

Objektinis Programavimas

Polimorfizmas



Turinys

1. Motyvacija
2. Virtualios funkcijos
3. Polimorfizmas
4. Virtualių funkcijų ypatumai
5. Virtualus destruktorius
6. Objektų "apipjaustymas"
 - Konteinerių "apipjaustymas"
 - Nuorodų wrapper'iai
7. Ankstyvas ir vėlyvas susiejimas (bind'ingas)
8. Operatorių persidengimas ir virtualios funkcijos
9. Virtuali lentelė (vtable)
10. Abstraktūs tipai

Motyvacija (1)

Turime Base ir Derived klasses:

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

class Base {
protected:
    std::string vardas;
public:
    Base(std::string v = "") : vardas{v} { }
    void whoAmI() { std::cout << "Aš esu " << vardas << " iš Base klasės\n"; }
};

class Derived : public Base {
protected:
    int amzius;
public:
    Derived(std::string v = "", int a = 0) : Base{v}, amzius{a} { }
    void whoAmI() { std::cout << "Aš esu " << vardas << " iš Derived klasės\n"; }
};
```

Motyvacija (2)

```
int main() {
    Base b{"Remigijus"};
    b.whoAmI();           // ką gausime čia?

    Derived d{"Remis", 36};
    d.whoAmI();           // ką gausime čia?

    Derived &refD = d;   // nuoroda (reference) į d objektą
    refD.whoAmI();        // ką gausime čia?

    Derived *ptrD = &d; // rodyklė (pointer) į d objektą
    ptrD->whoAmI();     // ką gausime čia?

    // Derived paveldi Base dalį, todėl galima nuoroda/rodyklė į Derived:
    Base &refB = d;      // Base tipo nuoroda į Derived objektą d
    Base *ptrB = &d;      // Base tipo rodyklė į Derived objektą d

    refB.whoAmI();        // ką gausime čia?
    ptrB->whoAmI();      // ką gausime čia?
}
```

Motyvacija (3)

- Kadangi refB ir ptrB yra **Base** tipo nuoroda ir rodyklė atitinkamai, todėl "mato" tik **Base** klasės narius, net jei jie yra į **Derived** (iš **Base**) objekta.
- Nors ir Derived::whoAmI() perrašo (paslepia) Base::whoAmI() **Derived** objektams, tačiau **Base** nuoroda/rodyklė nemato Derived::whoAmI().
- To pasekoje jie ir "sako", kad yra iš Base klasės.
- **Bet tai ką čia mes iš viso bandome padaryti?**

Virtualios funkcijos (1)

- **Virtualioji funkcija** yra speciai funkcija, kuri kai iškviečiama įvykdo labiausiai išvestinę (**derived**) sutampačią (**matching**) funkciją, egzistuojančią tarp bazine ir išvestinių klasių:
- Išvestinė funkcija laikoma sutampačia, jei jos deklaracija yra analogiška bazine funkcijos deklaracijai (pavadinimas, parametru tipai ir jų skaičius, const, ir **return** tipas).
- Kad funkcija taptų virtualia, užtenka prieš funkcijos deklaraciją (bazineje klasėje) pridėti **virtual** raktinį žodį ir viskas!

Virtualios funkcijos (2)

- Kai funkcija yra virtuali, tai ir visos jos išvestinės realizacijos yra taip pat virtualios funkcijos, tačiau **virtual** žodis jose yra ne būtinas (nors manoma, kad tai yra gera praktika).
- Kodėl **virtuali** (neegzistuojanti)? Todėl, kad kviečiant vienos klasės funkciją, iš tiesų yra iškviečiama išvestinėje klasėje esanti sutampanti (**matching**) funkcija.
- Ši ypatybė yra vadina **polimorfizmu** (iš graikų kalbos: **poly** (daug), **morphos** (forma)).

Polimorfizmas

Polimorfizmas objektiniame programavime naudojama savoka, kai operacija (metodas) gali būti vykdomas skirtingai, priklausomai nuo konkrečios klasės (ar duomenų tipo) realizacijos, metodo kvietėjui nieko nežinant apie tokius skirtumus.^{wiki}

^{wiki} [https://lt.wikipedia.org/wiki/Polimorfizmas_\(programavime\)](https://lt.wikipedia.org/wiki/Polimorfizmas_(programavime))

Pavyzdžio (iš motyvacijos) tēsinys

```
class Base {  
protected:  
    std::string vardas;  
public:  
    Base(std::string v = "") : vardas{v} {}  
    virtual void whoAmI() { // Padarome whoAmI virtualia funkcija  
        std::cout << "Aš esu " << vardas << " iš Base klasės\n";  
    }  
};  
  
int main() {  
    Derived d{"Remis", 36};  
    Base &refB = d;      // Base tipo nuoroda į Derived objektą d  
    Base *ptrB = &d;    // Base tipo rodyklė į Derived objektą d  
    refB.whoAmI();      // O ką dabar gausime čia?  
    ptrB->whoAmI();    // O ką dabar gausime čia?  
}
```

Klasikinis pavyzdys: ką gyvūnai sako? (1)



Klasikinis pavyzdys: ką gyvūnai sako? (2)

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

// Bazinė klasė
class Gyvunas {
protected:
    std::string vardas;
    // C-tor'ius yra protected, tam kad neleisti tiesiogiai kurti
    // Gyvunas tipo objektų, bet išvestinės klasės galės ji naudoti
    Gyvunas(std::string v) : vardas(v) {}
public:
    std::string getVardas() { return vardas; }
    std::string sako() { return "?"; }
};

// Pirma public tipo išvestinė klasė
class Katinas : public Gyvunas {
public:
    Katinas(std::string v) : Gyvunas(v) {}
    std::string sako() { return "Miauuu"; }
};
```

Klasikinis pavyzdys: ką gyvūnai sako? (3)

```
// Antra public tipo išvestinė klasė
class Suo : public Gyvunas {
public:
    Suo(std::string v) : Gyvunas(v) {}
    std::string sako() { return "Au au au"; }
};

// Per nuorodą (reference) perduodu Gyvunas objektą
void gyvunasSako(Gyvunas &gyv) {
    std::cout << gyv.getVardas() << " sako: " << gyv.sako() << '\n';
}

int main() {
    Katinas kate("Cipsas");
    Suo suo("Kebabas");
    gyvunasSako(kate);
    gyvunasSako(suo);
}
```

Klasikinis pavyzdys: ką gyvūnai sako? (4)

- Kai funkcija `sako()` iš Gyvunas klasės apibrėžta:

```
std::string sako() { return "?"; }
```

Output'a gauname:

Cipsas sako: ?

Kebabas sako: ?

- Bet kai funkcija `sako` iš Gyvunas papildome `virtual`:

```
virtual std::string sako() { return "?"; }
```

Output'a gauname:

Cipsas sako: Miauuu

Kebabas sako: Au au au

Virtualių funkcijų ypatumai (1)

- Virtualios funkcijos turi būti pilnai sutampančios (**matching**):

```
class Base {  
public:  
    virtual int getRandomNumber() { return 42; }  
};  
  
class Derived: public Base {  
public:  
    int getRandomNumber() const { return 99; } // const  
};  
  
int main() {  
    Derived d;  
    Base &refB = d;  
    std::cout << refB.getRandomNumber(); // Ką čia gausime?  
}
```

Virtualių funkcijų ypatumai (2)

Nuo C++11: raktinis žodis `override`

```
class Base {
public:
    virtual int getRandomNumber() { return 42; }
};

class Derived: public Base {
public:
    int getRandomNumber() const override { return 99; } // override
};

int main() {
    Derived d;
    Base &refB = d;
    std::cout << refB.getRandomNumber(); // Ką dabar gausime?
}

/* error: 'int Derived::getRandomNumber() const' marked 'override', but does not override */
```

Virtualių funkcijų ypatumai (3)

Nuo C++11: raktinis žodis final

- Pasiekti priešingam rezultatai t.y. neleisti funkcijos **override**, nuo C++11 atsirado **final**:

```
class Base {
public:
    virtual int getRandomNumber() final { return 42; } // final funkcija
};

class Derived: public Base {
public:
    int getRandomNumber() override { return 99; }
};

int main() {
    Derived d;
    Base &refB = d;
    std::cout << refB.getRandomNumber(); // Ką gausime?
}

/* error: virtual function 'virtual int Derived::getRandomNumber()' overriding final function */
```

Virtualių funkcijų ypatumai (4)

- Raktinis žodis **final** naudojamas ne tik funkcijų, bet ir klasių lygmenyje. Norint uždrausti klasės paveldimumą, galime ją padaryti final'ine:

```
class Base final {} // final klasė
class Derived: public Base {};
```

```
int main() {
    Derived d;
}
```

```
/* error: cannot derive from 'final' base 'Base' in derived type 'Derived' */
```

Virtualių funkcijų ypatumai (5)

- Jei funkcija turi būti virtual'i, tačiau norime pasiekti bazinę funkciją:

```
class Base {  
public:  
    virtual int getRandomNumber() { return 42; }  
};  
  
class Derived: public Base {  
public:  
    int getRandomNumber() { return 99; }  
};  
  
int main() {  
    Derived d;  
    Base &refB = d;  
    // Kviečia Derived::getRandomNumber()  
    std::cout << refB.getRandomNumber() << "\n";  
    // Kviečia Base::getRandomNumber() vietoj virtualios  
    std::cout << refB.Base::getRandomNumber();  
}
```

Virtualių funkcijų ypatumai (6)

- Nenaudoti virtualių funkcijų konstruktoriuose ir destruktoriuose!**
Kodėl?
 - Kai sukuriama išvestinė klasė, pirmiausia sukuriama jos bazine dalis, todėl jei bazine klasės konstruktoriuje kreiptumėmės į virtualią funkciją, kreiptumėmės į neegzistuojančią funkciją, nes išvestinės klasės dalis dar nebuvo sukurta. Todėl vietoje išvestinėje klasėje aprašytos funkcijos, iš tiesų kreiptūsi į bazineje klasėje esančią.
 - Jeigu virtuali funkcija yra iškviečiama bazine klasės destruktoriuje, tuomet ir vėl kreipsimės į bazineje klasėje esančią funkciją, kadangi išvestinės klasės dalis buvo jau sunaikinta.

Virtualių funkcijų ypatumai (7)

- Toks vaizdas, kad yra naudinga visuomet daryti **funkcijas** virtualiomis , tačiau ar iš tiesų visada yra taip?
- Reikia jvertinti, kad iškvesti virtualią funkciją yra mažiau efektyvu (užtrunka ilgiau) negu tradicinę funkciją.
- Taip pat objektai klasių su virtualiomis funkcijomis užima daugiau vienos atmintyje (apie tai truputį vėliau).
Įsitikinkite!
- Todėl efektyviausia daryti funkcijas virtualiomis tik tuomet, kada joms tokis funkcionalumas yra būtinas/logiškas.

Virtualus destruktorius (1)

- Naudodamiesi "**rule of 3**" ir "**rule of 5**" žinome, kad pagal nutylėjimą sukurtas numatytas destruktorius ne visada yra tai ko mums reikia.
- Kai susiduriame su paveldėjimu, jeigu destruktorius yra reikalingas, tai jis (visuomet?) turi būti virtual'us!

Virtualus destruktorius (2)

```
#include <iostream>
class Base {
public:
    ~Base() { std::cout << "D-tor ~Base()" << std::endl; }
};

class Derived : public Base {
private:
    double* elem;
public:
    Derived (int sz = 0) : elem{new double[sz]} { }
    ~Derived() {
        std::cout << "D-tor ~Derived()" << std::endl;
        delete[] elem; // atlaisviname resursus
    }
};

int main() {
    Derived *d = new Derived(10);
    Base *b = d;
    delete b; // kas nutiks čia?
}
```

Virtualus destruktorius (3)

- Iš tiesų darydami **delete b** norėtume iškvesti **Derived** klasės konstruktorių (kuris vėliau iškvies ir **Base** konstruktorių), nes priešingu atveju heap'e išskirtas **new double[sz]** liktis neatlaivintas.
- Tačiau mes pasiekiame padarydami **Base** konstruktorių **virtual'ų**.
- O kas nutiktų, jeigu **virtual'ų** padarytume **Derived**, bet ne **Base** konstruktorių?

Virtualus destruktorius (4)

```
class Base {  
public:  
    // virtualus destruktorius!  
    virtual ~Base() { std::cout << "D-tor ~Base()" << std::endl; }  
};  
  
// Derived klasė kaip anksčiau  
  
int main() {  
    Derived *d = new Derived(10);  
    Base *b = d;  
    delete b; // kas nutiks čia?  
}  
  
/* Dabar gauname:  
   D-tor ~Derived()  
   D-tor ~Base()  
*/
```

Virtualus destruktorius (5)

- Toks vaizdas, kad yra būtina (norint išvengti **memory leak'ų**) visuomet daryti destruktorius virtual'iais. Ar pritariate tam?
- Tačiau reikia atsiminti, kad destruktoriai yra funkcijos, todėl virtualūs veikia lėčiau, o taip pat ir užima daugiau vietos.

Virtualus destruktorius (6)

A base class destructor should be either public and virtual, or protected and nonvirtual - Herb Sutter

- Tuomet objektas iš **Derived** klasės į kurį **point'ina** **Base** klasės (su protected destruktoriumi) rodyklę negali būti ištrintas per **Base** **pointer'j**. Šiuo atveju tai yra būtent tai, ko mums reikia:

```
class Base {  
protected:  
    ~Base() { std::cout << "D-tor ~Base()" << std::endl; } // ne virtual  
};  
  
// Derived klasė kaip anksčiau  
  
int main() {  
    Derived *d = new Derived(10);  
    Base *b = d;  
    delete b; // kas nutiks čia?  
}  
  
/* error: 'Base::~Base()' is protected within this context */
```

Virtualus destruktorius (7)

- Tačiau šiuo atveju tai yra ne tai ko norime, nes jeigu **Base** klasėje būtų išskiriama dinaminė atmintis, mes jos (**run-time**) atlaisvinti negalėtume:

```
class Base {  
protected:  
    ~Base() { std::cout << "D-tor ~Base()" << std::endl; }  
};  
  
int main() {  
    Base *b = new Base();  
    delete b; // kas nutiks čia?  
}  
  
/* error: 'Base::~Base()' is protected within this context */
```

Virtualus destruktorius (8)

- Šiuo atveju Base gali būti **delete'd** tik per išvestinius objektus:

```
class Base {  
protected:  
    ~Base() { std::cout << "D-tor ~Base()" << std::endl; }  
};  
  
// Derived klasė kaip anksčiau  
  
int main() {  
    Derived *d = new Derived();  
    delete d; // kas nutiks čia?  
}  
  
/* D-tor ~Derived()  
D-tor ~Base() */
```

Virtualus destruktorius (9)

- Vadinasi, tokie **Base** klasės objektai negali būti sukurti **stack'e**!
- Dar daugiau, su tokiomis klasėmis negalima naudoti išmaniuju rodyklių (angl. **smart pointers**).
- Todėl C++11 kontekste galima būtų vadovautis tokiomis rekomendacijomis:
 1. Jeigu iš klasės bus kuriamos išvestinės klasės, tuomet jos destruktorių darykite **virtual'ų**.
 2. Jeigu neplanujate, tuomet padarykitę klasę **final**. Tai veiks, kaip **protected** konstruktoriaus atveju, bet išvengsime "nepageidaujamų šalutinių efektų".

Objektų "apipjaustymas" (angl. slicing) (1)

Prisiminkime temos pradžioje matytą pavyzdį:

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

class Base {
protected:
    std::string vardas;
public:
    Base(std::string v = "") : vardas{v} { }
    // virtual'i funkcija
    virtual void whoAmI() { std::cout << "Aš esu " << vardas << " iš Base klasės\n"; }
};

class Derived : public Base {
protected:
    int amzius;
public:
    Derived(std::string v = "", int a = 0) : Base{v}, amzius{a} { }
    void whoAmI() { std::cout << "Aš esu " << vardas << " iš Derived klasės\n"; }
};
```

Objektų "apipjaustymas" (angl. slicing) (2)

```
int main() {
    Derived d{"Remis", 36};
    d.whoAmI();           // ką gavome čia?

    Base &refB = d;      // Base tipo nuoroda į Derived objekta d
    Base *ptrB = &d;     // Base tipo rodyklė į Derived objekta d

    refB.whoAmI();        // ką gavome čia?
    ptrB->whoAmI();      // ką gavome čia?

    Base b = d;          // Base objektui priskiriame Derived objekta
    b.whoAmI();           // ką gausime čia?

    //Derived &d2 = b;    // O ką gautume tokiu atveju?
    //Derived d3 = b;     // Arba tokiu atveju?
}
```

Objektų "apipjaustymas" (angl. slicing) (3)

- Gavome:

Aš esu Remis iš Derived klasės

Aš esu Remis iš Derived klasės

Aš esu Remis iš Derived klasės

Aš esu Remis iš Base klasės

- Tokiu būdu, priskiriant **Base** klasės objektui objekta iš **Derived** klasės tiesiogiai (t.y. ne per nuorodą ar rodyklę) įvyksta **d** objekto "apipjaustymas", nes perkopijuojama tik **Base** dalis iš **Derived** objekto.
- Kadangi **b** neturi **Derived** dalių, todėl **b.whoAmI()** patampa **b.Base::whoAmI()**.

Objektų "apiipjaustymas" (angl. slicing) (4)

- Nors ir logika sako, kodėl mes kada nors galime norėti **Base** objektui priskirti **Derived** objekta, praktika sako visiškai ką kitką.

```
void kasAsEsu(const Base& b) {  
    b.whoAmI();  
}  
int main() {  
    Derived d{"Remis"};  
    kasAsEsu(d); // Ką čia gausime?  
}
```

Objektų "apipjauystymas" (angl. slicing) (5)

- Ups, **get'er** funkcijas visuomet tikslinga daryti **const**:

```
virtual void whoAmI() const {...}
```

- Ką gausime pridėję **const** Base ir Derived klasėse prie **whoAmI()**?

```
// O jeigu perduodame objektą tiesiogiai, o ne per rodykľę?  
void kasAsEsu(const Base b) { // vietoj 'Base&'  
    b.whoAmI();  
}
```

- Perduoti objektus per nuorodą & yra rekomenduotina praktika!

Konteinerių "apipjaustymas" (1)

- Panagrinėkime tokį pavyzdį:

```
#include <vector>
int main() {
    std::vector<Base> v;
    v.push_back(Base("Remigijus")); // Įdedame Base objektą į vektorių
    v.push_back(Base("Remis")); // Įdedame kitą Base objektą į vektorių
    v.push_back(Derived("Super-Remis")); // Įdedame Derived objektą į vektorių
    for (const auto& el : v) // Atspausdiname vektoriaus elementus
        el.whoAmI(); // Ką gausime?
}
```

- Kaip ir anksčiau, **Derived** objektas buvo "apipjaustytas".

Konteinerių "apipjaustymas" (2)

- Ar galime išsisukti su vektoriumi iš nuorodų?

```
#include <vector>
int main() {
    std::vector<Base&> v; // Vietoj Base turime Base& - ar gerai?
    v.push_back(Base("Remigijus"));
    v.push_back(Base("Remis"));
    v.push_back(Derived("Super-Remis"));
    for (const auto& el : v)
        el.whoAmI();
}
```

- Nuorodų negalime priskirti vėliau, galime tik inicIALIZUOTI.

Konteinerių "apipjaustymas" (3)

- Bet galime išsisukti su vektoriumi iš rodyklių!

```
#include <vector>
int main() {
    std::vector<Base*> v;
    v.push_back(new Base("Remigijus"));
    v.push_back(new Base("Remis"));
    v.push_back(new Derived("Super-Remis"));
    for (const auto& el : v)
        el->whoAmI(); // '-' vietoje '.'
}
```

- Ar viskas čia gerai?

Konteinerių "apipjaustymas" (4)

- Panagrinėkime, kaip viskas vyksta **main**'e su destruktoriais!

```
class Base {  
    // ... kaip buvo  
    virtual ~Base() { std::cout << "D-tor ~Base()" << std::endl; }  
};  
  
class Derived : public Base {  
    // ... kaip buvo  
    virtual ~Derived() { std::cout << "D-tor ~Derived()" << std::endl; }  
};  
  
main() {  
    // patyrinėti versiją su pointeriais ir versiją be ju  
}
```

Konteinerių "apipjaustymas" (5)

- Naudojant konteinerį su rodyklėmis, privalome neužmiršti atlaisvinti **dinaminės atminties**:

```
int main() {
    std::vector<Base*> v;
    v.push_back(new Base("Remigijus"));
    v.push_back(new Base("Remis"));
    v.push_back(new Derived("Super-Remis"));
    for (const auto& el : v)
        el->whoAmI();
    // Atlaisviname atminti
    for (const auto& el : v)
        delete el;
}
```

Konteinerių "apipjaustymas" (6)

- O jeigu visgi vietoje rodyklių labai norime naudoti nuorodas &?

std::reference_wrapper is a class template that wraps a reference in a copyable, assignable object. It is frequently used as a mechanism to store references inside standard containers (like std::vector) which cannot normally hold references.^{wrap}

^{wrap} https://en.cppreference.com/w/cpp/utility/functional/reference_wrapper

Nuorodų wrapperiai (1)

```
#include <functional> // std::reference_wrapper

int main() {
    std::vector<std::reference_wrapper<Base>> v; // vietoje Base&
    // v.push_back(Base("Remigijus")); // negalima tempinių objektų
    Base b1("Remigijus");
    Base b2("Remis");
    Derived d("Super-Remis");
    v.push_back(b1);
    v.push_back(b2);
    v.push_back(d);
    // panaudojame get() norėdami gauti elementus iš wrapper
    for (const auto& el : v)
        el.get().whoAmI();
}
```

Nuorodų wrapperiai (2)

Įdomesnis pavyzdys iš ^{wrap}

Demonstrates the use of reference_wrapper as a container of references, which makes it possible to access the same container using multiple indexes

```
#include <algorithm>
#include <list>
#include <vector>
#include <iostream>
#include <numeric>
#include <random>
#include <functional>
int main() { /* Žr. kitą skaidrę */ }
```

^{wrap} https://en.cppreference.com/w/cpp/utility/functional/reference_wrapper

Nuorodų wrapperiai (3)

Įdomesnis pavyzdys iš `wrap`

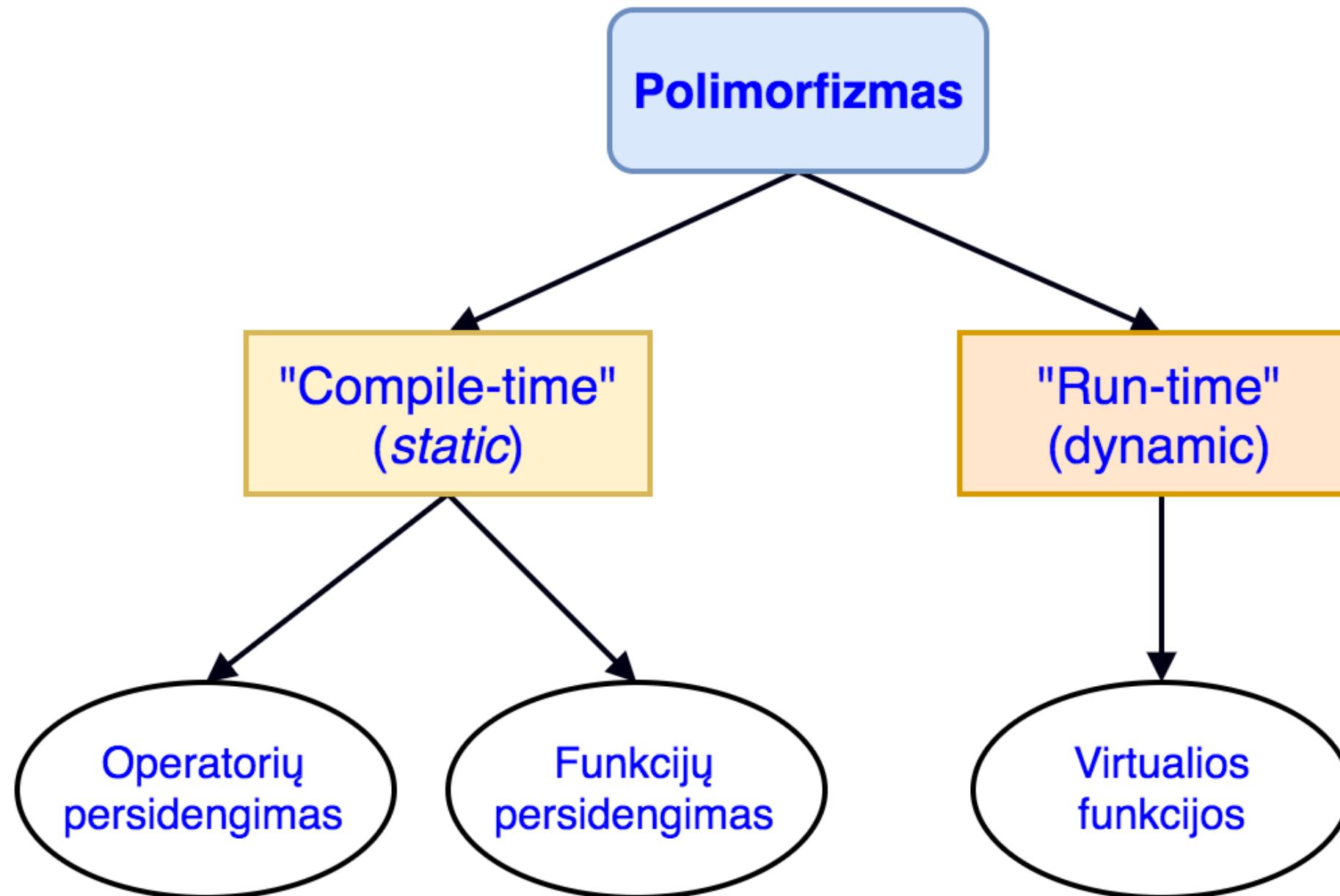
```
// Visi include iš ankstesnės skaidrės
int main() {
    std::list<int> l(10);
    std::iota(l.begin(), l.end(), -4);
    std::vector<std::reference_wrapper<int>> v(l.begin(), l.end());
    // can't use shuffle on a list (requires random access), but can use it on a vector
    std::shuffle(v.begin(), v.end(), std::mt19937{std::random_device{}()});
    std::cout << "Contents of the list: ";
    for (int n : l){ std::cout << n << ' ';}
    std::cout << "\nContents of the list, as seen through a shuffled vector:\n ";
    for (int i : v){ std::cout << i << ' ';}
    std::cout << "\n\nDoubling the values in the initial list...\n\n";
    for (int& i : l) { i *= 2; }
    std::cout << "Contents of the list: ";
    for (int n : l){ std::cout << n << ' ';}
    std::cout << "\nContents of the list, as seen through a shuffled vector:\n ";
    for (int i : v){ std::cout << i << ' ';}
}
```

^{wrap} https://en.cppreference.com/w/cpp/utility/functional/reference_wrapper

Ankstyvas ir vėlyvas susiejimas (bind'ingas) (1)

- **Binding'as** – procesas, kurio metu kintamųjų/funkcijų vardai konvertuojami į mašininius adresus.
- **Ankstyvas (statinis) binding'as** kai kompiliatorius/linkeris gali tiesiogiai funkcijos/kintamojo varda tapatinti su mašininiu adresu.
- **Vėlyvas (dinaminis) binding'as** kai kompiliatorius/linkeris negali tiesiogiai funkcijos/kintamojo vardo tapatinti su mašininiu adresu (**virtualios funkcijos**).

Ankstyvas ir vėlyvas susiejimas (bind'ingas) (2)



Operatorių persidengimas ir virtualios funkcijos (1)

- Dar kartą prisiminkime šį pavyzdį:

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

class Base {
protected:
    std::string vardas;
public:
    Base(std::string v = "") : vardas{v} { }
    virtual void whoAmI() const { std::cout << "Aš esu " << vardas << " iš Base klasės\n"; }
};

class Derived : public Base {
protected:
    int amzius;
public:
    Derived(std::string v = "", int a = 0) : Base{v}, amzius{a} { }
    void whoAmI() const { std::cout << "Aš esu " << vardas << " iš Derived klasės\n"; }
};
```

Operatorių persidengimas ir virtualios funkcijos (2)

- Iki šiol informaciją atspausdindavome panaudodami tam skirtą **whoAmI()** funkciją:
- O kaip dėl **operator<<** panaudojimo?

```
int main() {
    Derived d{"Remis", 36};
    Base &b = d;
    d.whoAmI(); // atspausdiname info apie d objektą
    b.whoAmI(); // atspausdiname info apie b objektą
    std::cout << d << std::endl; // Ką gausime čia?
}
```

Operatorių persidengimas ir virtualios funkcijos (3)

Realizuojame operator<< **kaip friend funkciją**

```
class Base {  
    /* ... viskas kaip buvo anksčiau */  
    friend std::ostream& operator<<(std::ostream &out, const Base &b){  
        out << "Aš esu " << b.vardas << " iš Base klasės\n";  
        return out;  
    }  
};  
  
class Derived : public Base {  
    /* ... viskas kaip buvo anksčiau */  
    friend std::ostream& operator<<(std::ostream &out, const Derived &d){  
        out << "Aš esu " << d.vardas << " iš Derived klasės\n";  
        return out;  
    }  
};
```

Operatorių persidengimas ir virtualios funkcijos (4)

- Atspausdiname panaudodami **operator<<**:

```
int main() {  
    Derived d{"Remis", 36};  
    Base &b = d;  
    std::cout << b << std::endl; // Ką gausime čia?  
    std::cout << d << std::endl; // Ką gausime čia?  
}
```

- Viskaip kaip ir veikia, tačiau **operator<<()** yra nevirtuali funkcija, todėl pirmu atveju iškvietė **operator<<()** iš **Base**, o ne iš **Derived**.

Operatorių persidengimas ir virtualios funkcijos (5)

- Tai sprendimas akivaizdus, padarome `operator<<()` kad būtų **virtual** ir woolia!?

```
class Base {  
    /* ... viskas kaip buvo */  
  
    // Padarėme operatorių virtual - ar viskas gerai?  
    virtual friend std::ostream& operator<<(std::ostream &out, const Base &b){  
        out << "Aš esu " << b.vardas << " iš Base klasės\n";  
        return out;  
    }  
};
```

error: 'virtual' is invalid in friend declarations

Operatorių persidengimas ir virtualios funkcijos (6)

Kodėl negalime virtual'inti operator<<?

1. Tik klasės nario funkcijos gali būti **virtual'iomis**, t.y. kurias galima **override'inti**, o **operator<<** yra realizuotas kaip **friend** funkcija (nes negali būti realizuotas kaip nario funkcija!), esanti už klasės ribų. Už klasės ribų funkcijas galime tik **overloadinti**, bet ne **override'inti**.
2. Net jeigu ir galėtume **virtual'inti** operatorių, bėda yra ta, kad input parametrai **Base::operator<<** ir **Derived::operator<<** skiriasi, t.y. nėra **matched**, todėl negalimas būtų **override'as**!

Operatorių persidengimas ir virtualios funkcijos (7)

O tai ką tokiu atveju daryti? (1)

```
class Base {  
    /* ... kaip buvo */  
    virtual void whoAmI() const { std::cout << "Aš esu " << vardas << " iš Base klasės\n"; }  
  
    // Overloadint'a (kita) virtuali whoAmI() funkcija  
    virtual std::ostream& whoAmI(std::ostream& out) const {  
        out << "Aš esu " << vardas << " iš Base klasės\n";  
        return out;  
    }  
  
    // Overloadint'as operator<< kaip friend funkcija, o dešininis operandas yra Base&  
    friend std::ostream& operator<<(std::ostream &out, const Base &b) {  
        // Visą darbą atliks whoAmI() funkcija, kuri yra virtuali!  
        return b.whoAmI(out);  
    }  
};
```

Operatorių persidengimas ir virtualios funkcijos (8)

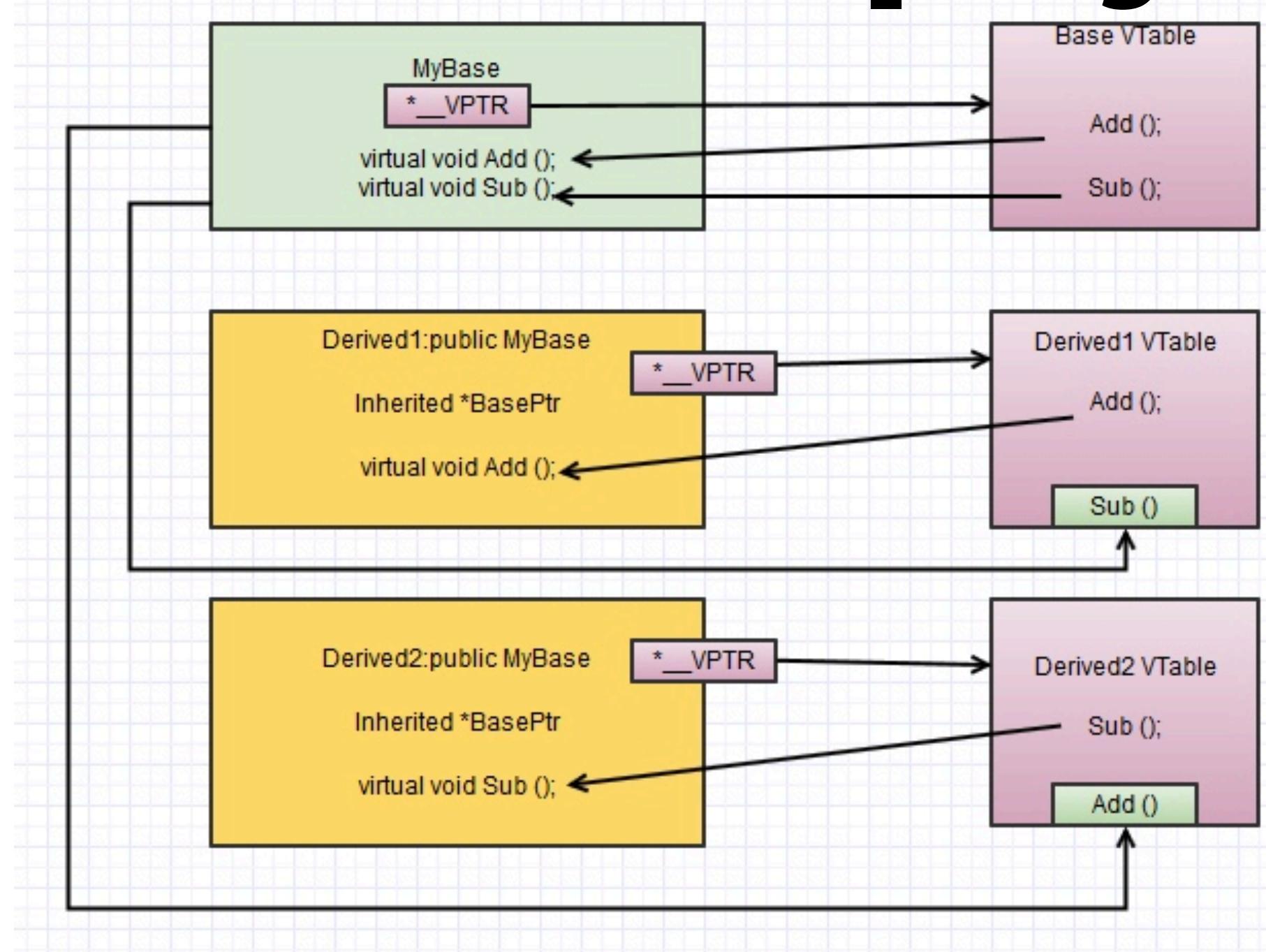
O tai ką tokiu atveju daryti? (2)

```
class Derived {  
    /* ... kaip buvo */  
    virtual void whoAmI() const { std::cout << "Aš esu " << vardas << " iš Derived klasės\n"; }  
    // Overloadint'a (kita) Derived virtualios whoAmI() realizacija  
    virtual std::ostream& whoAmI(std::ostream& out) const {  
        out << "Aš esu " << vardas << " iš Derived klasės\n";  
        return out;  
    }  
};  
  
int main() {  
    Base b("Remis");  
    std::cout << b << std::endl;  
    Derived d("Remigijus");  
    std::cout << d << std::endl; // naudoja Base::operator<<  
    Base &refB = d;  
    std::cout << refB << std::endl;  
}
```

Virtuali lentelė (vtable)

- Realizuodama virtualias funkcijas, C++ naudoja specialią vėlyvo susiejimo (**binding'o**) formą - virtualias lenteles (**virtual table**).
- Virtuali lentelė yra funkcijų paieškos lentelė (**lookup table**) reikalinga atitinkamų funkcijų iškvietimui dinaminiam-vėlyvam susiejimui.
- Virtualios lentelės dar vadinimos - **vtable** arba **virtualių funkcijų lentelėmis**.

Virtualios lentelės pavyzdys



Komentarai apie virtualias lenteles

- Kadangi klasėje yra virtualioji funkcija, C++ kompiliatorius automatiškai sukuria rodyklę *VPTR point'inančią į virtualią lentelę (štai kodėl objektai užima daugiau vienos!)
 - **vtable** yra **array** sudarytas iš funkcijų rodyklių nukreiptų į virtualias funkcijas.
- Rodyklę VPTR paveldi kiekviena išvestinė klasė, tačiau ji nukreipta (point'ina) į atitinkamos klasės virtualią lentelę.
- Kiekvienai atitinkamai klasei virtualią lentelę (**vtable**) kompiliatorius sukuria automatiškai.

Abstraktūs tipai (1)

- Iki šiol visos mūsų kurtos virtual'ios funkcijos buvo apibrėžtos.
- C++ yra galimybė kurti ir **visiškai** (angl. **pure**) **virtualias funkcijas (abstrakčias funkcijas)**, kurios yra visiškai **neapibrėžtos**.
- Tokios klasės turinčios abstrakčias funkcijas vadinamos **abstrakčiomis klasėmis**, t.y, negalima sukurti tų klasiu tipo objektų!
- Išvestinės klasės turi realizuoti šias visiškai virtualias funkcijas arba jos taip pat bus bazine abstrakčios klasės.

Abstraktūs tipai (2)

```
// Abstrakti klasė
class Base {
protected:
    std::string vardas;
public:
    Base(std::string v = "") : vardas{v} { }
    // Abstrakčią visiškai (virtualią) funkciją gauname priskyrę = 0
    virtual void whoAmI() = 0;
    virtual std::string getVardas() { return vardas; }; // virtuali f-ija
};

int main() {
    Base b; // ką čia gausime?
}
```

Abstraktūs tipai (3)

```
// Išvestinėje klasėje realizuojame whoAmI()
class Derived : public Base {
protected:
    int amzius;
public:
    Derived(int a = 0, std::string v = "") : Base{v}, amzius{a} { }
    void whoAmI() { std::cout << "Aš esu " << vardas << " iš Derived klasės\n"; }
};

int main() {
    Derived d(36, "Remis");
    d.whoAmI(); // ką čia gausime?
}
```

- Kas nutiktų jeigu **Derived** klasėje nerealizuotume **whoAmI()**?

Abstraktūs tipai (4)

- Sugržkime prie "ką gyvūnai sako?" realizacijos, kurioje bazine klasės konstruktorių tyčia padarėme **protected**:

```
class Gyvunas { // Bazine klasė
protected:
    std::string vardas;
    // C-tor'ius yra protected, tam kad neleisti tiesiogiai kurti
    // Gyvunas tipo objektų, bet išvestinės klasės galės ji naudoti
    Gyvunas(std::string v) : vardas(v) {}
public:
    std::string getVardas() { return vardas; }
    virtual std::string sako() { return "?"; }
};
```

- Tačiau mes galime sukurti išvestines klasses, kurios "užmiršta" realizuoti **sako()**.

Abstraktūs tipai (5)

```
// Trečia public tipo išvestinė klasė
class Arklys : public Gyvunas {
public:
    Arklys(std::string v) : Gyvunas(v) {}
    // std::string sako() { return "Igaga"; }
};

void gyvunasSako(Gyvunas &gyv) {
    std::cout << gyv.getVardas() << " sako: " << gyv.sako() << '\n';
}

int main() {
    Suo suo("Kebabas");
    Arklys arklys("Beris");
    gyvunasSako(suo);
    gyvunasSako(arklys); // Ką gausime čia?
}
```

Abstraktūs tipai (6)

- Norint to išvengti, teisingiau būtų naudoti visiškai virtualią funkciją:

```
class Gyvunas { // Abstrakčioji klasė; nebūtina c-tor protected
protected:
    std::string vardas;
    Gyvunas(std::string v) : vardas(v) {}
public:
    std::string getVardas() { return vardas; }
    virtual std::string sako() = 0; // Visiškai virtuali funkcija
};

int main() {
    Arklys arklys("Beris");
    gyvunasSako(arklys); // Ką dabar gausime čia?
}
```

Abstraktūs tipai (7)

- Pasirodo net visiškai virtualią funkciją galima realizuoti:

```
class Gyvunas { // Abstrakčioji klasė; nebūtina c-tor protected
    /* viskas kaip anksčiau */
    virtual std::string sako() = 0; // Visiškai virtuali funkcija
};

// Gali būti interpretuojama kaip rekomendacinė realizacija
std::string Gyvunas::sako() { return "Bla bla bla"; }

class Monstras : public Gyvunas {
public:
    Monstras(std::string v) : Gyvunas(v) {}
    // Panaudojame rekomendacine realizaciją
    std::string sako() { return Gyvunas::sako(); }
};
```

Sąsajų (interface) klasės (1)

Abstrakčių tipų panaudojimas - abstraktus konteineris

- Interfeiso klasė - neturi kintamųjų narių ir visos funkcijos yra abstrakčios.
- Apibrėžia reikiama funkcionalaumą, tačiau jo nerealizuojama.

```
// Interfeiso klasė
class Container {
public:
    virtual double& operator[](int) = 0; // abstrakti virtuali funkcija
    virtual int size() const = 0;          // abstrakti virtuali funkcija
    virtual~Container() {}                // virtualus destruktorius
};
```

Sąsajų (interface) klasės (2)

- Šį konteinerį **Container** galėsime panaudoti tokiaame kontekste:

```
void printElem(Container& c) {  
    for (int i=0; i!= c.size(); ++i)  
        std::cout << c[i] << ' ';  
    std::cout << std::endl;  
}
```

Sąsajų (interface) klasės (3)

```
#include<vector>
// VectorContainer realizacija paremta Container
class VectorContainer : public Container {
    std::vector<double> v; // čia gali būti ir mūsų kuriamas Vector
public:
    VectorContainer(int s) : v(s) { } // Vector'ius iš s elementų
    VectorContainer(std::initializer_list<double> lst) : v(lst) { } // Vector'ius iš lst
    ~VectorContainer() {}
    double& operator[](int i) { return v[i]; }
    int size() const { return v.size(); }
};

int main() {
    VectorContainer vc1(10);
    printElem(vc1);
    VectorContainer vc2{1,2,3,4,5,6,7,8,9,10};
    printElem(vc2);
}
```

?



WHAT
DOES THE
FOX
SAY?

wt