Maciej Sobieraj

Wykład 5

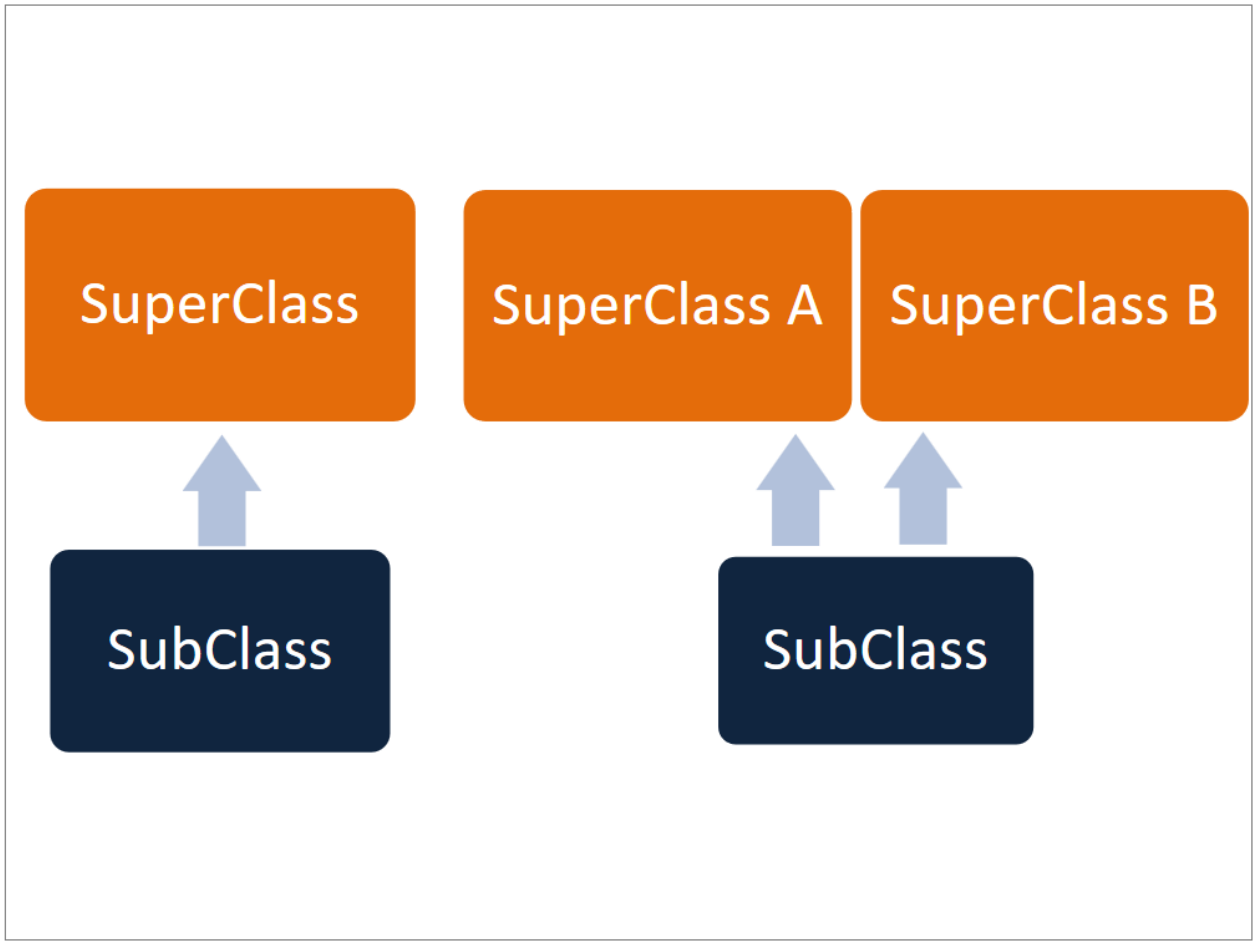
Outline

1. **Dziedziczenie**
    1. **Definiowanie hierarchii klas**
    2. Klasy, dziedziczenie i zgodność typów
    3. Polimorfizm i metody wirtualne
    4. Obiekty jako parametry i dynamiczne rzutowanie

Definiowanie prostej podklasy

- Możemy użyć każdej klasy jako podstawy (lub fundamentu) do zdefiniowania lub zbudowania innej klasy (**podklasy**).
- Możliwe jest także użycie **więcej niż jednej klasy do zdefiniowania podklasy**.
- Możemy również napisać o nadklasach jako klasach bazowych, a podklasach jako klasach **pochodnych**.

Definiowanie prostej podklasy



Definiowanie prostej podklasy

- Ta klasa będzie służyć jako **klasa bazowa**

```
#include <iostream>
using namespace std;
class Super {
private:
	int storage;
public:
	void put(int val) { storage = val; }
	int get(void) { return storage; }
};
int main(void) {
	Super object;

	object.put(100);
	object.put(object.get() + 1);
	cout << object.get() << endl;
	return 0;
}
```

Definiowanie prostej podklasy

- Jeśli chcemy zdefiniować klasę o nazwie ***Y*** jako **podklasę klasy bazowej** o nazwie ***X***, używamy następującej składni:

```
class Y : {visibility specifier} X { ... };
```

- Jeśli istnieje więcej niż jedna superklasa, musimy zarejestrować wszystkie przy użyciu przecinków jako separatorów:
  - **class** A : X, Y, Z { … };

Definiowanie prostej podklasy

- Klasa ***Sub*** nie wprowadza żadnych nowych zmiennych ani nowych funkcji.
- Kiedy **pominiemy specyfikator widoczności**, kompilator zakłada, że mamy zamiar zastosować "**prywatne dziedziczenie**".

```
class Sub : Super {
};

int main(void) {
      Sub object;

      object.put(100);
      object.put(object.get() + 1);
      cout << object.get() << endl;
      return 0;
}
```

Definiowanie prostej podklasy

- "Dziedziczenie prywatne" oznacza, że wszystkie **publiczne komponenty** nadklasy będą posiadały **dostęp prywatny**, a **prywatne składniki** nadklasy **nie będą dostępne w ogóle**.
- Musimy powiedzieć kompilatorowi, że chcemy **zachować poprzednio używaną politykę dostępu**. Robimy to za pomocą "publicznego" specyfikatora widoczności :
  - **class** Sub : **public** Super {
  - };

Definiowanie prostej podklasy

- Subklasa **straciła dostęp do prywatnych składników superklasy**.
- Nie możemy zapisać funkcji składowej klasy ***Sub***, która byłaby w stanie bezpośrednio manipulować zmienną ***storage***.

Definiowanie prostej podklasy

```
#include <iostream>

using namespace std;

class Super {
private:
	int storage;
public:
	void put(int val) { storage = val; }
	int get(void) { return storage; }
};

class Sub : public Super {
};

int main(void) {
	Sub object;

	object.put(100);
	object.put(object.get() + 1);
	cout << object.get() << endl;
	return 0;
}
```

Definiowanie prostej podklasy

- Słowo kluczowe ***protected*** oznacza, że każdy oznaczony nim komponent **zachowuje się jak składnik publiczny, gdy jest używany przez którąkolwiek z podklas i wygląda jak element prywatny dla pozostałych**.

Definiowanie prostej podklasy

```
#include <iostream>

using namespace std;

class Super {
protected:
	int storage;
public:
	void put(int val) { storage = val; }
	int get(void) { return storage; }
};

class Sub : public Super {
public:
	void print(void) { cout << "storage = " << storage << endl; }
};

int main(void) {
	Sub object;

	object.put(100);
	object.put(object.get() + 1);
	object.print();
	return 0;
}
```

Definiowanie prostej podklasy

When the component is declared as:	When the class is inherited as:	The resulting access inside the subclass is:
public	public	Public
protected		protected
private		none
public	protected	protected
protected		protected
private		none
public	private	private
protected		private
private		none

Definiowanie prostej podklasy

```
#include <iostream>
using namespace std;
class SuperA {
protected:
    int storage;
public:
    void put(int val) { storage = val; }
    int get(void) { return storage; }
};
class SuperB {
protected:
    int safe;
public:
    void insert(int val) { safe = val; }
    int takeout(void) { return safe; }
};
class Sub : public SuperA, public SuperB {
public:
    void print(void) {
        cout << "storage = " << storage << endl;
        cout << "safe    = " << safe << endl;
    }
};
int main(void) {
    Sub object;

    object.put(1);      object.insert(2);
    object.put(object.get() + object.takeout());
    object.insert(object.get() + object.takeout());
    object.print();
    return 0;
}
```

Outline

1. **Dziedziczenie**
    1. Definiowanie hierarchii klas
    2. **Klasy, dziedziczenie i zgodność typów**
    3. Polimorfizm i metody wirtualne
    4. Obiekty jako parametry i dynamiczne rzutowanie

Kompatybilność typów - najprostszy przypadek

- **Każda nowa klasa stanowi nowy typ danych**. Każdy obiekt skonstruowany na podstawie takiej klasy jest jak wartość nowego typu.
- Oznacza to, że **dowolne dwa obiekty mogą (lub nie muszą) być kompatybilne** w sensie ich typów.

Kompatybilność typów - najprostszy przypadek

```
#include <iostream>
using namespace std;
class Cat {
public:
    void MakeSound(void) { cout << "Meow! Meow!" << endl; }
};
class Dog {
public:
    void MakeSound(void) { cout << "Woof! Woof!" << endl; }
};
int main(void) {
    Cat *a_cat = new Cat();
    Dog *a_dog = new Dog();
    a_cat -> MakeSound();
    a_dog -> MakeSound();
    return 0;
}
```

Kompatybilność typów - najprostszy przypadek

```
#include <iostream>
using namespace std;
class Cat {
public:
    void MakeSound(void) { cout << "Meow! Meow!" << endl; }
};
class Dog {
public:
    void MakeSound(void) { cout << "Woof! Woof!" << endl; }
};
int main(void) {
    Cat *a_cat = new Cat();
    Dog *a_dog = new Dog();
    a_cat -> MakeSound();
    a_dog -> MakeSound();
    return 0;
}
```

Program wyświetli wynik:

Meow! Meow!

Woof! Woof!

Kompatybilność typów - najprostszy przypadek

- Obiekty klasy ***Cat*** nie są kompatybilne z obiektami klasy ***Dog***, chociaż struktura obu klas jest identyczna. Żadne z poniższych przypisań nie jest poprawne i oba z nich spowodują **błąd kompilatora**:
  - a_dog = a_cat;
  - a_cat = a_dog;
- **Obiekty pochodzące z klas leżących w różnych gałęziach drzewa dziedziczenia są zawsze niekompatybilne**.

Kompatybilność typów - najprostszy przypadek

- Klasy ***Dog*** oraz ***Cat*** są teraz potomkami (a dokładniej, dziećmi) wspólnej klasy bazowej ***Pet***.
- Wyposażyliśmy również wszystkie klasy w **konstruktory**.
  Nasze zwierzęta mogą również biegać.

Kompatybilność typów - najprostszy przypadek

- Podsumujmy to, co stworzyliśmy.
  - obiekty pochodzące z klasy ***Pet*** są w stanie biegać
  - obiekty wywodzące się z klas ***Dog*** i ***Cat*** są w stanie biegać (dziedziczą tę umiejętność z nadklasy); mogą również wydawać dźwięki (zwróć uwagę, że ta umiejętność nie jest dostępna dla obiektów klasy ***Pet***)
- A więc:
  - Obiekty ***Cat*** and ***Dog*** mogą robić wszystkie rzeczy, które mogą zrobić zwierzęta (***Pet***)
    Zwierzęta (***Pet***) nie mogą zrobić wszystkiego, co potrafi kot (***Cat***) i pies (***Dog***)

Kompatybilność typów - najprostszy przypadek

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected:
	string Name;
public:
	Pet(string n) { Name = n; }
	void Run(void) { cout << Name << ": I'm running" << endl; }
};
class Dog : public Pet {
public:
	Dog(string n) : Pet(n) {};
	void MakeSound(void) { cout << Name << ": Woof! Woof!" << endl; }
};
class Cat : public Pet {
public:
	Cat(string n) : Pet(n) {};
	void MakeSound(void) { cout << Name << ": Meow! Meow!" << endl; }
};
int main(void) {
	Pet a_pet("pet");
	Cat a_cat("Tom");
	Dog a_dog("Spike");
	a_pet.Run();
	a_dog.Run(); a_dog.MakeSound();
	a_cat.Run(); a_cat.MakeSound();
	return 0;
}
```

Kompatybilność typów - najprostszy przypadek

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected:
	string Name;
public:
	Pet(string n) { Name = n; }
	void Run(void) { cout << Name << ": I'm running" << endl; }
};
class Dog : public Pet {
public:
	Dog(string n) : Pet(n) {};
	void MakeSound(void) { cout << Name << ": Woof! Woof!" << endl; }
};
class Cat : public Pet {
public:
	Cat(string n) : Pet(n) {};
	void MakeSound(void) { cout << Name << ": Meow! Meow!" << endl; }
};
int main(void) {
	Pet a_pet("pet");
	Cat a_cat("Tom");
	Dog a_dog("Spike");
	a_pet.Run();
	a_dog.Run(); a_dog.MakeSound();
	a_cat.Run(); a_cat.MakeSound();
	return 0;
}
```

Program wyświetli:

pet: I'm running
Spike: I'm running
Spike: Woof! Woof!
Tom: I'm running
Tom: Meow! Meow!

Kompatybilność typów - najprostszy przypadek

- Możemy się domyślać, że:
  - **obiekty podklasy mają co najmniej te same możliwości, co obiekty nadklasy**
  - **obiekty superklasy mogą nie mieć takich samych możliwości jak obiekty podklasy**
- To prowadzi do następującego wniosku:
  - **obiekty superklasy są kompatybilne z obiektami podklasy**
  - **obiekty podklasy nie są kompatybilne z obiektami nadklasy**

Kompatybilność typów - najprostszy przypadek

- Oznacza to, że:
  - **Możemy** zrobić następującą rzecz:
    - a_pet = **new** Dog("Huckleberry");
    - a_pet -> Run();
  - Ale **nie możemy** zrobić tego:
    - ~~a_pet -> MakeSound();~~
  - ponieważ Zwierzęta (***Pets***) nie wiedzą, jak wydawać dźwięki (przynajmniej w naszym świecie zajęć)
  - **nie wolno** ci wykonywać następujących czynności:
    - ~~a_dog = new Pet("Strange pet");~~

Kompatybilność typów - najprostszy przypadek

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected:
    string Name;
public:
    Pet(string n) { Name = n; }
    void Run(void) { cout << Name << ": I'm running" << endl; }
};
class Dog : public Pet {
public:
    Dog(string n) : Pet(n) {};
    void MakeSound(void) { cout << Name << ": Woof! Woof!" << endl; }
};
class Cat : public Pet {
public:
    Cat(string n) : Pet(n) {};
    void MakeSound(void) { cout << Name << ": Meow! Meow!" << endl; }
};
int main(void) {
    Pet *a_pet1 = new Cat("Tom");
    Pet *a_pet2 = new Dog("Spike");

    a_pet1 -> Run();
    // 'a_pet1 -> MakeSound();' is not allowed here!
    a_pet2 -> Run();
    // 'a_pet2 -> MakeSound();' is not allowed here!

    return 0;
}
```

Program wyświetli:

Tom: I'm running

Spike: I'm running

Kompatybilność typów - najprostszy przypadek

- Dlaczego nie wolno nam rozkazywać naszemu zwierzakowi, aby wydał dźwięk?
- Problem pochodzi z **testów statycznych wykonanych przez kompilator** podczas kompilacji.
- Kompilator jest przekonany, że zwierzęta nie mogą wydawać dźwięków i nie pozwoli nam nawet tego spróbować.

Kompatybilność typów - najprostszy przypadek

- Możemy to zrobić za pomocą operatorów rzutowania:

```
static_cast<target_type>(an_expression)
```

  - static_cast<Dog *>(a_pet)
- zmusza kompilator do założenia, że ***a_pet*** jest (tymczasowo) przekonwertowana na wskaźnik typu ***Dog ****.

Kompatybilność typów - najprostszy przypadek

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected:
	string Name;
public:
	Pet(string n) { Name = n; }
	void Run(void) { cout << Name << ": I'm running" << endl; }
};
class Dog : public Pet {
public:
	Dog(string n) : Pet(n) {};
	void MakeSound(void) { cout << Name << ": Woof! Woof!" << endl; }
};
class Cat : public Pet {
public:
	Cat(string n) : Pet(n) {};
	void MakeSound(void) { cout << Name << ": Meow! Meow!" << endl; }
};
int main(void) {
	Pet *a_pet1 = new Cat("Tom");
	Pet *a_pet2 = new Dog("Spike");
	a_pet1 -> Run();
	static_cast<Cat *>(a_pet1) -> MakeSound();
	a_pet2 -> Run();
	static_cast<Dog *>(a_pet2) -> MakeSound();
	return 0;
}
```

Kompatybilność typów - najprostszy przypadek

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected:
	string Name;
public:
	Pet(string n) { Name = n; }
	void Run(void) { cout << Name << ": I'm running" << endl; }
};
class Dog : public Pet {
public:
	Dog(string n) : Pet(n) {};
	void MakeSound(void) { cout << Name << ": Woof! Woof!" << endl; }
};
class Cat : public Pet {
public:
	Cat(string n) : Pet(n) {};
	void MakeSound(void) { cout << Name << ": Meow! Meow!" << endl; }
};
int main(void) {
	Pet *a_pet1 = new Cat("Tom");
	Pet *a_pet2 = new Dog("Spike");
	a_pet1 -> Run();
	static_cast<Cat *>(a_pet1) -> MakeSound();
	a_pet2 -> Run();
	static_cast<Dog *>(a_pet2) -> MakeSound();
	return 0;
}
```

Program wyświetli:

Tom: I'm running
Tom: Meow! Meow!
Spike: I'm running
Spike: Woof! Woof!

Kompatybilność typów - najprostszy przypadek

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected:
	string Name;
public:
	Pet(string n) { Name = n; }
	void Run(void) { cout << Name << ": I'm running" << endl; }
};
class Dog : public Pet {
public:
	Dog(string n) : Pet(n) {};
	void MakeSound(void) { cout << Name << ": Woof! Woof!" << endl; }
};
class Cat : public Pet {
public:
	Cat(string n) : Pet(n) {};
	void MakeSound(void) { cout << Name << ": Meow! Meow!" << endl; }
};
int main(void) {
	Pet *a_pet1 = new Cat("Tom");
	Pet *a_pet2 = new Dog("Spike");
	a_pet2 -> Run();
	static_cast<Cat *>(a_pet2) -> MakeSound();
	a_pet1 -> Run();
	static_cast<Dog *>(a_pet1) -> MakeSound();
	return 0;
}
```

Kompatybilność typów - najprostszy przypadek

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected:
	string Name;
public:
	Pet(string n) { Name = n; }
	void Run(void) { cout << Name << ": I'm running" << endl; }
};
class Dog : public Pet {
public:
	Dog(string n) : Pet(n) {};
	void MakeSound(void) { cout << Name << ": Woof! Woof!" << endl; }
};
class Cat : public Pet {
public:
	Cat(string n) : Pet(n) {};
	void MakeSound(void) { cout << Name << ": Meow! Meow!" << endl; }
};
int main(void) {
	Pet *a_pet1 = new Cat("Tom");
	Pet *a_pet2 = new Dog("Spike");
	a_pet2 -> Run();
	static_cast<Cat *>(a_pet2) -> MakeSound();
	a_pet1 -> Run();
	static_cast<Dog *>(a_pet1) -> MakeSound();
	return 0;
}
```

Program wyświetli:

Spike: I'm running
Spike: Meow! Meow!
Tom: I'm running
Tom: Woof! Woof!

Kompatybilność typów - najprostszy przypadek

- Kompilator **nie może sprawdzić, czy konwertowany wskaźnik jest zgodny z obiektem, na który wskazuje**.
- **Pełna weryfikacja ważności wskaźnika jest możliwa wtedy i tylko wtedy, gdy program jest wykonywany**
- Język "C ++" ma drugi operator konwersji zaprojektowany specjalnie dla tego przypadku.
- Jego nazwa jest nieco sugestywna: ***dynamic_cast***.

Kompatybilność typów – końcowy przypadek

- Reguła stwierdzająca, że **obiekty leżące na wyższych poziomach są zgodne z obiektami na niższych poziomach** hierarchii klas, działa nawet wtedy, gdy **łańcuch dziedziczenia jest arbitralnie długi.**

Kompatybilność typów – końcowy przypadek

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected:
    string Name;
public:
    Pet(string n) { Name = n; }
    void Run(void) { cout << Name << ": I'm running" << endl; }
};
class Cat : public Pet {
public:
    Cat(string n) : Pet(n) {};
    void MakeSound(void) { cout << Name << ": Meow! Meow!" << endl; }
};
class Persian : public Cat {
public:
    Persian(string n) : Cat(n) {};
};
int main(void) {
    Pet  *a_pet;
    Persian *a_persian;
    a_pet = a_persian = new Persian("Mr. Bigglesworth");
    a_persian -> MakeSound();
    static_cast<Persian *>(a_pet) -> MakeSound();
    return 0;
}
```

Kompatybilność typów – końcowy przypadek

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected:
    string Name;
public:
    Pet(string n) { Name = n; }
    void Run(void) { cout << Name << ": I'm running" << endl; }
};
class Cat : public Pet {
public:
    Cat(string n) : Pet(n) {};
    void MakeSound(void) { cout << Name << ": Meow! Meow!" << endl; }
};
class Persian : public Cat {
public:
    Persian(string n) : Cat(n) {};
};
int main(void) {
    Pet  *a_pet;
    Persian *a_persian;
    a_pet = a_persian = new Persian("Mr. Bigglesworth");
    a_persian -> MakeSound();
    static_cast<Persian *>(a_pet) -> MakeSound();
    return 0;
}
```

Program wyświetli:

Mr. Bigglesworth: Meow! Meow!
Mr. Bigglesworth: Meow! Meow!

Outline

1. **Dziedziczenie**
    1. Definiowanie hierarchii klas
    2. Klasy, dziedziczenie i zgodność typów
    3. **Polimorfizm i metody wirtualne**
    4. Obiekty jako parametry i dynamiczne rzutowanie

Przesłanianie metody w podklasie

- Kiedy podklasa **deklaruje metodę o nazwie poprzednio znanej w jej nadklasie, oryginalna metoda jest nadpisywana**.
- Efekty nadpisania mogą zostać odwrócone (lub unieważnione), jeśli używasz operatora ***static_cast*** w odwrotnej kolejności.

Przesłanianie metody w podklasie

```
#include <iostream>
using namespace std;
class Pet {
protected:
    string Name;
public:
    Pet(string n) { Name = n; }
    void MakeSound(void) { cout << Name << " the Pet says: Shh! Shh!" << endl; }
};
class Cat : public Pet {
public:
    Cat(string n) : Pet(n) { }
    void MakeSound(void) { cout << Name << " the Cat says: Meow! Meow!" << endl; }
};
class Dog : public Pet {
public:
    Dog(string n) : Pet(n) { }
    void MakeSound(void) { cout << Name << " the Dog says: Woof! Woof!" << endl; }
};
int main(void) {
    Cat *a_cat;
    Dog *a_dog;

    a_cat = new Cat("Kitty");
    a_dog = new Dog("Doggie");
    a_cat -> MakeSound();
    static_cast<Pet *>(a_cat) -> MakeSound();
    a_dog -> MakeSound();
    static_cast<Pet *>(a_dog) -> MakeSound();
    return 0;
}
```

Przesłanianie metody w podklasie

```
#include <iostream>
using namespace std;
class Pet {
protected:
    string Name;
public:
    Pet(string n) { Name = n; }
    void MakeSound(void) { cout << Name << " the Pet says: Shh! Shh!" << endl; }
};
class Cat : public Pet {
public:
    Cat(string n) : Pet(n) { }
    void MakeSound(void) { cout << Name << " the Cat says: Meow! Meow!" << endl; }
};
class Dog : public Pet {
public:
    Dog(string n) : Pet(n) { }
    void MakeSound(void) { cout << Name << " the Dog says: Woof! Woof!" << endl; }
};
int main(void) {
    Cat *a_cat;
    Dog *a_dog;

    a_cat = new Cat("Kitty");
    a_dog = new Dog("Doggie");
    a_cat -> MakeSound();
    static_cast<Pet *>(a_cat) -> MakeSound();
    a_dog -> MakeSound();
    static_cast<Pet *>(a_dog) -> MakeSound();
    return 0;
}
```

Program wyświetli:

Kitty the Cat says: Meow! Meow!
Kitty the Pet says: Shh! Shh!
Doggie the Dog says: Woof! Woof!
Doggie the Pet says: Shh! Shh!

Przesłanianie metody w podklasie

```
#include <iostream>
using namespace std;
class Pet {
protected:
	string Name;
public:
	Pet(string n) { Name = n; }
	void MakeSound(void) { cout << Name << " the Pet says: Shh! Shh!" << endl; }
};
class Cat : public Pet {
public:
	Cat(string n) : Pet(n) { }
	void MakeSound(void) { cout << Name << " the Cat says: Meow! Meow!" << endl; }
};
class Dog : public Pet {
public:
	Dog(string n) : Pet(n) { }
	void MakeSound(void) { cout << Name << " the Dog says: Woof! Woof!" << endl; }
};
int main(void) {
	Pet *a_pet1, *a_pet2;
	Cat *a_cat;
	Dog *a_dog;

	a_pet1 = a_cat = new Cat("Kitty");
	a_pet2 = a_dog = new Dog("Doggie");
	a_pet1 -> MakeSound();
	a_cat  -> MakeSound();
	a_pet2 -> MakeSound();
	a_dog  -> MakeSound();
	return 0;
}
```

Przesłanianie metody w podklasie

```
#include <iostream>
using namespace std;
class Pet {
protected:
	string Name;
public:
	Pet(string n) { Name = n; }
	void MakeSound(void) { cout << Name << " the Pet says: Shh! Shh!" << endl; }
};
class Cat : public Pet {
public:
	Cat(string n) : Pet(n) { }
	void MakeSound(void) { cout << Name << " the Cat says: Meow! Meow!" << endl; }
};
class Dog : public Pet {
public:
	Dog(string n) : Pet(n) { }
	void MakeSound(void) { cout << Name << " the Dog says: Woof! Woof!" << endl; }
};
int main(void) {
	Pet *a_pet1, *a_pet2;
	Cat *a_cat;
	Dog *a_dog;

	a_pet1 = a_cat = new Cat("Kitty");
	a_pet2 = a_dog = new Dog("Doggie");
	a_pet1 -> MakeSound();
	a_cat  -> MakeSound();
	a_pet2 -> MakeSound();
	a_dog  -> MakeSound();
	return 0;
}
```

Program wyświetli:

Kitty the Pet says: Shh! Shh!
Kitty the Cat says: Meow! Meow!
Doggie the Pet says: Shh! Shh!
Doggie the Dog says: Woof! Woof!

Przesłanianie metody w podklasie

- **Polimorfizm jest metodą wykorzystywaną aby na nowo zdefiniować zachowanie superklasy** (ale tylko tej, która wyraźnie zgadza się być traktowana w ten sposób!) **Bez zmiany jej implementacji**.
- Słowo "**polimorfizm**" oznacza, że jedna i ta sama klasa może pokazywać wiele ("poly" - jak w "poligamii") form ("morphs") nie definiowanych przez samą klasę, ale **przez jej podklasy.**

Przesłanianie metody w podklasie

- Słowo ***virtual*** oznacza, że metoda zostanie ponownie **zdefiniowana** (zastąpiona) na poziomie oryginalnej klasy.

Przesłanianie metody w podklasie

```
#include <iostream>
using namespace std;
class Pet {
protected:
    string Name;
public:
    Pet(string n) { Name = n; }
    virtual void MakeSound(void) { cout << Name << " the Pet says: Shh! Shh!" << endl; }
};
class Cat : public Pet {
public:
    Cat(string n) : Pet(n) { }
    void MakeSound(void) { cout << Name << " the Cat says: Meow! Meow!" << endl; }
};
class Dog : public Pet {
public:
    Dog(string n) : Pet(n) { }
    void MakeSound(void) { cout << Name << " the Dog says: Woof! Woof!" << endl; }
};
int main(void) {
    Pet *a_pet1, *a_pet2;
    Cat *a_cat;
    Dog *a_dog;

    a_pet1 = a_cat = new Cat("Kitty");
    a_pet2 = a_dog = new Dog("Doggie");
    a_pet1 -> MakeSound();
    a_cat  -> MakeSound();
    static_cast<Pet *>(a_cat) -> MakeSound();
    a_pet2 -> MakeSound();
    a_dog  -> MakeSound();
    static_cast<Pet *>(a_dog) -> MakeSound();
    return 0;
}
```

Przesłanianie metody w podklasie

```
#include <iostream>
using namespace std;
class Pet {
protected:
    string Name;
public:
    Pet(string n) { Name = n; }
    virtual void MakeSound(void) { cout << Name << " the Pet says: Shh! Shh!" << endl; }
};
class Cat : public Pet {
public:
    Cat(string n) : Pet(n) { }
    void MakeSound(void) { cout << Name << " the Cat says: Meow! Meow!" << endl; }
};
class Dog : public Pet {
public:
    Dog(string n) : Pet(n) { }
    void MakeSound(void) { cout << Name << " the Dog says: Woof! Woof!" << endl; }
};
int main(void) {
    Pet *a_pet1, *a_pet2;
    Cat *a_cat;
    Dog *a_dog;

    a_pet1 = a_cat = new Cat("Kitty");
    a_pet2 = a_dog = new Dog("Doggie");
    a_pet1 -> MakeSound();
    a_cat  -> MakeSound();
    static_cast<Pet *>(a_cat) -> MakeSound();
    a_pet2 -> MakeSound();
    a_dog  -> MakeSound();
    static_cast<Pet *>(a_dog) -> MakeSound();
    return 0;
}
```

Program wyświetli:

Kitty the Cat says: Meow! Meow!
Kitty the Cat says: Meow! Meow!
Kitty the Cat says: Meow! Meow!
Doggie the Dog says: Woof! Woof!
Doggie the Dog says: Woof! Woof!
Doggie the Dog says: Woof! Woof!

Przesłanianie metody w podklasie

```
#include <iostream>
using namespace std;
class Pet {
protected:
    string Name;
public:
    Pet(string n) { Name = n; MakeSound(); }
    virtual void MakeSound(void) { cout << Name << " the Pet says: Shh! Shh!" << endl; }
};
class Cat : public Pet {
public:
    Cat(string n) : Pet(n) { }
    void MakeSound(void) { cout << Name << " the Cat says: Meow! Meow!" << endl; }
};
class Dog : public Pet {
public:
    Dog(string n) : Pet(n) { }
    void MakeSound(void) { cout << Name << " the Dog says: Woof! Woof!" << endl; }
};
int main(void) {
    Cat *a_cat;
    Dog *a_dog;

    a_cat = new Cat("Kitty");
    a_dog = new Dog("Doggie");

    return 0;
}
```

Przesłanianie metody w podklasie

```
#include <iostream>
using namespace std;
class Pet {
protected:
	string Name;
public:
	Pet(string n) { Name = n; MakeSound(); }
	virtual void MakeSound(void) { cout << Name << " the Pet says: Shh! Shh!" << endl; }
};
class Cat : public Pet {
public:
	Cat(string n) : Pet(n) { }
	void MakeSound(void) { cout << Name << " the Cat says: Meow! Meow!" << endl; }
};
class Dog : public Pet {
public:
	Dog(string n) : Pet(n) { }
	void MakeSound(void) { cout << Name << " the Dog says: Woof! Woof!" << endl; }
};
int main(void) {
	Cat *a_cat;
	Dog *a_dog;

	a_cat = new Cat("Kitty");
	a_dog = new Dog("Doggie");

	return 0;
}
```

Program wyświetli:

Kitty the Pet says: Shh! Shh!

Doggie the Pet says: Shh! Shh!

Przesłanianie metody w podklasie

- Wzywamy metodę ***MakeSound*** jako część konstruktora ***Pet***.
- Program wyświetli następujące linie :
  - Kitty the Pet says: Shh! Shh!
  - Doggie the Pet says: Shh! Shh!
- Oznacza to, że powiązanie między pierwotnymi funkcjami i ich polimorficznymi implementacjami jest ustalane podczas tworzenia obiektu podklasy, a nie wcześniej.

Przesłanianie metody w podklasie

```
#include <iostream>
using namespace std;
class Pet {
protected:
    string Name;
public:
    Pet(string n) { Name = n; }
    virtual void MakeSound(void) { cout << Name << " the Pet says: Shh! Shh!" << endl; }
    void WakeUp(void) { MakeSound(); }
};
class Cat : public Pet {
public:
    Cat(string n) : Pet(n) { }
    void MakeSound(void) { cout << Name << " the Cat says: Meow! Meow!" << endl; }
};
class Dog : public Pet {
public:
    Dog(string n) : Pet(n) { }
    void MakeSound(void) { cout << Name << " the Dog says: Woof! Woof!" << endl; }
};
int main(void) {
    Cat *a_cat;
    Dog *a_dog;

    a_cat = new Cat("Kitty");
    a_cat -> WakeUp();
    a_dog = new Dog("Doggie");
    a_dog -> WakeUp();

    return 0;
}
```

Przesłanianie metody w podklasie

```
#include <iostream>
using namespace std;
class Pet {
protected:
    string Name;
public:
    Pet(string n) { Name = n; }
    virtual void MakeSound(void) { cout << Name << " the Pet says: Shh! Shh!" << endl; }
    void WakeUp(void) { MakeSound(); }
};
class Cat : public Pet {
public:
    Cat(string n) : Pet(n) { }
    void MakeSound(void) { cout << Name << " the Cat says: Meow! Meow!" << endl; }
};
class Dog : public Pet {
public:
    Dog(string n) : Pet(n) { }
    void MakeSound(void) { cout << Name << " the Dog says: Woof! Woof!" << endl; }
};
int main(void) {
    Cat *a_cat;
    Dog *a_dog;

    a_cat = new Cat("Kitty");
    a_cat -> WakeUp();
    a_dog = new Dog("Doggie");
    a_dog -> WakeUp();

    return 0;
}
```

Program wyświetli:
Kitty the Cat says: Meow! Meow!
Doggie the Dog says: Woof! Woof!

Przesłanianie metody w podklasie

- Metoda wirtualna może zostać wywołana nie tylko z zewnątrz, ale także z wewnątrz.
- Kod generuje następujące dane wyjściowe:
  - Kitty the Cat says: Meow! Meow!
  - Doggie the Dog says: Woof! Woof!

Outline

1. **Dziedziczenie**
    1. Definiowanie hierarchii klas
    2. Klasy, dziedziczenie i zgodność typów
    3. Polimorfizm i metody wirtualne
    4. **Obiekty jako parametry i dynamiczne rzutowanie**

Przekazywanie obiektu jako parametru funkcji

- **Każdy obiekt może być używany jako parametr funkcji** i odwrotnie, **dowolna funkcja może mieć parametr jako obiekt dowolnej klasy**.
- Możemy przekazać obiekt do funkcji: **przez wskaźnik i przez referencję**.

Przekazywanie obiektu jako parametru funkcji

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected:
	string name;
public:
	void NameMe(string name) { this -> name = name; }
	void MakeSound(void) { cout << name << " says: no comments" << endl; }
};
void PlayWithPetByPointer(string name, Pet *pet) {
	pet -> NameMe(name);
	pet -> MakeSound();
}
void PlayWithPetByReference(string name, Pet &pet) {
	pet.NameMe(name);
	pet.MakeSound();
}
int main(void) {
	Pet *p1  = new Pet;
	Pet p2;
	PlayWithPetByPointer("anonymous", p1);
	PlayWithPetByReference("no_name", p2);
	PlayWithPetByPointer("no_name", &p2);
	PlayWithPetByReference("anonymous", *p1);
	return 0;
}
```

Przekazywanie obiektu jako parametru funkcji

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected:
	string name;
public:
	void NameMe(string name) { this -> name = name; }
	void MakeSound(void) { cout << name << " says: no comments" << endl; }
};
void PlayWithPetByPointer(string name, Pet *pet) {
	pet -> NameMe(name);
	pet -> MakeSound();
}
void PlayWithPetByReference(string name, Pet &pet) {
	pet.NameMe(name);
	pet.MakeSound();
}
int main(void) {
	Pet *p1  = new Pet;
	Pet p2;
	PlayWithPetByPointer("anonymous", p1);
	PlayWithPetByReference("no_name", p2);
	PlayWithPetByPointer("no_name", &p2);
	PlayWithPetByReference("anonymous", *p1);
	return 0;
}
```

Program wyświetli:

anonymous says: no comments
no name says: no comments
no name says: no comments
anonymous says: no comments

Przekazywanie obiektu przez wartosć

```
#include<iostream>
#include <string>
using namespace std;
class Pet {
protected:
	string name;
public:
	void NameMe(string name) { this -> name = name; }
	void MakeSound(void) { cout << name << " says: no comments" << endl; }
};
void NamePetByValue(string name, Pet pet) {
	pet.NameMe(name);
}
void NamePetByPointer(string name, Pet *pet) {
	pet -> NameMe(name);
}
void NamePetByReference(string name, Pet &pet) {
	pet.NameMe(name);
}
int main(void) {
	Pet pet;
	pet.NameMe("no_name");
	NamePetByValue("Alpha", pet);
	pet.MakeSound();
	NamePetByPointer("Beta", &pet);
	pet.MakeSound();
	NamePetByReference("Gamma", pet);
	pet.MakeSound();
	return 0;
}
```

Przekazywanie obiektu przez wartosć

```
#include<iostream>
#include <string>
using namespace std;
class Pet {
protected:
	string name;
public:
	void NameMe(string name) { this -> name = name; }
	void MakeSound(void) { cout << name << " says: no comments" << endl; }
};
void NamePetByValue(string name, Pet pet) {
	pet.NameMe(name);
}
void NamePetByPointer(string name, Pet *pet) {
	pet -> NameMe(name);
}
void NamePetByReference(string name, Pet &pet) {
	pet.NameMe(name);
}
int main(void) {
	Pet pet;
	pet.NameMe("no_name");
	NamePetByValue("Alpha", pet);
	pet.MakeSound();
	NamePetByPointer("Beta", &pet);
	pet.MakeSound();
	NamePetByReference("Gamma", pet);
	pet.MakeSound();
	return 0;
}
```

Program wyświetli:

no_name says: no comments
Beta says: no comments
Gamma says: no comments

Przekazywanie obiektu podklasy

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected: string name;
public:    Pet(string name) { this -> name = name; }
          void MakeSound(void) { cout << name << " is silent :(" << endl; }
};
class Dog : public Pet {
public: Dog(string name) : Pet(name) {}
        void MakeSound(void) { cout << name << " says: Woof!" << endl; }
};
class GermanShepherd : public Dog {
public: GermanShepherd(string name) : Dog(name) {}
        void MakeSound(void) { cout << name << " says: Wuff!" << endl; }
};
class MastinEspanol : public Dog {
public: MastinEspanol(string name) : Dog(name) {}
        void MakeSound(void) { cout << name << " says: Guau!" << endl; }
};
void PlayWithPet(Pet &pet) {
      pet.MakeSound();
}
```

Przekazywanie obiektu podklasy

```
int main(void) {
	Pet pet("creature");
	Dog dog("Dog");
	GermanShepherd gs("Hund");
	MastinEspanol mes("Perro");
	PlayWithPet(pet);
	PlayWithPet(dog);
	PlayWithPet(gs);
	PlayWithPet(mes);
	return 0;
}
```

Przekazywanie obiektu podklasy

- Program wyświetli:
  - creature is silent :(
  - Dog is silent :(
  - Hund is silent :(
  - Perro is silent :(

Przekazywanie obiektu podklasy

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected:string name;
public:   Pet(string name) { this -> name = name; }
        void MakeSound(void) { cout << name << " is silent :(" << endl; }
};
class Dog : public Pet {
public:    Dog(string name) : Pet(name) {}
      void MakeSound(void) { cout << name << " says: Woof!" << endl; }
};
class GermanShepherd : public Dog {
public:    GermanShepherd(string name) : Dog(name) {}
      void MakeSound(void) { cout << name << " says: Wuff!" << endl; }
};
class MastinEspanol : public Dog {
public:    MastinEspanol(string name) : Dog(name) {}
      void MakeSound(void) { cout << name << " says: Guau!" << endl; }
};
void PlayWithPet(Pet *pet) {
      pet -> MakeSound();
}
```

```
int main(void) {
      Pet *pet = new Pet("creature");
      Dog *dog = new Dog("Dog");
      GermanShepherd *gs = new GermanShepherd("Hund");
      MastinEspanol *mes = new MastinEspanol("Perro");
      PlayWithPet(pet);
      PlayWithPet(dog);
      PlayWithPet(gs);
      PlayWithPet(mes);
      return 0;
}
```

Operator dynamic_cast

- Po pierwsze, zmodyfikowaliśmy metodę ***MakeSound*** w klasie najwyższego poziomu - jest teraz **wirtualna**.
- Po drugie, stworzyliśmy drzewo klas. Dodaliśmy dwa dodatkowe poziomy do drzewa.

Operator dynamic_cast

- Jeśli operator ***dynamic_cast*** jest używany w następujący sposób:
  - dynamic_cast<pointer_type>(pointer_to_object)
- **i** konwersja obiektu ***pointer_to_object*** na typ ***pointer_type*** jest możliwa, wtedy wynikiem konwersji jest **nowy wskaźnik, który jest w pełni użyteczny**
- W przeciwnym razie wynik konwersji jest równy NULL.

Operator dynamic_cast

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected: string name;
public:     Pet(string name) : name(name) {}
        virtual void MakeSound(void) { cout << name << " is silent :(" << endl; }
};
class Dog : public Pet {
public:   Dog(string name) : Pet(name) {}
      void MakeSound(void) { cout << name << " says: Woof!" << endl; }
};
class GermanShepherd : public Dog {
public:   GermanShepherd(string name) : Dog(name) {}
      void MakeSound(void) { cout << name << " says: Wuff!" << endl; }
      void Laufen(void) { cout << name << " runs (gs)!" << endl; }
};
class MastinEspanol : public Dog {
public:   MastinEspanol(string name) : Dog(name) {}
      void MakeSound(void) { cout << name << " says: Guau!" << endl; }
      void Ejecutar(void) { cout << name << " runs (mes)!" << endl; }
};
void PlayWithPet(Pet *pet) {
      GermanShepherd *gs;
      MastinEspanol *mes;
      pet -> MakeSound();
      if(gs = dynamic_cast<GermanShepherd *>(pet))
            gs -> Laufen();
      if(mes = dynamic_cast<MastinEspanol *>(pet))
            mes -> Ejecutar();
}
```

Operator dynamic_cast

```
int main(void) {
        Pet *pet = new Pet("creature");
        Dog *dog = new Dog("Dog");
        GermanShepherd *gs = new GermanShepherd("Hund");
        MastinEspanol *mes = new MastinEspanol("Perro");
        PlayWithPet(pet);
        PlayWithPet(dog);
        PlayWithPet(gs);
        PlayWithPet(mes);
        return 0;
}
```

Operator dynamic_cast

- Program wyświetli wynik:
  - creature is silent :(
  - Dog says: Woof!
  - Hund says: Wuff!
  - Hund runs (gs)!
  - Perro says: Guau!
  - Perro runs (mes)!

Operator dynamic_cast

- Funkcja ***PlayWithPet*** nie ma wskaźnika, ale referencję. W konsekwencji zmienione zostały również dwie części programów :
  - główna funkcja wywołuje ***PlayWithPet*** w nieco inny sposób (spójrz)
    forma wykorzystania ***dynamic_cast*** jest tutaj zupełnie inna; operator przyjmuje następującą formę:
    - dynamic_cast<reference_type>(reference_to_object)
- i zwraca nowo przekształcone (przekonwertowane) odniesienie

Operator dynamic_cast

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected: string name;
public:     Pet(string name) : name(name) {}
        virtual void MakeSound(void) { cout << name << " is silent :(" << endl; }
};
class Dog : public Pet {
public:   Dog(string name) : Pet(name) {}
      void MakeSound(void) { cout << name << " says: Woof!" << endl; }
};
class GermanShepherd : public Dog {
public:   GermanShepherd(string name) : Dog(name) {}
      void MakeSound(void) { cout << name << " says: Wuff!" << endl; }
      void Laufen(void) { cout << name << " runs (gs)!" << endl; }
};
class MastinEspanol : public Dog {
public:   MastinEspanol(string name) : Dog(name) {}
      void MakeSound(void) { cout << name << " says: Guau!" << endl; }
      void Ejecutar(void) { cout << name << " runs (mes)!" << endl; }
};
void PlayWithPet(Pet &pet) {
      pet.MakeSound();
      dynamic_cast<GermanShepherd &>(pet).Laufen();
      dynamic_cast<MastinEspanol &>(pet).Ejecutar();
}
int main(void) {
      Pet pet("creature");
      Dog dog("Dog");
      GermanShepherd gs("Hund");
      MastinEspanol mes("Perro");
      PlayWithPet(pet);
      PlayWithPet(dog);
      PlayWithPet(gs);
      PlayWithPet(mes);
      return 0;
}
```

Operator dynamic_cast

- Program, skompilowany i uruchomiony, generuje następujące, rozczarowujące wyniki:
  - creature is silent :(
  - terminate called after throwing an instance of 'std::bad_cast'
  - what(): std::bad_cast
  - This application has requested the Runtime to terminate it in an unusual way.
  - Please contact the application's support team for more information.

Operator dynamic_cast

- Jest tu coś nowego: deklaracja **try-catch**:
  - try {
  - thing_we_want_to_try_although_we_are_not_quite_sure_if_it_is_reasonable;
  - } catch(…) {}

Operator dynamic_cast

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected: string name;
public:     Pet(string name) : name(name) {}
        virtual void MakeSound(void) { cout << name << " is silent :(" << endl; }
};
class Dog : public Pet {
public:   Dog(string name) : Pet(name) {}
      void MakeSound(void) { cout << name << " says: Woof!" << endl; }
};
class GermanShepherd : public Dog {
public:   GermanShepherd(string name) : Dog(name) {}
      void MakeSound(void) { cout << name << " says: Wuff!" << endl; }
      void Laufen(void) { cout << name << " runs (gs)!" << endl; }
};
class MastinEspanol : public Dog {
public:   MastinEspanol(string name) : Dog(name) {}
      void MakeSound(void) { cout << name << " says: Guau!" << endl; }
      void Ejecutar(void) { cout << name << " runs (mes)!" << endl; }
};
```

Operator dynamic_cast

```
void PlayWithPet(Pet &pet) {
        pet.MakeSound();
        try {
                dynamic_cast<GermanShepherd &>(pet).Laufen();
        } catch(...) {}
        try {
                dynamic_cast<MastinEspanol &>(pet).Ejecutar();
        } catch(...) {}
}
int main(void) {
        Pet pet("creature");
        Dog dog("Dog");
        GermanShepherd gs("Hund");
        MastinEspanol mes("Perro");
        PlayWithPet(pet);
        PlayWithPet(dog);
        PlayWithPet(gs);
        PlayWithPet(mes);
        return 0;
}
```

Operator dynamic_cast

- Wynik działania programu:

  - creature is silent :(
  - Dog says: Woof!
  - Hund says: Wuff!
  - Hund runs (gs)!
  - Perro says: Guau!
  - Perro runs (mes)!