Wszystko jest obiektem

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

Techniki obiektowe przybliżają kodowanie do prawdziwego świata.

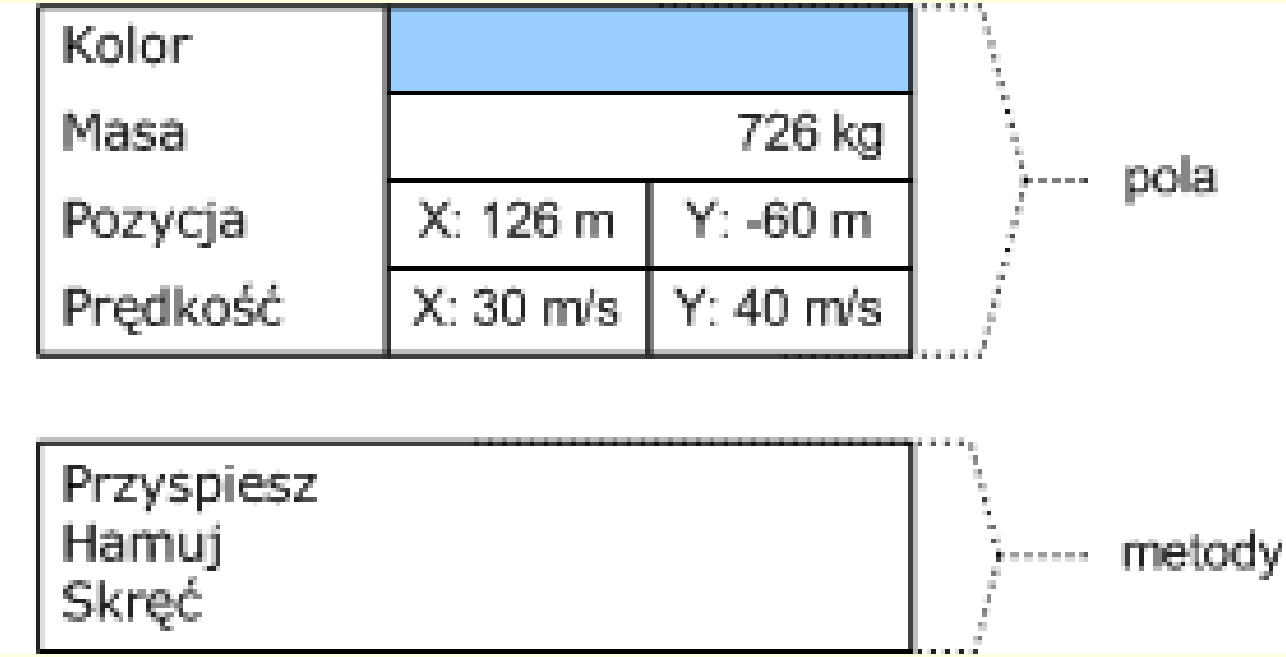
Obiekt składa się z opisujących go danych oraz może wykonywać ustalone czynności .

Podobnie jak omówione niedawno struktury, obiekty zawierają pola , czyli zmienne.
Ich rolą jest przechowywanie pewnych informacji o obiekcie - jego charakterystyki.

**Obiekt może wykonywać pewne działania,
a więc uruchamiać zaprogramowane funkcje;
nazywamy je metodami albo funkcjami
składowymi .
Czynią one obiekt tworem aktywnym
– nie jest on jedynie pojemnikiem na dane,
lecz może samodzielnie nimi manipulować.**

Załóżmy, że chcemy mieć w programie obiekt jadącego samochodu

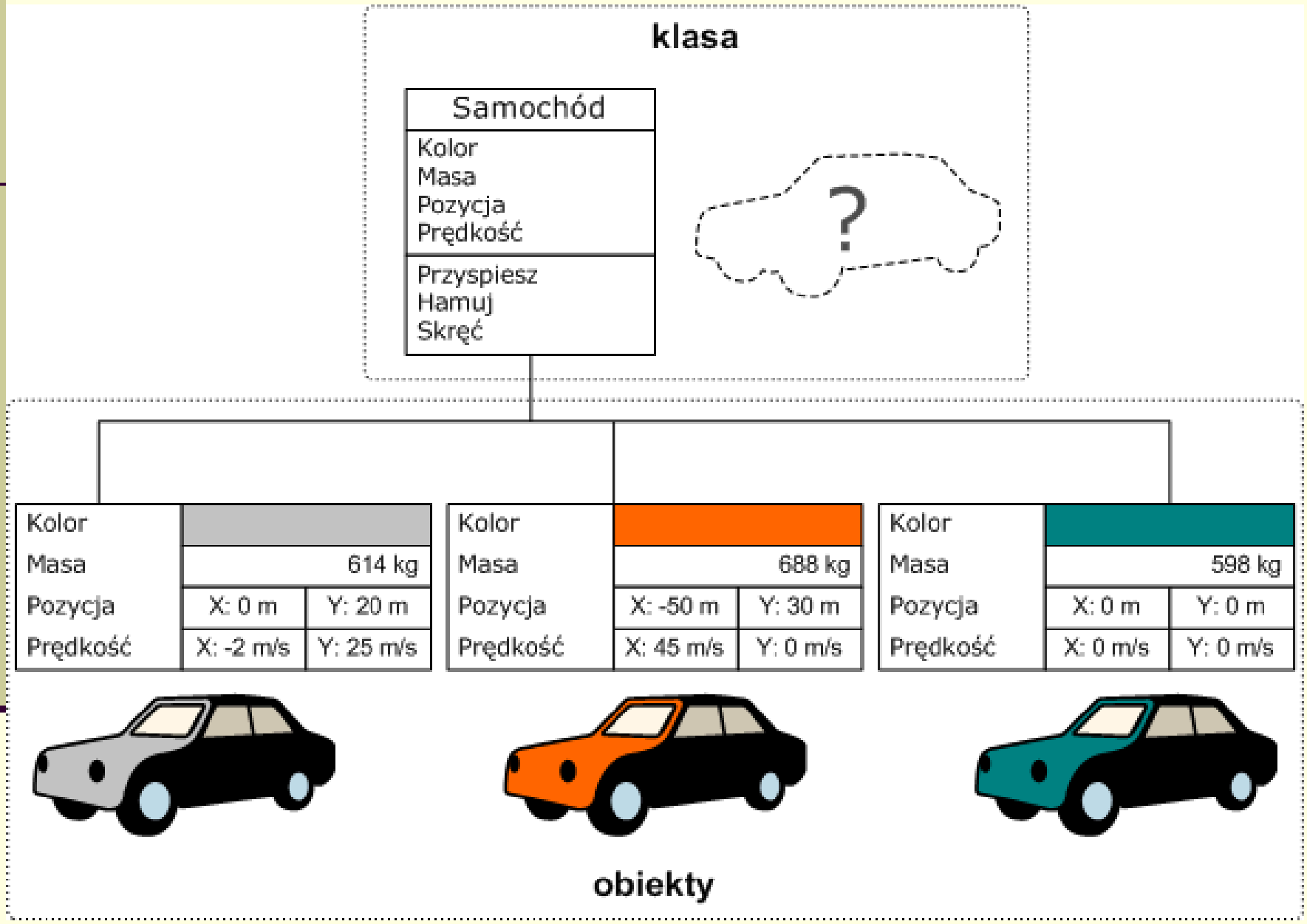
**Ustalamy więc dla niego pola,
które będą go określały, oraz metody,
które będzie mógł wykonywać.**



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

Pola i metody klasy nazywamy **składnikami klasy**

Klasa jest zatem czymś w rodzaju wzorca - matrycy, wedle którego "produkowane" są kolejne obiekty (instancje) w programie.
Mogą one różnić się od siebie, ale tylko co do wartości poszczególnych pól; wszystkie będą jednak należeć do tej samej klasy i będą mogły wykonywać na sobie te same metody.


klasa
Samochód
Kolor
Masa
Pozycja
Prędkość
Przyspiesz
Hamuj
Skręć
?
Kolor
Masa
614 kg
Pozycja
X: 0 m
Y: 20 m
Prędkość
X: -2 m/s
Y: 25 m/s
Kolor
Masa
688 kg
Pozycja
X: -50 m
Y: 30 m
Prędkość
X: 45 m/s
Y: 0 m/s
Kolor
Masa
598 kg
Pozycja
X: 0 m
Y: 0 m
Prędkość
X: 0 m/s
Y: 0 m/s
obiekty

Definicja klasy

```
class nazwa_klasy
 {
... // dane i funkcje składowe ;
 };
```

Przykład definicji klasy

```
class CCar
{
private :
float m_fMasa;
COLOR m_Kolor;
VECTOR2 m_vPozycja;
public :
VECTOR2 vPredkosc;
//
// metody
void Przyspiesz( float fIle);
void Hamuj( float fIle);
void Skrec( float fKat);
};
```

Definicja klasy składa się z m.in. z deklaracji zmiennych różnego typu. Trzeba pamiętać, że są to tylko deklaracje, a nie definicje. Zmienne zadeklarowane w definicji klasy zostaną zdefiniowane (czyli zaistnieją w pamięci jako liczby) dopiero gdy stworzymy jakiś obiekt danej klasy.

Oznacza to, że w definicji klasy nie można nadawać zmiennym wartości, ponieważ jeszcze nie istnieją.

Tworzenie obiektów

```
CCar Samochod;

// przypisanie wartości polu
Samochod.vPredkosc.x = 100.0 ;
Samochod.vPredkosc.y = 50.0 ;
// wywołanie metody obiektu
Samochod.Przyspiesz ( 10.0 );
```

Dostęp do składników klasy

Definiując klasę możemy ograniczyć dostęp do niektórych składników klasy. Są trzy rodzaje dostępu do składników:

public

oznacza, że składniki deklarowane po tej etykiecie są dostępne z każdego miejsca programu,

private

oznacza, że składniki deklarowane po tej etykiecie dostępne są tylko dla funkcji składowych tej klasy; funkcje globalne (zwykłe) nie mają dostępu do tych składników, a więc z funkcji main również nie ma dostępu do tych danych,

protected

tak jak w przypadku private z tą tylko różnicą, że dostęp do takich składników mają jeszcze klasy wywodzące się z tej klasy

Specyfikatory dostępu:

public

private

protected

UWAGA:
Jeżeli nie ma podanego zakresu dostępu, domyślnie, dane i funkcje zadeklarowane wewnątrz klasy są prywatne

W samym kodzie programu dostęp do zmiennych wewnątrz klasy, które są nazywane polami jest uzależniony od tego czy mamy do czynienia ze zmienną typu klasowego, czy wskaźnikiem na ten obiekt.

W przypadku zmiennej o typie klasa, dostęp do pól uzyskuje się operatorem kropka (.):

```
obiekt.pole = wartosc; //przypisanie wartości
                       polu w obiekcie
```

Jeżeli mamy do czynienia ze wskaźnikiem, operatorem jest strzałka ->

```
obiekt->pole = wartosc;
```

Przykład definicji prostej klasy:

```
class Ulamek
{
public:
   int licznik;
   int mianownik;
};
```

Przykład deklaracji obiektu zdefiniowanej klasy :

```
Ulamek ul1;
```

Dostęp do składników obiektu:

```
ul1.licznik;
ul1. mianownik;
```

HERMETYZACJA

Hermetyzacja pozwala na ukrycie pewnych danych i funkcji obiektu przed bezpośrednim dostępem z zewnątrz.

Przykład

Poprawimy klasę Ulamek w taki sposób, aby dostęp do danych: licznik i mianownik był możliwy przez metody, które „wiedzą” jak postępować z licznikiem i mianownikiem. W klasie Ulamek niedopuszczalna powinna być operacja przypisania mianownikowi wartości 0.

ulamek_klasa_def

ulamek_klasa_progr

Zadanie

Napisz klasę `wektor`, która będzie miała metodę:
a) pobierającą współrzędne wektora,
b) wyświetlającą współrzędne wektora w postaci
na przykład [x;y] ,
c) obliczającą długość wektora

Wykorzystaj zdefiniowaną klasę w programie.

Przykład

Sposób deklarowania obiektów klasy `ulamek`

```
Ulamek ul1;  //na początku deklaracja obiektu
ul1.zapisz(4,5);  //a potem uruchamiamy
                  //odpowiedniąfunkcję
```

Wygodniej byłoby nadać wartość początkową przy deklaracji, tak, jak na przykład, deklarując zmienną typu int, możemy nadać jej początkową wartość:

```
int z=6;
```

Konstruktor

Konstruktor jest to funkcja w klasie, wywoływana w trakcie tworzenia każdej instancji. Funkcja może stać się konstruktorem gdy spełni poniższe warunki

- ✓Ma identyczną nazwę jak nazwa klasy
- ✓Nie zwraca żadnej wartości (nawet **void)**
- ✓Jest ustawiona w przestrzeni publicznej klasy (sekcja **public:)**

Należy dodać że każda klasa ma swój konstruktor. Nawet jeżeli nie zadeklarujemy go jawnie zrobi to za nas kompilator (stworzy wtedy konstruktor bezparametrowy).

Konstruktor - definicja

Konstruktorem nazywamy specjalną metodę automatycznie uruchamianą w trakcie definiowania (tworzenia) obiektu pozwalającą na nadanie początkowych wartości danym obiektu.

Destruktor - definicja

Destruktorem nazywamy specjalną metodę bezparametrową, która jest wykonywana zawsze w momencie usuwania obiektu.

Destruktor ma nazwę taką jak nazwa klasy, ale poprzedzoną znakiem ~

```
~ulamek ()
{cout<<„usuwam obiekt” ;
}  //jeśli w swojej klasie nie zdefiniujemy destruktora- zostanie
 on wygenerowany przez kompilator
```

Destruktor jest metodą, która uruchamia się automatycznie zwykle tuż przed zakończeniem programu. Sami możemy spowodować wywołanie destruktora dla obiektów znajdujących się w pamięci dynamicznej

Funkcja zaprzyjaźniona - definicja

Funkcja zaprzyjaźniona to taka funkcja zewnętrzna (niebędąca składnikiem klasy), dla której w definicji klasy umieszczona jest deklaracja przyjaźni za pomocą słowa kluczowego `friend` .

przykład

W przypadku klasy Ulamek funkcja zewnętrzna miałaby problem z dostępem do licznika i mianownika, gdyż są to składowe prywatne obiektu typu Ulamek. aby funkcja niebedąca elementem klasy miała dostęp do jej składników prywatnych, musimy w klasie zapisać deklarację przyjaźni (zaprzyjaźnienia się z tą funkcją) .

przeciążanie nazw funkcji

W języku C++ możliwe jest nadanie dwóm lub większej liczbie definicji funkcji tej samej nazwy.
Jest to przeciążenie.
Gdy przeciążamy nazwę funkcji, definicje funkcji muszą mieć różną liczbę parametrów formalnych lub parametry formalne różnych typów.
Przy wywołaniu funkcji kompilator używa definicji funkcji, dla której liczba parametrów formalnych i ich typy odpowiadają argumentom wywołania funkcji.

Zadanie

Napisz klasę `wektor`, która będzie miała metodę:
a) pobierającą współrzędne wektora,
b) wyświetlającą współrzędne wektora w postaci
na przykład [x;y] ,
c) obliczającą długość wektora

oraz funkcję zaprzyjaźnioną:
d) sumującą dwa wektory,
e) obliczającą iloczyn wektora przez liczbę.
Wykorzystaj zdefiniowane funkcje w programie.

Dziedziczenie

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

Wszystkie klasy, które powstają w ten sposób (nazywamy je **pochodnymi),**
posiadają pewne elementy wspólne.
Części te są dziedziczone z klas bazowych, gdyż tam właśnie zostały zdefiniowane.

Definicja klasy pochodnej

```
class nazwa_klasy :[operator_widoczności] nazwa_klasy_bazowej,
                                    [operator_widoczności] nazwa_klasy_bazowej ...
 //podajemy klasy bazowe, z których chcemy dziedziczyć.
//Czynimy to, wpisując dwukropek po nazwie definiowanej właśnie klasy
//i podając  dalej listę jej klas bazowych, oddzielonych przecinkami.
//Kolejne specyfikatory, które opcjonalnie możemy umieścić przed każdą
//nazwą_klasy_bazowej  wpływają na proces  dziedziczenia, a dokładniej
//na prawa dostępu, na jakich klasa pochodna otrzymuje składowe
// klasy bazowej.
{
deklaracje_składowych
};
```

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

Dziedziczenie to technika pozwalająca na definiowanie nowej klasy przy wykorzystaniu klasy już wcześniej istniejącej.

Załóżmy, że mamy klasę

```
class punkt {
public:
     float x, y ;

     punkt(float, float );          // konstruktor
     void wypisz() ;
     void przesun(float, float) ;
};
```

```
class lepszy_punkt : public punkt {
public :
     char opis[10] ;
                                  // ——————————————————— konstruktor
     lepszy_punkt(float =0, float =0, char* =NULL);
     void wypisz() ;              // <—— dodatkowa funkcja
} ;
```

Mamy dzięki temu nową klasę. Klasa `lepszy_punkt` wywodzi się od klasy `punkt`. Mówimy, że klasa `lepszy_punkt` jest **klasą pochodną** klasy `punkt`. Natomiast klasa `punkt` jest dla klasy `lepszy_punkt` **klasą podstawową**.

W klasie pochodnej możemy:

- – zdefiniować dodatkowe dane składowe,
- – zdefiniować dodatkowe funkcje składowe,

  Definiowanie nowych funkcji składowych – bez definiowania dodatkowych danych składowych także ma sens. Jest to jakby wyposażenie klasy w nowe zachowania;

- – zdefiniować składnik (najczęściej funkcję składową), który istnieje już w klasie podstawowej.

  Powoduje to jakby korektę czegoś, co nam z klasy podstawowej nie bardzo odpowiadało i czego nie chcemy w tym kształcie dziedziczyć.

Zwróć uwagę na pierwszą linijkę definicji klasy `lepszy_punkt`. Po dwukropku jest napisane, że klasa `lepszy_punkt` wywodzi się od klasy `punkt`. To wyrażenie po dwukropku to tzw. **lista pochodzenia**. Na tej liście umieszczona jest informacja od czego wywodzi się dana klasa pochodna.

W definicji klasy podstawowej „punkt” jest taka sama funkcja „wypisz”. W rezultacie w klasie pochodnej są dwie funkcje „wypisz” Funkcja wypisz z klasy pochodnej zasłania (nie przesłania) funkcję wypisz z klasy podstawowej. Nie jest to przesłanianie, gdyż funkcje mają inny zakres ważności.

Istnieją więc dwie funkcje `wypisz`: pierwsza mająca zakres klasy podstawowej, a druga mająca zakres klasy pochodnej. Ile razy pracując na obiektach klasy pochodnej wywołana zostanie funkcja `wypisz`, to kompilator aktywuje tę z klasy pochodnej.

Czy to znaczy, że - jeśli jakiś składnik (dana lub funkcja) jest zasłonięty - to nie można się do niego odnosić?

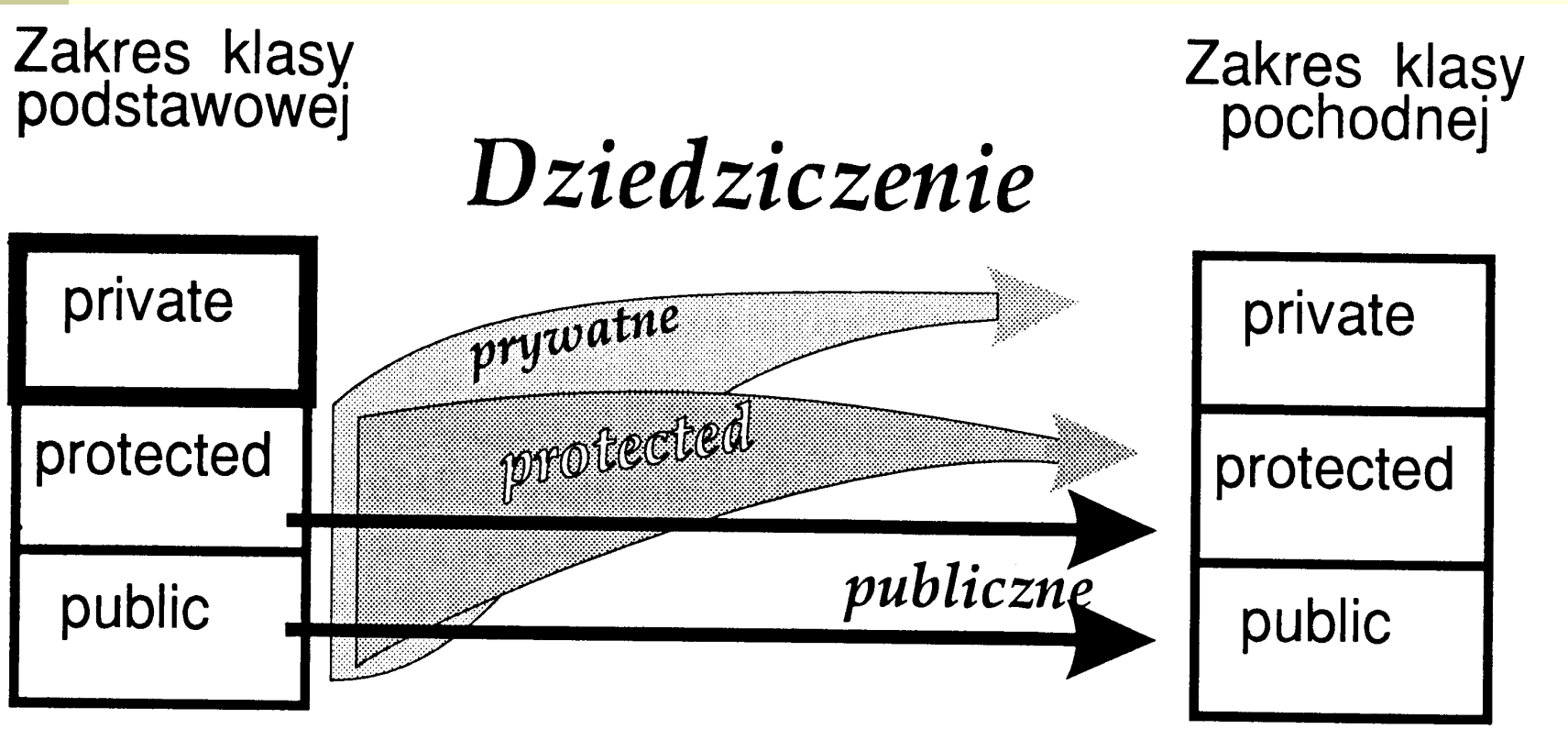
Można. Trzeba jednak uczynić to posługując się kwalifikatorem zakresu. Chodzi o dwa dwukropki zwane inaczej - operatorem zakresu. W naszym wypadku do zasłoniętej funkcji `wypisz` odnosimy się tak,

```
lepszy_punkt obiekt ;          // definicja obiektu

obiekt.wypisz();               // wywołanie funkcji  wypisz
                               // z klasy pochodnej „lepszy_punkt“

obiekt.punkt::wypisz();        // wywołanie funkcji  wypisz
                               // z klasy podstawowej „punkt“
```

Definiując obiekt klasy pochodnej powinniśmy pamiętać, że równocześnie w jego wnętrzu tkwi odziedziczony fragment będący jakby obiektem klasy podstawowej.



Zwracam jednak uwagę, że strzałki pokazują to, do którego obszaru dostępu klasy pochodnej następuje dziedziczenie, a nie to, czy dziedziczenie następuje w ogóle. Dziedziczone są wszystkie składniki. Brak strzałki przy obszarze `private` w klasie podstawowej oznacza, że zmiana rodzaju dostępu nie następuje. Odziedziczone elementy mają nadal dostęp `private`, ale w zakresie ważności klasy podstawowej.

Konstruktor klasy pochodnej

wygląda tak, jak zwykły konstruktor, z tym, że na liście inicjalizacyjnej można umieścić wywołanie konstruktora klasy podstawowej.

```
class A {
public:
      A() ;                              // konstr domniemany
      A(float) ;                         // inny konstruktor
      // ...
};
////////////////////////////////////////////////////////////
class B : public A {
public:
      B() ;                              // deklaracja konstruktora
};
////////////////////////////////////////////////////////////
//  definicja konstruktora klasy  B
B::B(int x) : A()                        // <— lista inicjalizacyjna z wywołaniem
                                         // konstruktora klasy podstawowej
{
      // ... ciało konstruktora
}
```

Specyfikator dostępu private : poprzedza deklaracje składowych, które mają być dostępne **jedynie dla metod definiowanej klasy. Oznacza to, iż nie można się do nich dostać, używając obiektu lub wskaźnika na niego oraz operatorów „ .” lub „->” .**

Ta wyłączność znaczy również, że prywatne składowe **nie są dziedziczone i nie ma do nich dostępu w klasach pochodnych, gdyż nie wchodzą w ich skład.**

specyfikator protected ("chronione")
także nie pozwala, by użytkownicy obiektów naszej klasy ingerowali w pola i metody. Jak sama nazwa wskazuje, są one **chronione przed takim dostępem z zewnątrz.**
Jednak w przeciwieństwie do deklaracji private, składowe zaznaczone przez protected
są dziedziczone i występują w klasach pochodnych, będąc dostępnymi dla ich własnych metod.

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

PRZYKŁAD

```
class Cprostokat
{
private :
// wymiary prostokąta
float  szerokosc, wysokosc;
protected :
// pozycja na ekranie
float  X,  Y;
public :
// konstruktor
Cprostokat() { X = Y = 0.0 ;
szerokosc = wysokosc = 10.0 ; }
// metody
float Pole()  { return szerokosc * wysokosc; }
float Obwod()  { return 2 * (szerokosc+wysokosc); }
};
```

PRZYKŁAD – dziedziczenie po klasie Cprostokat

```
class Ckwadrat : public Cprostokat // dziedziczenie z Cprostokat
{
private :
// zamiast szerokości i wysokości mamy tylko długość boku
float  DlugoscBoku;
// pola X i  Y są dziedziczone z klasy bazowej, więc nie ma
// potrzeby ich powtórnego deklarowania
public :
// konstruktor
Ckwadrat { DlugoscBoku = 10.0 ; }
// nowe metody
float Pole()  { return DlugoscBoku * DlugoscBoku; }
float Obwod() { return 4 * DlugoscBoku; }
};
```

Co nie jest dziedziczone?

konstruktory

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

destruktory

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

operator przypisania =

Dziedziczenie w obrębie klas strumieni

Strumień cin należy do klasy wszystkich strumieni wejściowych
ale nie należy do klasy strumieni wejściowych działających na plikach,
ponieważ cin nie ma funkcji składowej open, ani funkcji składowej close
ale oba te strumienie są w pewnym sensie do siebie podobne,
ponieważ w obu tych strumieniach można użyć operatora >> (ekstrakcji)

Strumien wejsciowy cin jest typu istream, a strumien wejsciowy
działający na pliku jest typu ifstream,
klasa ifstream jest klasa wywiedzioną z klasy istream
(jest klasą pochodna klasy istream)

suma_ofstream_istream.cpp

SZABLONY

**Szablony służą do definiowania funkcji oraz klas,
w których parametrami są nazwy typów.
To pozwala projektować funkcje, których argumenty mogą mieć
w różnych sytuacjach różne typy oraz klasy
o wiele bardziej uniwersalne od omawianych dotychczas**

szablony_funkcji.cpp

Definicja i deklaracja funkcji

template<class T> //nazywa się to często prefiksem
//szablonu
Jego zadaniem, jest poinformowanie kompilatora, że definicja lub deklaracja funkcji, która zaraz nastąpi jest szablonem, w którym T to parametr określający typ.
W tym kontekście słowo „class” oznacza w zasadzie „typ”

UWAGA

Standard ANSI dopuszcza użycie w prefiksie szablonu słowa kluczowego „typename” zamiast „class”

(*ang. American National Standards Institute - Amerykański Narodowy Instytut Standardów) - organizacja zajmująca się opracowywaniem norm obowiązujących w przemyśle komputerowym. Koordynuje ona działalność grup, które ustanawiają standardy w określonych dziedzinach, np. Institute of Electrical and Electronics Engineers (IEEE). Dwa najbardziej znane standardy wprowadzone przez ANSI to SCSI (Small Computer Systems Interface) oraz ASCII.*)

Kilka parametrów określających typ

template<class T1, class T2 >

//jednakże wiekszośc szablonów funkcji ma tylko jeden parametr okreslajacy typ. Nie wolno przechowywać nieużywanych parametrów określających typ, to znaczy: każdy parametr typu musi być używany w szablonie funkcji przynajmniej raz.

Składnia szablonu klasy jest zasadniczo taka sama, jak szablonów funkcji

```
template<class T>
//parametru T okreslającego typ uzywa się w definicji klasy dokładnie tak samo, jak każdego inne go typu
```