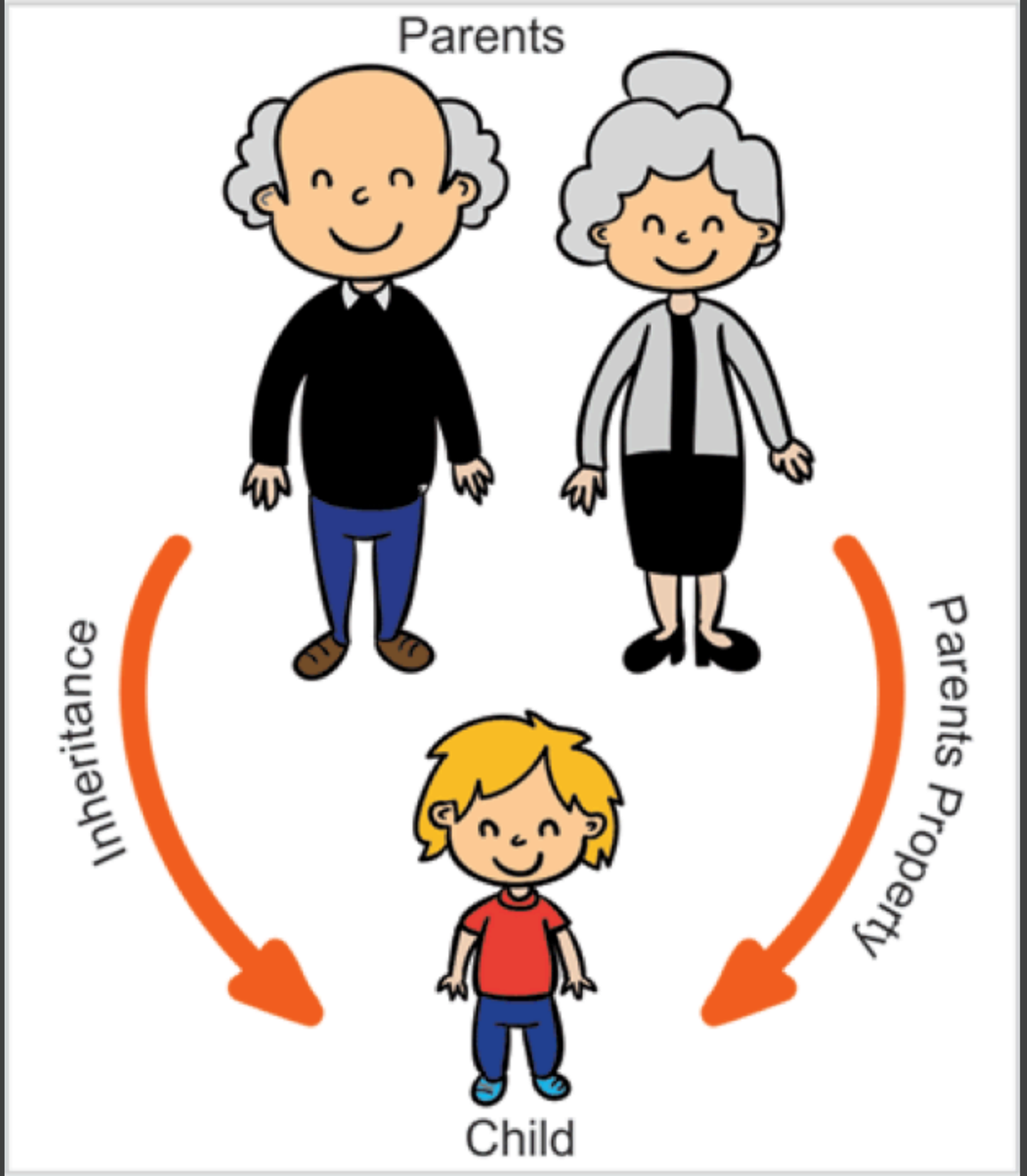
Objektinis Programavimas

Paveldėjimas (*Inheritance*)



Turinys

1. Motyvacija
2. Paveldėjimas
3. Paveldėjimo formos
4. Išvestinės klasės kūrimas
5. UML diagramos
6. Prieigos specifikatoriai

Motyvacija (1)

Kelios kodo eilutės, kelios sugaištos minutės ir 4-a praktinė užduotis "*atlikta*"! 😄

```
#include <iostream>
#include <vector>

template<typename T>
class Vector : public std::vector<T> {
public:
    using std::vector<T>::vector; // naudoti c-tor'ius iš std::vector
};

int main() {
    Vector<int> v(10,1);
    std::cout << "v[9] = " << v[9] << std::endl;
    v.push_back(2);
    std::cout << "v[10] = " << v[10] << std::endl;
    std::cout << "v[11] = " << v[11] << std::endl; // Ką gausime čia?
}
```

Motyvacija (2)

Norint prisitaikyti egzistuojančių klasių funkcionalumą savo poreikiams

```
#include <iostream>
#include <vector>

template<typename T>
class Vector : public std::vector<T> {
public:
    using std::vector<T>::vector;                          // naudoti c-tor'ius iš std::vector
    T& operator[](int i) { return std::vector<T>::at(i); } // patikrina range
    const T& operator[](int i) const { return std::vector<T>::at(i); } // patikrina range const objektams
};

int main() {
    Vector<int> v(10,1);
    std::cout << "v[9] = " << v[9] << std::endl;
    v.push_back(2);
    std::cout << "v[10] = " << v[10] << std::endl;
    std::cout << "v[11] = " << v[11] << std::endl; // O ką dabar gausime?
}
```

Motyvacija (3)

Norint išplėsti egzistuojančių klasių funkcionalumą

```
/* Vector kodas iš ankstesnės skaidrės */
// Vector'ių sudėtis
template<typename T>
Vector<T> operator+(const Vector<T>& a, const Vector<T>& b) {
  if (a.size() != b.size())
    throw std::runtime_error("Vektorių dydžio neatitikimas!");
  auto size = a.size();
  Vector<T> c(size);
  for (size_t i = 0; i != a.size(); ++i)
    c[i] = a[i] + b[i];
  return c;
}

int main() {
    Vector<int> v1(10,1);
    Vector<int> v2(10,2);
    Vector<int> v3 = v1 + v2; // sudėdame Vector'ius
}
```

Paveldėjimas (*inheritance*) (1)

— Objektiškai orientuotame programavime (OOP) **paveldėjimas** (*inheritance*) yra vienas svarbiausių ir naudingiausių principų/mechanizmų.

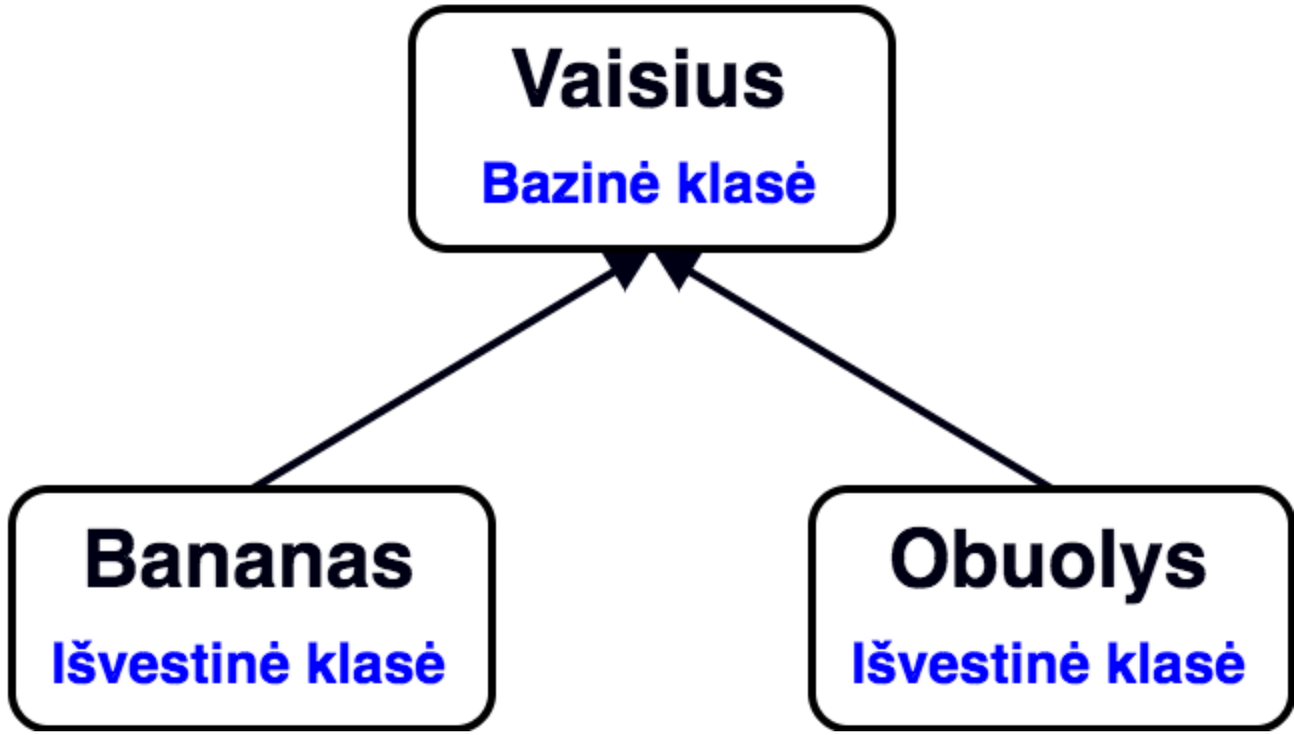
— C++ kalboje klasės gali paveldėti kitos klasės ar net kelių klasių duomenis (*member variables*) ir metodus (*member functions*).

— Tokiu būdu galime kurti naujas klases išplečiant egzistuojančių funkcionalumą.

Paveldėjimas (*inheritance*) (2)

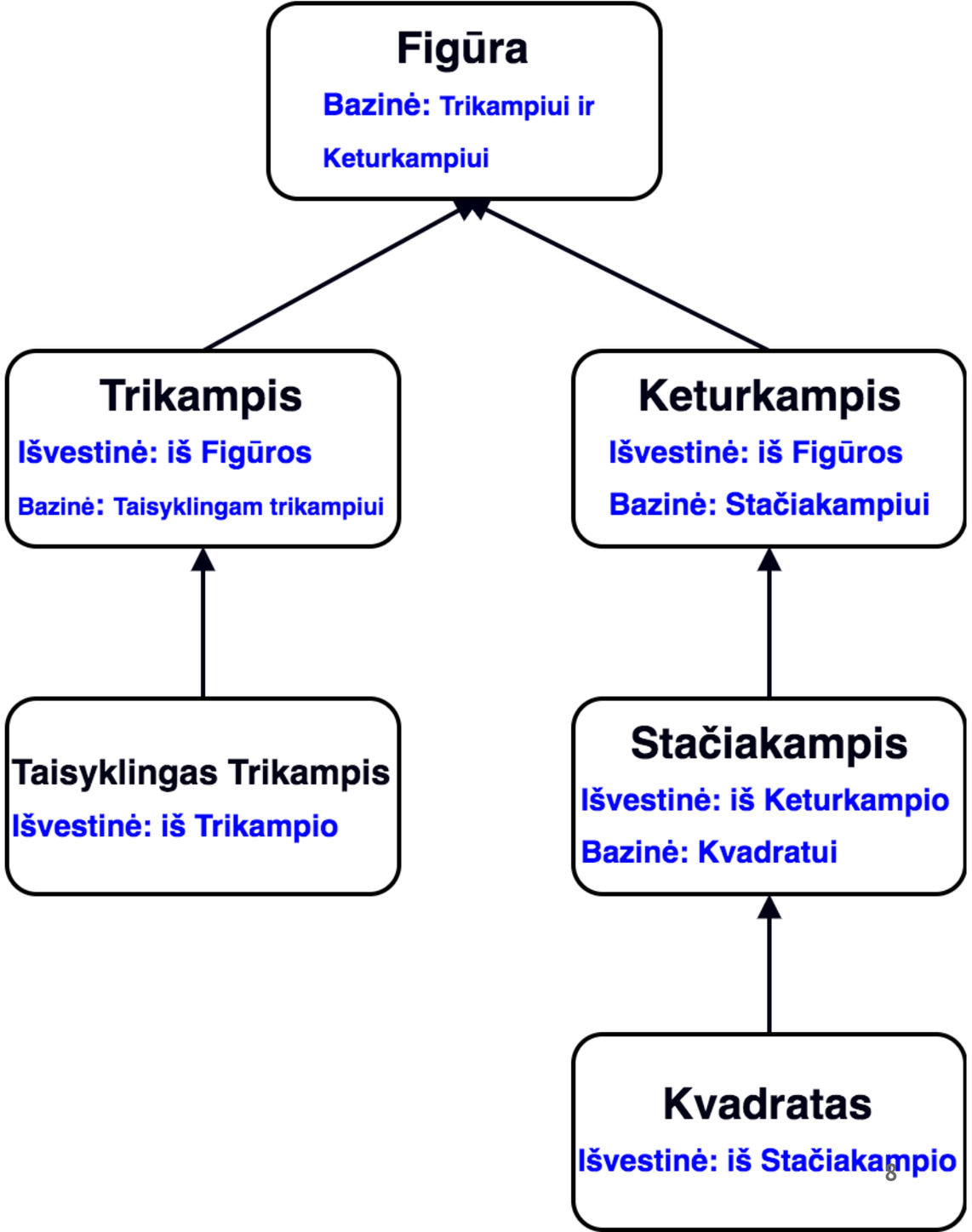
— Klasė, iš kurios paveldimos visos savybes vadinama **bazinė klasė** (*base class*) ar (*parent class*), o ją papildanti - **išvestinė klasė** (*derived class*) ar (*child class*).

— Jei bazinėje klasėje ištaisomos klaidos ar realizuojamos naujos funkcijos, visa tai automatiškai paveldi išvestinės klasės!

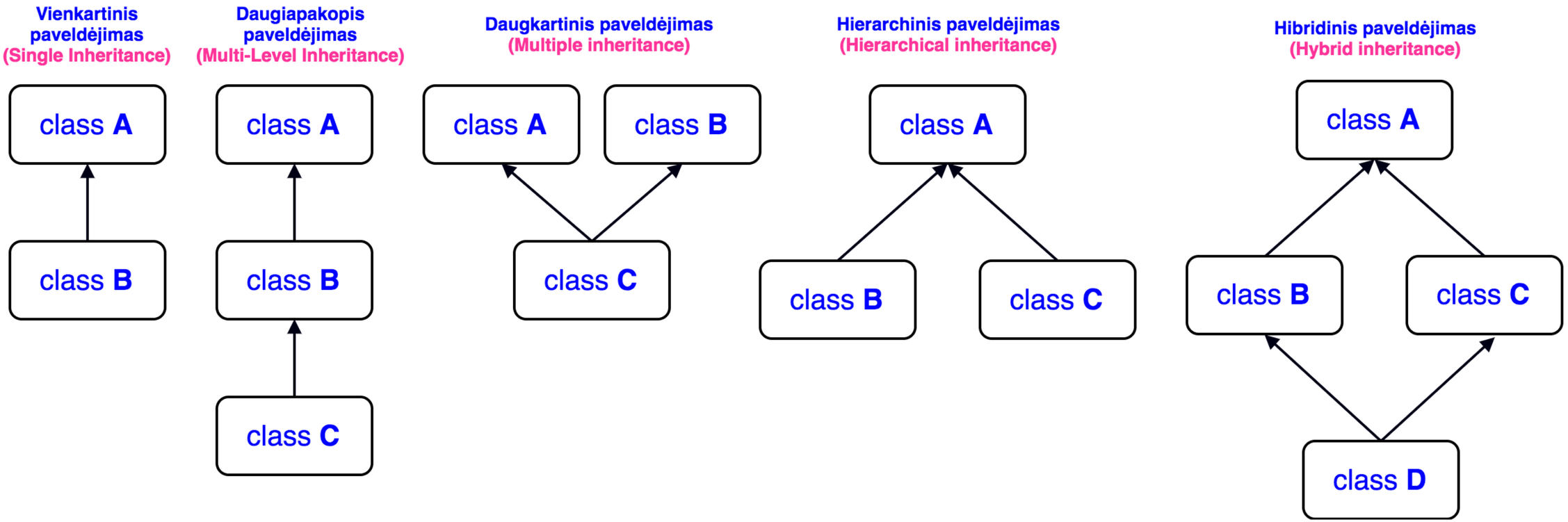


Paveldėjimas (*inheritance*) (3)

— Tam tikros klasės tuo pat metu gali būti išvestinės (iš ankstesnės bazinės klasės) ir bazinės klasės iš jų naujai išvedamoms klasėms.



Paveldėjimo formos



Bazinės (base) klasės kūrimas

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

class Base {
public:
    std::string vardas;
    Base(std::string v = "") : vardas{v} { }
    std::string getVardas() const { return vardas; }
};

int main() {
    Base b{"Remigijus"}; // C++ išskiria atmintį ir iškviečia konstruktorių ją įnicializuoti.
    std::cout << b.getVardas() << std::endl;
    return 0;
}
```

— `Base` yra tradicinė **bazinė** klasė.

Išvestinės (derived) klasės kūrimas (1)

```
class DerivedClass : accessSpecifier BaseClass { ... };
```

— Prieigos specifikatorius (`accessSpecifier`) gali būti: **public**, **protected** ir **private** (numatytasis).

— Specifikatorius nustato išvestinės klasės paveldimumo lygį (*apie tai netrukus*).

— Bazinė klasė neturi priėjimo prie išvestinės klasės duomenų ir funkcijų!

Išvestinės (derived) klasės kūrimas (2)

```
#include<iostream>
#include<string>
#include "Base.h" // Base klasės realizacija

class Derived : public Base {
public:
    int amzius;
    Derived(int a = 0) : amzius{a} {  }
    int getAmzius() const { return amzius; }
};

int main() {
    Base b{"Remigijus"};
    std::cout << b.getVardas() << std::endl;
    Derived d{36};
    std::cout << d.getAmzius() << std::endl;
    std::cout << d.getVardas() << std::endl; // Ką gausime čia?
    return 0;
}
```

Išvestinės (derived) klasės kūrimas (3)

- Išvestinė klasės objektų konstravimas vyksta dvejomis pakopomis: pirmiausia sukonstruojama **bazinė** dalis (aukščiausiai esanti paveldimumo medyje), o tuomet **išvestinė(s)** dalis(-ys), iki žemiausiai paveldimumo medyje esančio palikuonio.
- Taigi, kai mes sukuriame išvestinį objektą, pirmiausia iškviečiamas bazinis numatytas konstruktorius ir tik po to išvestinis konstruktorius.

Išvestinės (derived) klasės kūrimas (4)

— Papildžius **Base** ir **Derived** konstruktorius:

```
// Konstruktoriai
Base(std::string v = "") : vardas{v} { std::cout << "Base c-tor\n"; }
Derived(int a = 0) : amzius{a} { std::cout << "Derived c-tor\n"; }

int main() {
    Derived d{36};
    std::cout << d.getAmzius() << std::endl;
    return 0;
}

/* Gauname:
 Base c-tor
 Derived c-tor
 36
```

Išvestinės (derived) klasės kūrimas (5)

```
// Includinam viską ko reikia

int main() {
    Base b{"Remigijus"};
    std::cout << b.getVardas() << std::endl;
    Derived d{36};
    std::cout << d.getAmzius() << std::endl;
    std::cout << d.getVardas() << std::endl; // Ką čia gausime?
    return 0;
}
```

— Kaip inicializuoti **d.vardas** reikšmę?

— Ok, ignoruojame, kad `vardas` yra **public** 😄 Tuoj jis bus `private`, kaip ir turėtų būti 😉

Išvestinės (derived) klasės kūrimas (6)

```
// Pirmas bandymas: per member-initializer list'a
Derived(int a = 0, std::string v = "") : amzius(a), vardas{v} { }

int main() {
    Derived d{36, "Remis"}; // Ar viskas gerai?
    return 0;
}
```

— Tik nepaveldėti kintamieji gali būti (member) inicializuoti išvestinio konstruktoriaus! Kodėl?

— Kas jei **Base** klasės kintamieji yra `const` ar & tipo?

Išvestinės (derived) klasės kūrimas (7)

```
// Antras bandymas: konstruktoriaus viduje
Derived(int a = 0, std::string v = "") : amzius(a) { vardas = v; }

int main() {
    Derived d{36, "Remis"}; // Ar viskas gerai?
    return 0;
}
```

— Paveldėtiems kintamiesiems galima pakeisti jų reikšmes konstruktoriaus viduje!

Išvestinės (derived) klasės kūrimas (8)

- Tačiau tokiu būdu `vardas` reikšmė yra priskiriama du kartus: 1-ą kartą per **Base** initializer-list'ą, 2-ą kartą **Derived** konstruktoriaus viduje.
- Be to, šios reikšmės nemato **Base** konstruktorius.
- Galiausiai kaip ir prieš tai atveju, kas jei **Base** klasės kintamieji yra `const` ar & tipo?
- Kas būtų, jei `Base::std::string vardas` būtų (kaip ir turėtų būti) `private`?

Išvestinės (derived) klasės kūrimas (9)

```
// Trečias bandymas: pasirinkti reikiamą Base c-tor'ių
Derived(int a = 0, std::string v = "") : Base{v}, amzius(a) { }

int main() {
    Derived d{36, "Remis"}; // Ar viskas gerai?
    return 0;
}
```

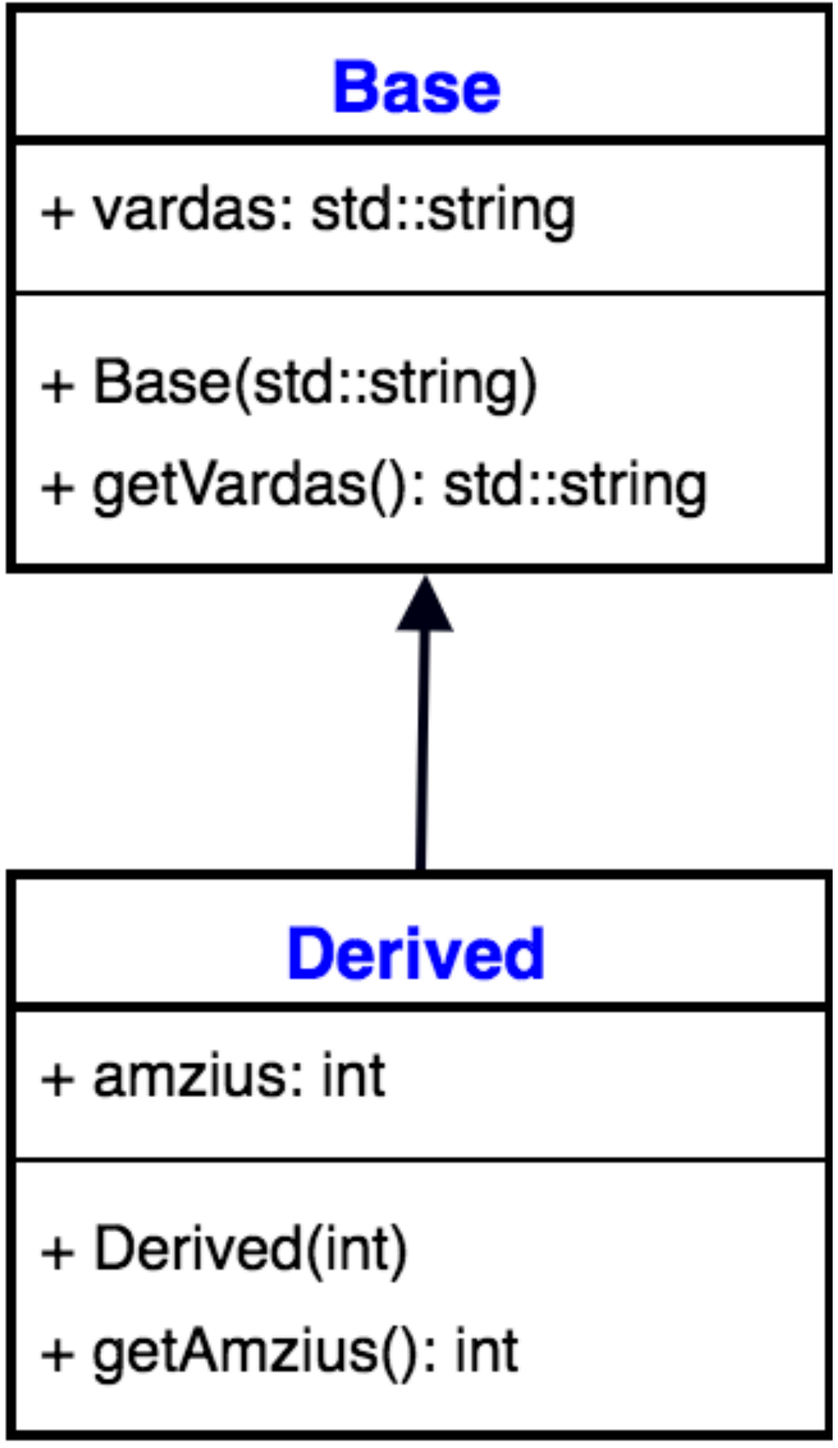
— Kaip dabar su `const` ir & tipo kintamaisiais?

— Kaip dėl `std::string vardas` ir `int amzius` tapimo `private` tipo kintamaisiais?

UML diagramos

Klasių struktūrą ir paveldimumo schemas patogu pateikti UML (*Unified Modeling Language*) diagrama.

- Nurodomi klasių pavadinimai, duomenys, konstruktoriai, destruktoriai ir metodai.
- Paveldimumas žymimas rodykle link bazinės klasės.
- Išvestinės klasės pateikiama tik pavadinimas ir papildomi laukai.



Prieigos specifikatoriai (1)

— Iki šiol pavyzdžiuose naudojome **public** paveldėjimą.

— Mes jau esame susipažinę su **private** ir **public**:

```
class Base {
private:
    std::string vardas; // pasiekia Base nariai, friend'ai, bet ne išvestinės klasės
public:
    std::string pavarde; // pasiekia be kas
};
```

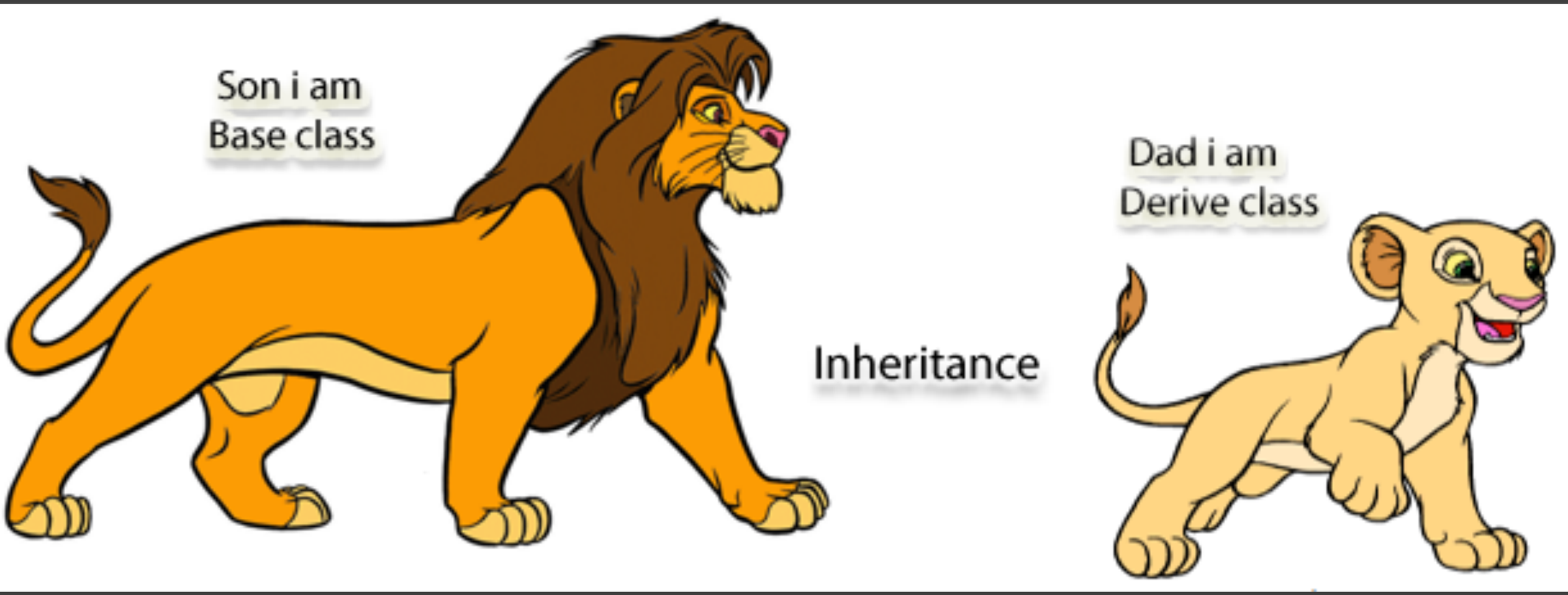
— Paveldėjime naudojama ir **protected**, kurios narius pasiekia klasės nariai, draugai ir išvestinės klasės.

Prieigos specifikatoriai (2)

```
class Base {
private:
    std::string vardas; // pasiekia Base nariai, friend'ai, bet ne išvestinės klasės
public:
    std::string pavarde; // pasiekia be kas
protected:
    std::string pravarde; // pasiekia Base nariai, friend'ai ir išvestinės klasės
};

class Derived: public Base {
public:
    Derived() {
        vardas   = "Remigijus";   // negalima pasiekti private iš išvestinės klasės
        pavarde  = "Paulavičius"; // galima pasiekti public iš išvestinės klasės
        pravarde = "Hmm?";        // galima pasiekti protected iš išvestinės klasės
    }
};

int main() {
    Base b;
    b.vardas   = "Neremigijus";   // Ar galima?
    b.pavarde  = "Nepaulavičius"; // Ar galima?
    b.pravarde = "Nehmm";         // Ar galima?
}
```


Son i am
Base class
Inheritance
Dad i am
Derive class