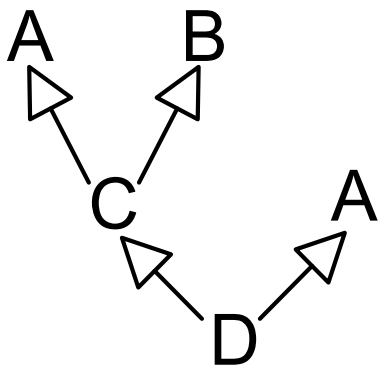
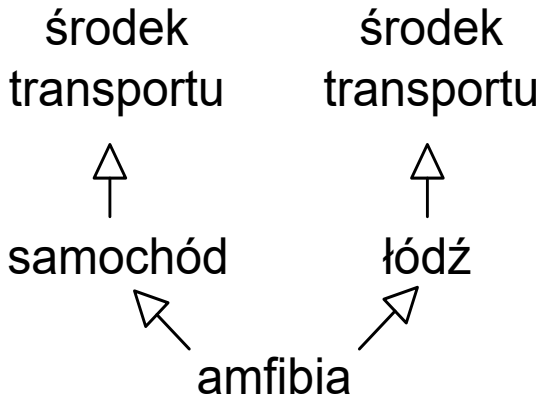
Wirtualne klasy bazowe

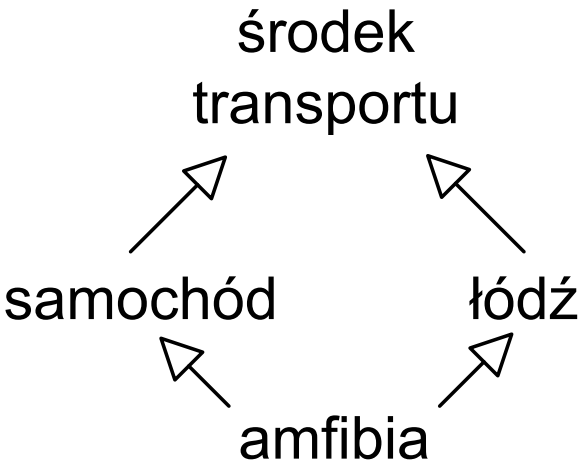
- Na liście bezpośrednich przodków dana klasa może pojawić się tylko i wyłącznie jeden raz
- Ale nic nie stoi na przeszkodzie, żeby klasa znalazła się wielokrotnie na wyższym poziomie dziedziczenia
  - W przypadku dziedziczenia zwykłego w klasie pochodnej dostaniemy zwielokrotnioną tą samą informację
- W przypadku drugiego grafu dostęp do składników nie jest jednoznaczny





Wirtualne klasy bazowe...

- Istnieje jednak stosunkowo proste rozwiązanie na duplikowanie informacji w klasie pochodnej
  - Podstawowa klasa wirtualna
- Słowo **`virtual`** pojawia się na liście dziedziczenia przed nazwą klasy
- Przy takim dziedziczeniu graf wygląda następująco

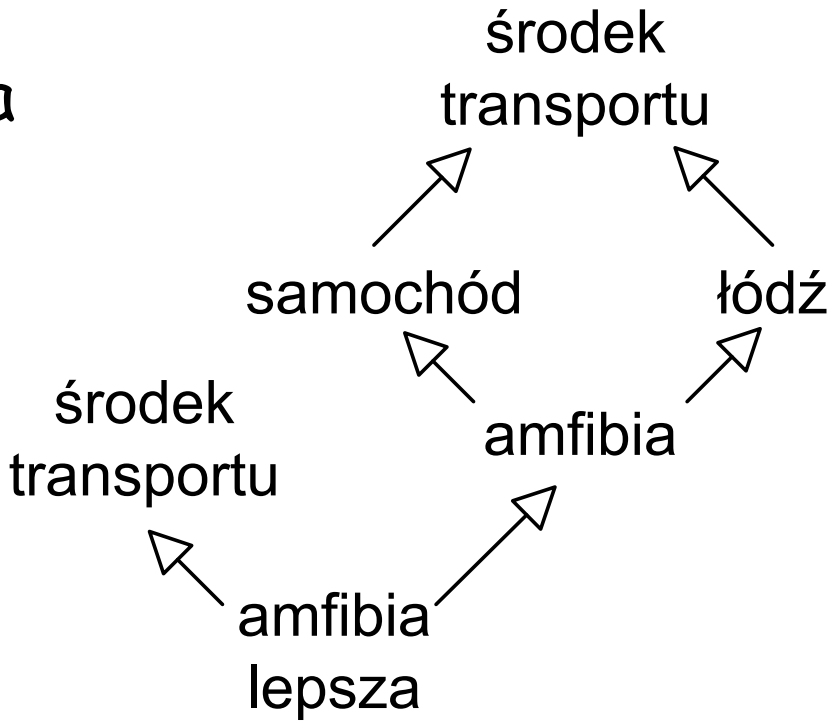


Wirtualne klasy bazowe...

- Deklarowanie dziedziczenia wirtualnego
  - ```
    class samochod: virtual public srodek_trans
    {...};
    class lodz: virtual public srodek_trans
    {...};
    class amfibia: public samochod, public lodz
    {...};
    ```
- Otrzymujemy zmniejszoną klasę amfibia bez duplikatów
- Nie ma ryzyka niejednoznaczności, mimo iż do składników możemy dostać się na dwa sposoby
- Przykład cpp_7.14
    ```

Dziedziczenie wirtualne i niewirtualne jednocześnie

- Ta sama klasa może być odziedziczona wirtualnie i niewirtualnie
- W pokazanym przykładzie **`amfibia_lepsza`** posiada
  - Wspólny zestaw składników dla wszystkich wirtualnych dziedziczeń
  - Oraz osobny zestaw odziedziczony w zwykły sposób



Klasa finalna

- Standard C++ nie zawiera słowa kluczowego **`final`**, które w łatwy sposób pozwala na stworzenie klasy po której nie będzie możliwe dalsze dziedziczenie
- Z pomocą wirtualnego dziedziczenia możliwe jest stworzenie klasy finalnej
- Należy wykorzystać to iż konstruktor klasy podstawowej wirtualnej wywoływany jest z klasy najbardziej pochodnej
- Co jeszcze jest niezbędne?

Konstrukcja i inicjalizacja w klasach wirtualnych

- Za konstrukcję wirtualnego dziedzictwa odpowiada klasa najbardziej pochodna
  - Klasa najbardziej pochodna to taka, która tworzy obiekt nie będący już pod-obiektem innego obiektu, czyli jest najniżej w hierarchii dziedziczenia
  - Stoi to poniekąd w sprzeczności z tym, że na liście inicjalizacyjnej stoi wywołanie konstruktora klasy bezpośrednio nadrzędnej
    - W takim przypadku kompilator i tak uruchomi konstrukcję obiektu odziedziczonego wirtualnie z klasy najbardziej pochodnej
- Przykład cpp_7.15

Kiedy dziedziczyć, a kiedy osadzać składniki (klasy)

- Dziedziczenie wybieramy w sytuacji kiedy dany obiekt jest rodzajem innego
  - Np. kwadrat jest rodzajem figury geometrycznej, samochód jest rodzajem pojazdu
- Zawieranie obiektów składowych używamy w sytuacji, gdy jeden obiekt składa się z innych obiektów
  - Np. samochód składa się (miedzy innymi) z czterech kół, radio składa się z tranzystorów
- Nie zawsze jest oczywiste, czy lepsze jest dziedziczenie, czy też może zawieranie

Aspekty dziedziczenia

- Klasa pochodna powinna przesłaniać tylko te funkcje, które zostały zadeklarowane jako wirtualne w klasie podstawowej
- Jeżeli klasa pochodna ma zostać klasą bazową to powinna także wszystkie funkcje, które mogą być przysłonięte deklarować jako wirtualne
- Przesłaniane funkcje powinny mieć te same domyślne wartości parametrów w klasie pochodnej, co w klasie podstawowej
- Klasa bazowa oczywiście powinna posiadać wirtualny destruktor
- Jedynie publiczne dziedziczenie określa relacje generalizacji. Pozostałe przypadki dziedziczenia umożliwiają tylko wykorzystanie już istniejącego kodu

Pułapki projektowania z użyciem dziedziczenia

- Struktura hierarchii klas stanowi jedną z fundamentalnych decyzji podejmowanych na etapie projektowania. Dlatego bardzo ważne jest popełnienie możliwe jak najmniejszej liczby błędów (a najlepiej w ogóle)
- Ograniczanie dziedziczenia
  - Jaki jest związek między klasą kwadrat i prostokąt?
    - W dziedziczeniu kwadrat nie jest prostokątem, ponieważ nie wszystkie operacje dostępne dla prostokąta można wykonać dla kwadratu (dodatkowy warunek na długość boków)
  - Rozwiązanie
    - Zignorowanie konsekwencji – odpowiedzialność za poprawność kodu spada na programistę
    - Eliminacja konsekwencji – np. wyświetlenie błędu, wyrzucenie wyjątku itp.
    - Eliminacja przyczyn błędu – inny projekt klas np. klasa bazowa opisująca czworokąty

Błędy projektowania

- Dziedziczenie zmieniające wartość
  - Np. ułamek zwykły (6/5) oraz ułamek zwykły z liczbą całkowitą (1 i 1/5)
    - Nowy ułamek wprowadza nową składową, ale zmienia senes odziedziczonej składowej
  - Stan składowych klasy reprezentujących pewną wartość nie może zmieniać się w klasie pochodnej
  - W tym przypadku powinnyśmy mówić o zawieraniu, a nie o dziedziczeniu
  - Ewentualnie odziedziczyć tak, żeby wartości nie uległy zmianie

Błędy projektowania...

- Dziedziczenie zmieniające interpretację wartości
  - Np. mamy klasę ułamek reprezentująca tylko ułamki dodatnie
  - Dziedziczymy tą klasę, aby stworzyć ułamki ze znakiem
  - W takiej sytuacji dodana nowa składowa zmienia interpretację wartości odziedziczonych składowych
  - Semantyka składowych klasy podstawowej nie może zmieniać się w klasie pochodnej

Dziedziczenie uwagi

- Nie każde określenie „bycia czymś" wyrażone zdaniem określa relacje dziedziczenia
- Dziedziczenia należy unikać gdy
  - Nie wszystkie dziedziczone operacje są przydatne
  - Zmienia się znaczenie dziedziczonych operacji
  - Zmienia się znaczenie dziedzicznych składowych
  - Właściwości składowych klasy bazowej zostają zawężone w klasie pochodnej
- Przypisanie dowolnego obiektu klasy pochodnej obiektowi klasy podstawowej zawsze musi mieć sens (opuszczenie dodatkowych składowych zawartych w klasie pochodnej)

Szablony

- W językach programowanie takich jak C++ gdzie istnieje ścisła kontrola typów często występuję potrzeba wielokrotnego zdefiniowania takiej samej funkcji, ale pracującej na różnych typach danych
- Rozwiązaniem jest wykorzystanie makrodefinicji znanych z języka C
  - Mechaniczne podstawianie, które może stwarzać problemy
  - Nie zalecane!!!
- Dlatego w języku C++ wprowadzono szablony, które rozwiązują większość problemów
  - Mają też swoje wady (o tym później)

Makrodefinicje

- Do generowania „funkcji" wykonujących to samo zadanie na różnych typach danych w języku C można było wykorzystywać makrodefinicje
  - `#define max(a, b) (((a) < (b)) ? (b) : (a))`
- Jednak użycie makrodefinicji może spowodować duże problemy
  - W szczególności kiedy argumentami nie są liczby ani zmienne, ale wyrażenia
    - `max(a++, b++);`
  - Ponieważ rozwinięcie **`max`** daje rezultat
    - `(((a++) < (b++)) ? (b++) : (a++))`

Szablony

- Szablony reprezentują funkcje, a nawet typy danych tworzone przez programistów (klasy)
  - Ale same nie są funkcjami ani klasami
- Nie zostają one zaimplementowane dla określonego typu danych, ponieważ zostanie on zdefiniowany później
  - W większości sytuacji parametryzowane są typem, ale nie jest to reguła
- Aby użyć szablonu kompilator lub programista musi określić dla jakiego typu ma on zostać użyty

Szablony klas wykorzystanie - tablica

std::array

Defined in header <array>

```
template<
    class T,
    std::size_t N
> struct array;
```
(since C++11)

https://en.cppreference.com/w/cpp/container/array

- Prosta tablica statyczna
  - Alokacja na stosie
  - Elementy określonego typu (możliwe konwersje)
  - Znany rozmiar czasie kompilacji
  - Zamiennik zwykłej tablicy
- Przykład cpp9.0a

Szablony klas wykorzystanie - wektor

std::vector

Defined in header <vector>

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;                                                    (1)

namespace pmr {
    template <class T>
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;   (2)  (since C++17)
}
```

1) std::vector is a sequence container that encapsulates dynamic size arrays.

2) std::pmr::vector is an alias template that uses a polymorphic allocator

https://en.cppreference.com/w/cpp/container/vector

- Dynamiczna tablica
  - Alokacja pamięci na stercie
  - Elementy określonego typu (możliwe konwersje)
  - Ciągły obszar pamięci
  - Rozmiar rośnie w miarę potrzeb – UWAGA
- Przykład cpp9.0b

Szablony klas wykorzystanie - string

std::basic_string

Defined in header <string>

```
template<
    class CharT,
    class Traits = std::char_traits<CharT>,                          (1)
    class Allocator = std::allocator<CharT>
> class basic_string;
```

```
namespace pmr {
    template <class CharT, class Traits = std::char_traits<CharT>>
    using basic_string = std::basic_string< CharT, Traits,           (2)  (since C++17)
                                            std::polymorphic_allocator<CharT>>
}
```

- Dynamiczna tablica znaków
  - **`std::string`** to jest **`std::basic_string<char>`**
  - Alokacja pamięci na stercie
  - Elementy określonego typu **`char`**
  - Ciągły obszar pamięci
- Przykład cpp9.0c

Szablony funkcji wykorzystanie - find

std::find

Defined in header <algorithm>

```
template< class InputIt, class T >
constexpr InputIt find( InputIt first, InputIt last, const T& value );
```

https://en.cppreference.com/w/cpp/algorithm/find

- Algorytm do wyszukiwania
  - Obsługuje dowolne typy
  - Znajduje pierwszy element zgodny ze wzorcem
  - Przeszukuje podany zakres - nie musi być cały kontener
  - Są też inne wersje np. find_if
- Przykład cpp9.0d

Szablony funkcji wykorzystanie - sort

std::sort

```
template< class RandomIt >
constexpr void sort( RandomIt first, RandomIt last );
template< class RandomIt, class Compare >
void sort( RandomIt first, RandomIt last, Compare comp );
```

https://en.cppreference.com/w/cpp/algorithm/sort

- Algorytm do sortowania
  - Obsługuje dowolne typy
  - Domyślnie sortuje używając operatora <
  - W wersji drugiej potrafi użyć obiektu funkcyjnego służącego jako narzędzie do porównywania
- Przykład cpp9.0e

Szablony funkcji wykorzystanie – for_each

std::for_each

```
template< class InputIt, class UnaryFunction >
UnaryFunction for_each( InputIt first, InputIt last, UnaryFunction f );
```

https://en.cppreference.com/w/cpp/algorithm/for_each

- Algorytm do wykonywania operacji na elementach
  - Stosowany zamienne z zakresową pętlą for
  - Działa w trybie tylko do odczytu albo modyfikowania
  - Przyjmuje jako argumenty zakres oraz funkcję/funktor
- Przykład cpp9.0f

Szablony i STL - inne (podstawowe) ciekawostki

- `std::pair`
- `std::tuple`
- `std::unique_ptr`
- `std::shared_ptr`
- `std::less`
- `std::greater`
- `std::bind`
- `std::ref, std::cref`
- `std::initializer_list`
- …