

Programowanie obiektowe

na przykładzie języka C++

dr hab. Piotr Białas

pok 443, tel 5571

pon. 11⁰⁰-12⁰⁰, śr. 10⁰⁰-11⁰⁰

pbialas@th.if.uj.edu.pl

<http://th.if.uj.edu.pl/~pbialas/OOP/>

Wykład 2 – przypomnienie

- Programowanie obiektowe polega na znajdowaniu odpowiednich obiektów (klas)
- Każdy obiekt (klasa) jest realizacją pewnej abstrakcji (pojęcia)
- Każda klasa (obiekt) posiada swój zakres obowiązków i swoich współpracowników
- Znajdowanie klas, ustalanie zakresu ich obowiązków oraz współpracowników może i powinno odbywać się w oderwaniu od konkretnego języka programowania (przynajmniej na wczesnym etapie projektu)
- Pomóc w tym może technika kart CRC

Wykład 2 – przypomnienie

- Podczas projektowania klas należy zwrócić uwagę na operacje
 - Tworzenia: realizowane przez konstruktory
 - Niszczenia: realizowane przez destruktor
 - Kopiowania: realizowane przez konstruktor kopiujący i operator przypisania
- Kompilator C++ dostarcza standardowych realizacji każdej z tych operacji. Należy upewnić się czy to jest to co nam odpowiada!
Jeśli nie, to musimy te operacje przeddefiniować lub zablokować

Wykład 3

- System typów
 - Konwersja typów (jawna i niejawna)
 - Polimorfizm
- Dziedziczenie (dla interfejsu)
 - Interfejs i implementacja
 - Klasy abstrakcyjne
 - Interfejsy

System typów

- Każda klasa definiuje pewien nowy typ
- Każdy obiekt posiada określony typ
- Zmienne też mogą mieć określony typ i wtedy mogą przechowywać(wskazywać) tylko obiekty określonego typu. Mówimy wtedy że język ma silny system typów
- W językach z silnym system typów kompilator może sprawdzić poprawność przypisań i wywołań funkcji

```
#ifndef _stypes_
#define _stypes_

#include<iostream>
using namespace std;

class TypeA {
public:
    virtual void aFunction() {
        cerr << "AType"<<endl;
    }
};

class TypeB {
public:
    virtual void bFunction() {
        cerr << "BType"<<endl;
    }
};
#endif
```

```
#include "types.h"
```

```
main()
```

```
{
```

```
    TypeA a1,a2;
```

```
    TypeB b1,b2;
```

```
    a1=a2;
```

```
    a1=b1;
```

```
    b2=a2;
```

```
    b1=666;
```

```
    b1.bFunction();
```

```
    a1.bFunction();
```

```
}
```

```
styping.cpp: In function `int main()':
styping.cpp:12: error: no match for 'operator=' in 'a1 = b1'
styping.cpp:11: error: candidates are: TypeA& TypeA::operator=(const
TypeA&)
styping.cpp:13: error: no match for 'operator=' in 'b2 = a2'
stypes.h:14: error: candidates are: TypeB& TypeB::operator=(const
TypeB&)
styping.cpp:15: error: no match for 'operator=' in 'b1 = 666'
stypes.h:14: error: candidates are: TypeB& TypeB::operator=(const
TypeB&)
styping.cpp:18: error: `bFunction' undeclared (first use this
function)
styping.cpp:18: error: (Each undeclared identifier is reported only
once for
    each function it appears in.)
```


System typów

- Silny system typów pozwala poprawnie używać zdefiniowanych przez nas abstrakcji
- Wykorzystanie abstrakcji niezgodne z przeznaczeniem jest wyłapywane przez kompilator a nie powoduje błędów wykonania
- Umożliwia przeładowywanie funkcji i operatorów
- Ale taki system jest zbyt restrykcyjny uniemożliwiając mieszenie typów które w naturalny sposób można mieszać (np int i double) oraz nie zezwala na polimorfizm
- Silny system typów można “osłabić” poprzez:
 - niejawną konwersję typów
 - dziedziczenie – relacja “jest”: typ B jest typem A

Niejawna konwersja typów

- W C++ (i innych językach) kompilator może dokonywać automatycznej konwersji typów (klas) jeśli użytkownik na to pozwoli. Ponadto wbudowane typy numeryczne są niejawnie konwertowane na siebie.
- Jest to wygodne ale może powodować niejednoznaczności jeśli dany typ można przekształcić na kilka innych
- Zaciemnia to też mechanizm przeładowywania

Dziedziczenie

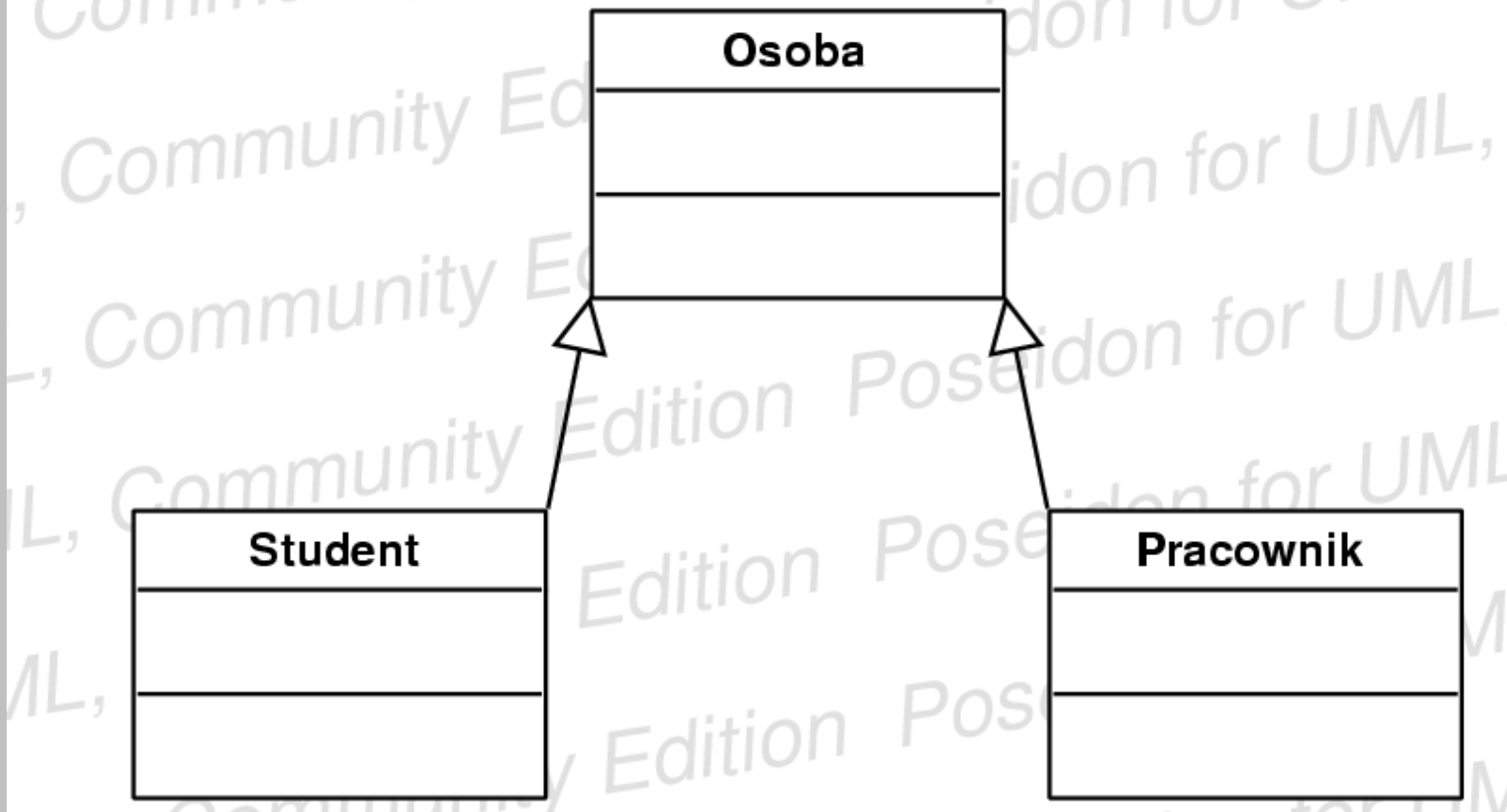
- Dziedziczenie to stwierdzenie że obiekty jednej klasy są jednocześnie obiektami jakiejś innej klasy
- “Jest” może oznaczać
 - Ma taką samą implementację
 - Ma takie same metody (interfejs)

Dziedziczenie i interfejsy

- Są dwa główne powody dziedziczenia:
 - Dla interfejsu
 - Klasa dziedzicząca może być użyta wszędzie tam gdzie klasa dziedziczona
 - Typ Klasy dziedziczącej jest typem klasy dziedziczonej
 - Dla implementacji
 - Klasa dziedzicząca wykorzystuje i/lub rozszerza działanie klasy dziedziczonej
-

Dziedziczenie i interfejsy

Osoba stanowi wspólny interfejs dla Studenta i pracownika tzn Student i Pracownik udostępniają taki sam zestaw funkcji



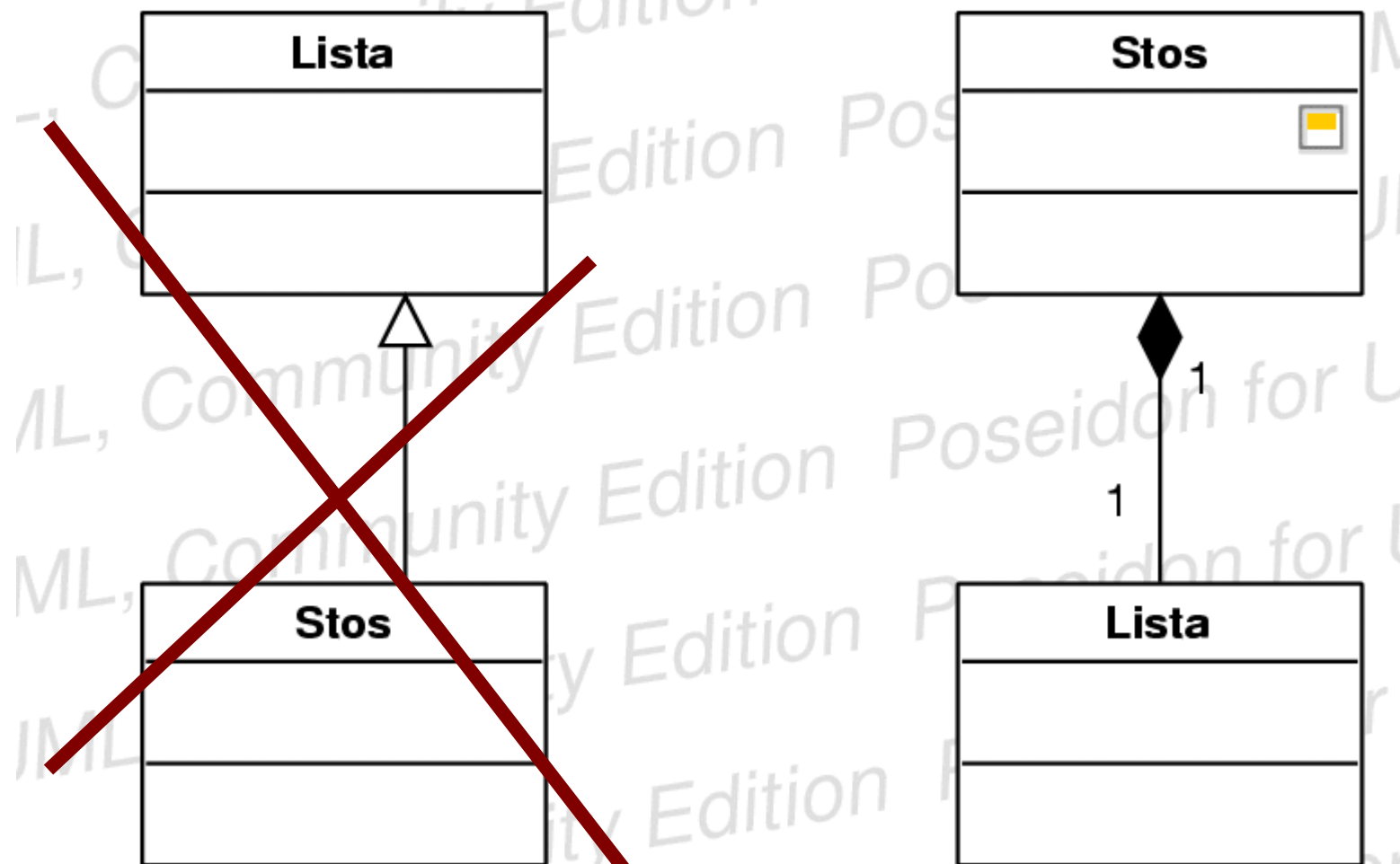
Student i Pracownik wykorzystują implementację klasy Osoba

Dziedziczenie i interfejsy

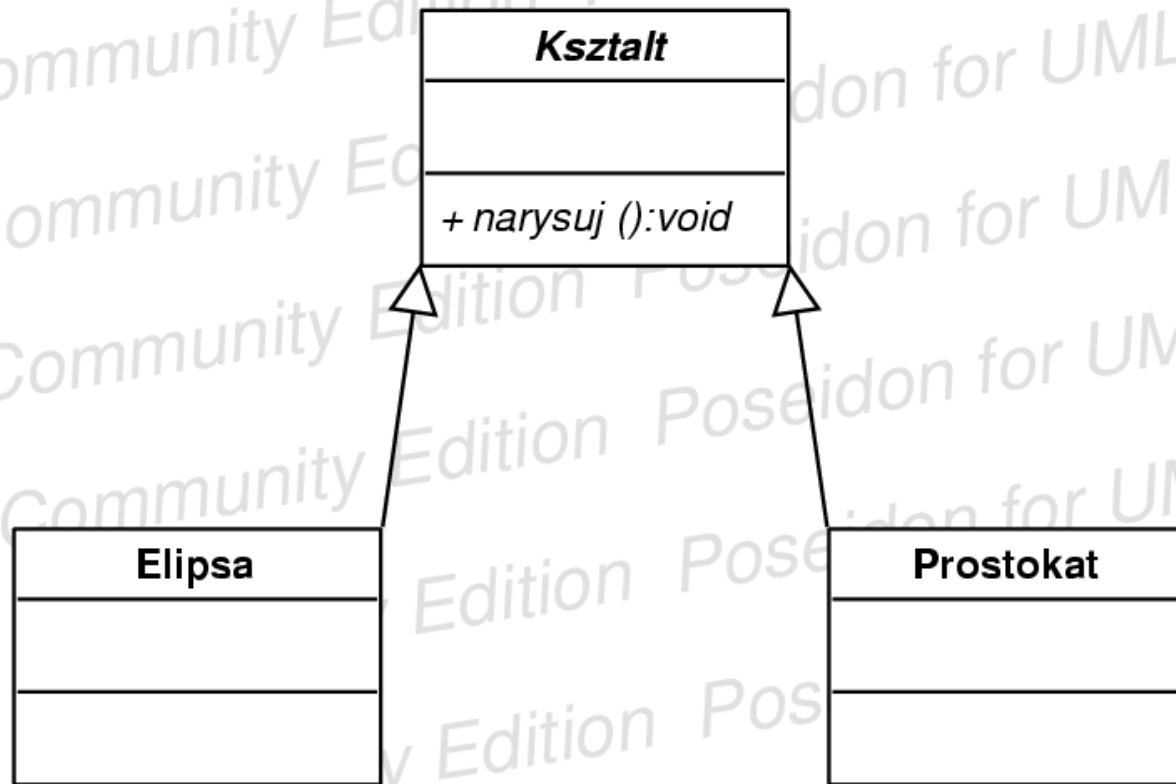
- Oba powody dziedziczenia można oczywiście mieszać i często dziedziczymy zarówno interfejs jak i implementację
- Ale te dwa podejścia są niezależne!
 - Można dziedziczyć implementację bez interfejsu choć jest to niewskazane
 - Takie dziedziczenie można zastąpić złożeniem
 - Można dziedziczyć sam interfejs. Takie klasy służące tylko do definicji interfejsu nazywamy klasami abstrakcyjnymi. W języku Java jest to osobne pojęcie

Dziedziczenie dla implementacji

Dziedziczenie tylko dla implementacji można zastąpić złożeniem



Interfejsy i klasy abstrakcyjne



Klasa abstrakcyjna nie posiada instancji obiektów bo niektóre metody są niezaimplementowane

Jeśli wszystkie metody są abstrakcyjne to klasę nazywamy interfejsem
Mówimy że jakaś klasa implementuje interfejs

Interfejs

- Specyfikacja interfejsu to kontrakt pomiędzy użytkownikiem danej klasy a jej projektantem
- Projektant zobowiązuje się dostarczać pewnych funkcji (metod)
- W zależności od klasy implementującej działanie tych funkcji może być różne (ale powinno dotyczyć tego samego!)
- Zmiana interfejsu to sprawa kosztowna (tak jak złamanie kontraktu)

Polimorfizm

- Jeśli jakaś zmienna może przechowywać obiekty różnych typów to jaką funkcja będzie wywołana przy wywołaniu danej metody zależy od typu obiektu aktualnie przechowywanego w tej zmiennej.
- Takie późne łączenie nazwy funkcji z jej treścią nazywamy polimorfizmem
- ```
Kształt *k;
Kształt *e = new Elipsa();
Kształt *p = new Prostokat();

k=e; k->narysuj();
k=p; k->narysuj();
```

# Zastosowanie polimorfizmu

```
while(NULL != (k=nextKształt()))
{
 k->narysuj();
}
```

A tak by to wyglądało nieobiektoowo

```
while(NULL != (k=nextKształt()))
{
 switch (k->type){
 case ELIPSA : narysujElipse(k); break;
 case PROSTOKAT : narysujProstokat(k); break;
 }
}
```

W tym wypadku implementujący musi znać wszystkie typy kształtów i dodanie dodatkowego kształtu wymaga zmiany kodu

# A jak to jest w C++ czyli paskudne szczegóły

- Wszystkie te techniki można oczywiście implementować w C++, ...
- ..., ale jak zwykle C++, zgodnie z filozofią zaczerpniętą z C pozostawia praktycznie pełną kontrolę programiście
- dlatego aby np. wykorzystać polimorfizm trzeba to sobie zażyczyć deklarując funkcje jako wirtualne
- dodatkowo polimorfizm będzie działał tylko wtedy jeśli do obiektu będziemy się odwoływać przez referencje albo wskaźnik
- “It is not a bug, it is a feature” :)

# Dziedziczenie w C++

```
#ifndef _types_
#define _types_
#include<iostream>
using namespace std;

class BaseType {
public:
virtual void aFunction() {cerr <<"BaseType"<<endl;}
};

class SubTypeA : public BaseType {
public:
 virtual void aFunction() { cerr << "AType"<<endl;}
};

class SubTypeB : public BaseType {
public:
 virtual void aFunction() { cerr << "BType"<<endl; }
};
#endif
```

```
#include "types.h"
main()
{
 BaseType t1,t2;
 SubTypeA a1,a2;
 SubTypeB b1,b2;

 t1=t2;
 a1=a2;

 t1=a1;
 t2=b2;

 a1=t2;
 b2=t1;

 a1=b2;
 b1=a2;
}
```

```
typing.cpp: In function `int main()':
typing.cpp:18: error: no match for 'operator=' in 'a1 = t2'
typing.cpp:13: error: candidates are: SubTypeA& SubTypeA::operator=
(const
 SubTypeA&)
typing.cpp:19: error: no match for 'operator=' in 'b2 = t1'
types.h:17: error: candidates are: SubTypeB& SubTypeB::operator=(const
 SubTypeB&)
typing.cpp:21: error: no match for 'operator=' in 'a1 = b2'
typing.cpp:13: error: candidates are: SubTypeA& SubTypeA::operator=
(const
 SubTypeA&)
typing.cpp:22: error: no match for 'operator=' in 'b1 = a2'
types.h:17: error: candidates are: SubTypeB& SubTypeB::operator=(const
 SubTypeB&)
```

# Zmienne, referencje i wskaźniki

- W C++ do obiektu możemy odwoływać się poprzez
  - “zwykłe” zmienne
    - zwykła zmienna to w zasadzie nazwa obszaru pamięci w której znajduje się obiekt
    - oznacza to w praktyce że jednej zmiennej odpowiada jeden i ten sam obiekt
  - wskaźniki
    - wskaźnik to zmienna zawierająca adres do obszaru pamięci w którym znajduje się obiekt
    - oczywiście wskaźnik może wskazywać na różne obiekty
  - referencje
    - referencja to alternatywna nazwa obszaru pamięci już zajętej przez obiekt (automatyczny wskaźnik)
    - w zasadzie używamy referencji tylko do przekazywania i zwracania argumentów funkcji



# Zmienne, wskaźniki, referencje



```
Obiekt x;
Obiekt *px;

Obiekt &rx=x;
```



```
x 000010
rx 000010
px 000f00
```

# Dziedziczenie i zmienne

```
#include "types.h"
main()
{
 BaseType type;
 SubTypeA a;
 SubTypeB b;

 type.aFunction
();
 a.aFunction();
 b.aFunction();

 type=a;
 type.aFunction
();
}
```

BaseType  
AType  
BType  
BaseType

To powoduje “okrojenie” obiektu a do obiektu typu b

# Dziedziczenie i wskaźniki

```
#include "types.h"

main()
{
 BaseType type;
 SubTypeA a;
 SubTypeB b;

 BaseType *ptype;

 ptype=&type;
 ptype->aFunction();

 ptype=&a;
 ptype->aFunction();

 ptype=&b;
 ptype->aFunction();
}
```

BaseType  
AType  
BType

# Dziedziczenie i referencje

```
#include "types.h"

void fref(BaseType &t) {t.aFunction(); };
void fval(BaseType t) {t.aFunction(); };

main()
{
 BaseType type;
 SubTypeA a;
 SubTypeB b;

 BaseType &refA = a;
 refA.aFunction();

 fval(a);
 fref(a);
 fref(b);
}
```

AType  
BaseType  
AType  
BType

# Polimorfizm w C++

- Aby korzystać z polimorfizmu w C++ musimy mieć dostęp do obiektu przez wskaźnik lub referencję
- oraz zadeklarować funkcje jako wirtualne
- Co oznacza że dość łatwo możemy uzyskać nie ten wynik o jaki nam chodzi :)
- Zachowanie polimorficzne jest naturalniejsze: to rodzaj typ obiektu powinien decydować jaka funkcja zostanie wywołana a nie typ wskaźnika
- Głównym przeciwskazaniem do użycia funkcji wirtualnych jest to że nie mogą być inline

# Klasy abstrakcyjne

- W podanych przykładach klasa dziedziczona (nadrzędna) sama posiadała implementację i można było tworzyć obiekty jej typu
- Często nie chcemy lub nie możemy podać implementacji klasy bazowej np w przypadku klasy kształt. Taka klasa nie może mieć żadnych instancji i nazywamy ją klasą abstrakcyjną
- Klasa jest abstrakcyjna jeśli posiada choć jedną nie zaimplementowaną metodę
- Klasa której wszystkie metody są abstrakcyjne nazywamy interfejsem

# Czyste funkcje wirtualne

```
class Stos {
 public:
 virtual int pop() = 0;
 virtual void push(int) = 0;
 virtual bool isEmpty() = 0;
};

class StosDyn : public {

}
```

Taka deklaracja stwierdza że klasa StosDyn musi dostarczać funkcji zdefiniowanych w klasie Stos (no chyba że sama jest abstrakcyjna)