od wersji C++11 służy do tego słowo kluczowe final. Próba dziedziczenia z takiej klasy spowoduje błędy kompilacji.

```
class KlasaFinalna final {
public:
    KlasaFinalna();
};

class Pochodna: public KlasaFinalna {

};
```

W C++, w przeciwieństwie na przykład do Javy, dopuszczane jest też dziedziczenie wielobazowe. Oznacza to, że możliwe jest, aby klasa miała dwóch i więcej przodków. Powoduje to, że hierarchia klas nie przypomina już drzewa, tylko tylko skierowany graf acykliczny.

Istnieje teoria mówiąca, że opieranie projektów klas o dziedziczenie wielobazowe, zwanym również dziedziczeniem wielokrotnym, jest błędem projektowym. Przedstawię jednak taką możliwość, gdyż sam język C++ na to pozwala, tym bardziej, że temat ten jest często poruszany w celach dydaktycznych na zajęciach.

Rozważmy taki przykład:

```
class KlasaA {
public:
    KlasaA();
    int poleA;
};

class KlasaB {
public:
    KlasaB();
    int poleB;
};

class KlasaC : public KlasaA, public KlasaB {
public:
```

```
    KlasaC();
}
```

Klasa KlasaC dziedziczy po klasie KlasaA oraz KlasaB. Oznacza to, że taki zapis będzie poprawny:

```
KlasaC obiekt;
obiekt.poleA = 0;
obiekt.poleB = 10;
```

Co w sytuacji, gdy pola będą się powtarzać? Czy jedna klasa nadpisze drugie pole? Nie. Wówczas konieczna jest dodatkowa klasyfikacja nazwą klasy, jak na tym przykładzie:

```
class KlasaA {
public:
    KlasaA();
    int pole;
    void metoda()
    {

    }
};

class KlasaB {
public:
    KlasaB();
    int pole;
    void metoda()
    {

    }
};

class KlasaC : public KlasaA, public KlasaB {
public:
    KlasaC();
    void nowa_metoda()
    {
        KlasaA::metoda();
        KlasaB::metoda();
    }
}

KlasaC obiektC;

obiektC.KlasaA::pole = 2;
obiektC.KlasaB::pole = 0;
obiektC.KlasaA::metoda();
```

Funkcje wirtualne

W poprzedniej części widzieliśmy przykład dziedziczenia wielokrotnego, gdzie klasa pochodna dziedziczyła od dwóch klas bazowych metodę o tej samej nazwie. Da się jednak inaczej. Wyobraź sobie, że definiujesz klasę Zwierzę. Wiadome, że ssak oddycha płucami, ryba skrzelami, a płazy do wymiany gazowej używają również skóry (jako ciekawostkę podam fakt, że niektóre płazy w ogóle nie posiadają płuc i skóra jest ich podstawowym medium wymiany gazowej!). Czy zamiast definiowania osobnej metody oddychania dla każdej gromady dałoby się utworzyć jedną, wspólną, a następnie ponownie ją zdefiniować w każdej klasie pochodnej? Zauważ, że takie podejście znacznie uprościłoby sprawę i zdjęło z programisty konieczność znajomości biologii przy każdym wywołaniu metody.

Jak się na pewno domyślasz, jest taka możliwość! Spróbujmy tak:

```
class Zwierze {
public:
    void oddychaj()
    {
        cout << "Oddycham";
    }
};

class Ssak : public Zwierze {
public:
    void oddychaj()
    {
        cout << "Oddycham plucami";
    }
};

class Ryba : public Zwierze {
public:
    void oddychaj()
    {
        cout << "Oddycham skrzelami";
    }
};
```

Następnie zdefiniujemy wskaźnik na obiekt klasy Zwierze, a później ustawimy go na obiekt klasy pochodnej.

```
Zwierze *wskaznik;
Zwierze moje_zwierze;
Ssak moj_ssak;
Ryba moja_ryba;

// ustawiamy wskaznik klasy bazowej na obiekt klasy bazowej
wskaznik = &moje_zwierze;
wskaznik->oddychaj();

// następnie przepimany go na obiekty klasy pochodnej
wskaznik = &moj_ssak;
wskaznik->oddychaj();

wskaznik = &moja_ryba;
```

```
wkaznik->oddychaj();
```

Po wywołaniu tych metod na ekranie trzy razy pokaże się tekst „Oddycham plucami”. Oznacza to, że pomimo iż przepięliśmy wskaźnik na klasę pochodną, wywołana została metoda klasy bazowej. Stało się tak dlatego, że wskaźnik był typu zmiennej bazowej, więc kompilator sięgnął po jej wersję metody.

Istnieje jednak możliwość poinstruowania kompilator, aby sam zorientował się z obiektem jakiej klasy ma do czynienia i wywołał odpowiednią dla niej funkcję. Służy do tego słowo-klucz virtual. Taką metodę nazywamy metodą wirtualną. Spróbujmy zatem jeszcze raz:

```
class Zwierze {
public:
    virtual void oddychaj()
    {
        cout << "Oddycham";
    }
};

class Ssak : public Zwierze {
public:
    void oddychaj()
    {
        cout << "Oddycham plucami";
    }
};

class Ryba : public Zwierze {
public:
    void oddychaj()
    {
        cout << "Oddycham skrzelami";
    }
};
```

A następnie

```
Zwierze *wskaznik;
Zwierze moje_zwierze;
Ssak moj_ssak;
Ryba moja_ryba;

// ustawiamy wskaznik klasy bazowej na obiekt klasy bazowej
wskaznik = &moje_zwierze;
wskaznik->oddychaj();

// następnie przepimany go na obiekty klasy pochodnej
wskaznik = &moj_ssak;
wskaznik->oddychaj();

wskaznik = &moja_ryba;
wkaznik->oddychaj();
```

Tym razem zadziałało zgodnie z naszymi oczekiwaniami. Za każdym razem gdy mamy do czynienia z metodą wirtualną, kompilator wstrzyma się z decyzją co do faktycznie przywoływanej metody. Jej podjęcie nastąpi dopiero w stosowanej chwili podczas działania gotowej aplikacji;

nazywamy to późnym wiązaniem (ang. late binding) funkcji. Dzięki temu możemy na przykład pytać użytkownika jaki typ obiektu ma zamiar podpiąć pod dany wskaźnika albo losować obiekt jaki ma zostać przypisany. Takie czynności znane są dopiero podczas działania programu, mimo to zostanie wywołana metoda odpowiednia dla danej funkcji.

Zauważmy również, że to podejście ma sens jedynie podczas pracy ze wskaźnikami. Gdy do obiektów odnosimy się przez nazwę zmiennej, kompilator już w trakcie kompilacji wie z obiektem jakiej klasy ma do czynienia, więc wirtualność metod nie ma kompletnie znaczenia.

W C++ słowo kluczowe virtual można pominąć w deklaracjach w klasach pochodnych. Tylko deklaracja funkcji musi być poprzedzona słowem kluczowym funkcji. Przed jej definicją nie umieszczamy słowa kluczowego, chyba że jej deklaracja jest jednocześnie jej definicją (implementacja metody zawarta jest w ciele funkcji).

Atrybut virtual możemy dodawać do każdej funkcji. Dotyczy to również konstruktorów oraz destruktorów. W przypadku konstruktorów jest to zbędne, ponieważ wywołanie konstruktora klasy pochodnej pociągnie za sobą wywołanie konstruktora klasy bazowej.

Zupełnie inaczej sprawa ma się z destruktorami. Tutaj użycie omawianego modyfikatora jest nie tylko możliwe, ale też prawie zawsze konieczne. Nieobecność wirtualnego destruktora w klasie bazowej może bowiem prowadzić do tzw. wycieków pamięci, czyli bezpowrotnej utraty zaalokowanej pamięci operacyjnej.

W naszym pierwszym przykładzie utworzyliśmy implementację metody oddychaj() dla klasy Zwierze. Co w sytuacji, gdy nie chcielibyśmy nigdy tworzyć obiektu tej klasy i używać jej tylko do dziedziczenia? Nie byłoby więc potrzeby definiowania jej metod, ponieważ i tak musiałby zostać przedefiniowane w klasach potomnych.

Funkcje takie można zdefiniować jako metody czysto wirtualne. Robi się to następująco:

```
class Zwierze {
public:
    virtual void oddychaj() = 0;
};
```

Dość nietypowy zapis nie ma żadnego związku ze zwracaną przez funkcję wartością, gdyż jak widzimy, w tym przypadku jest to void. Jest to najwidoczniej kaprys twórców C++.

Dodanie do klasy choć jednej metody czysto wirtualnej powoduje, że klasa ta staje się klasą abstrakcyjną. Z tego powodu nie jest przeznaczona do instancjowania (tworzenia z niej obiektów), a jedynie do wyprowadzania klas pochodnych.