Objektinis Programavimas

Polimorfizmas

Turinys

1. Motyvacija
2. Virtualios funkcijos
3. Polimorfizmas
4. Virtualių funkcijų ypatumai
5. Virtualus destruktorius
6. Ankstyvas ir vėlyvas susiejimas (bind'ingas)
7. Virtuali lentelė (vtable)
8. Abstraktūs tipai

Motyvacija (1)

Turime `Base` **ir** `Derived` **klases:**

```
#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(int a = 0, std::string v = "") : 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{36, "Remis"};
    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 rodyklė 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**) objektą.

— 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 speciali funkcija, kuri kai iškviečiama įvykdo labiausiai išvestinę (**derived**) sutampančią (**matching**) funkciją, egzistuojančią tarp bazinės ir išvestinių klasių:

— Išvestinė funkcija laikoma sutampančia, jei jos deklaracija yra analogiška bazinės funkcijos deklaracijai (pavadinimas, parametrų tipai ir skaičius, `const`, ir return tipas).

— Kad funkcija taptų virtualia, užtenka prieš funkcijos deklaraciją (bazinėje 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** žodelis jose yra ne būtinas (nors daug kas mano, kad tai yra gera praktika).

— Kodėl **virtuali** (neegzistuojanti)? Todėl, kad kviečiant vienos klasės funkciją, iš tiesų yra iškviečiama kitoje klasė esanti sutampanti (**matching**) funkcija.

— Ši ypatybė yra vadinama **polimorfizmu** (iš graikų kalbos: **poly** (daug), **morphos** (forma)).

Polimorfizmas

Polimorfizmas objektiniame programavime naudojama sąvoka, 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)

Pavyzdžio (iš motyvacijos) tęsinys

```
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{36, "Remis"};
    Base &refB = d;     // Base tipo rodyklė 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)

```cpp
#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 jį 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 bazinė dalis, todėl jei bazinės 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 bazinėje klasėje esančią.
  - Jeigu virtuali funkcija yra iškviečiama bazinės klasės destruktoriuje, tuomet ir vėl kreipsimės į bazinėje klasėje esančią funkciją, kadangi išvestinės klasės dalis buvo jau sunaikinta.

Virtualių funkcijų ypatumai (7)

- Toks vaizdas, kad yra naudinga visuomet daryti **funkcijas** `virtual`'iomis , tačiau ar iš tiesų visada yra taip?
  - Reikia įvertinti, kad iškviesti virtualią funkciją yra mažiau efektyvu (užtrunka ilgiau) negu tradicinę funkciją.
  - Taip pat objektai klasių su virtualiomis funkcijos užima daugiau vietos atmintyje (apie tai truputį vėliau). Įsitikinkite!
- Todėl efektyviausia daryti funkcijas virtualiomis tik tuomet, kada joms toks funkcionalumas yra būtinas/logiškas.

`Virtual`us destruktorius (1)

— Naudodamiesi "**rule of 3**" ir "**rule of 5**" žinome, kad pagal nutylėjimą sukurtas numatytasis 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?
}
```

`Virtual`**us destruktorius (3)**

— Iš tiesų darydami delete b norėtume iškviesti **Derived** klasės konstruktorių (kuris vėliau iškvies ir **Base** konstruktorių), nes priešingu atveju heap'e išskirtas new double[sz] liks neatlaisvintas.

— Tą mes pasiekiame padarydami **Base** konstruktorių virtual'ų.

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()
 */
```

`Virtual`us 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

— Objektas iš **Derived** klasės į kurį point'ina **Base** klasės (su `protected` destruktoriumi) rodyklė negali būti ištrintas per **Base** pointer'į. 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() */
```

`Virtual`us destruktorius (9)

— Vadinasi **Base** klasės objektai negali būti sukurti **stack**'e!

— Dar daugiau, su tokiomis klasėmis negalima naudoti išmaniųjų 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 neplanuojate, tuomet padarykitę klasę final. Tai veiks, kaip protected konstruktoriaus atveju, bet išvengsime "nepageidaujamų šalutinių efektų".

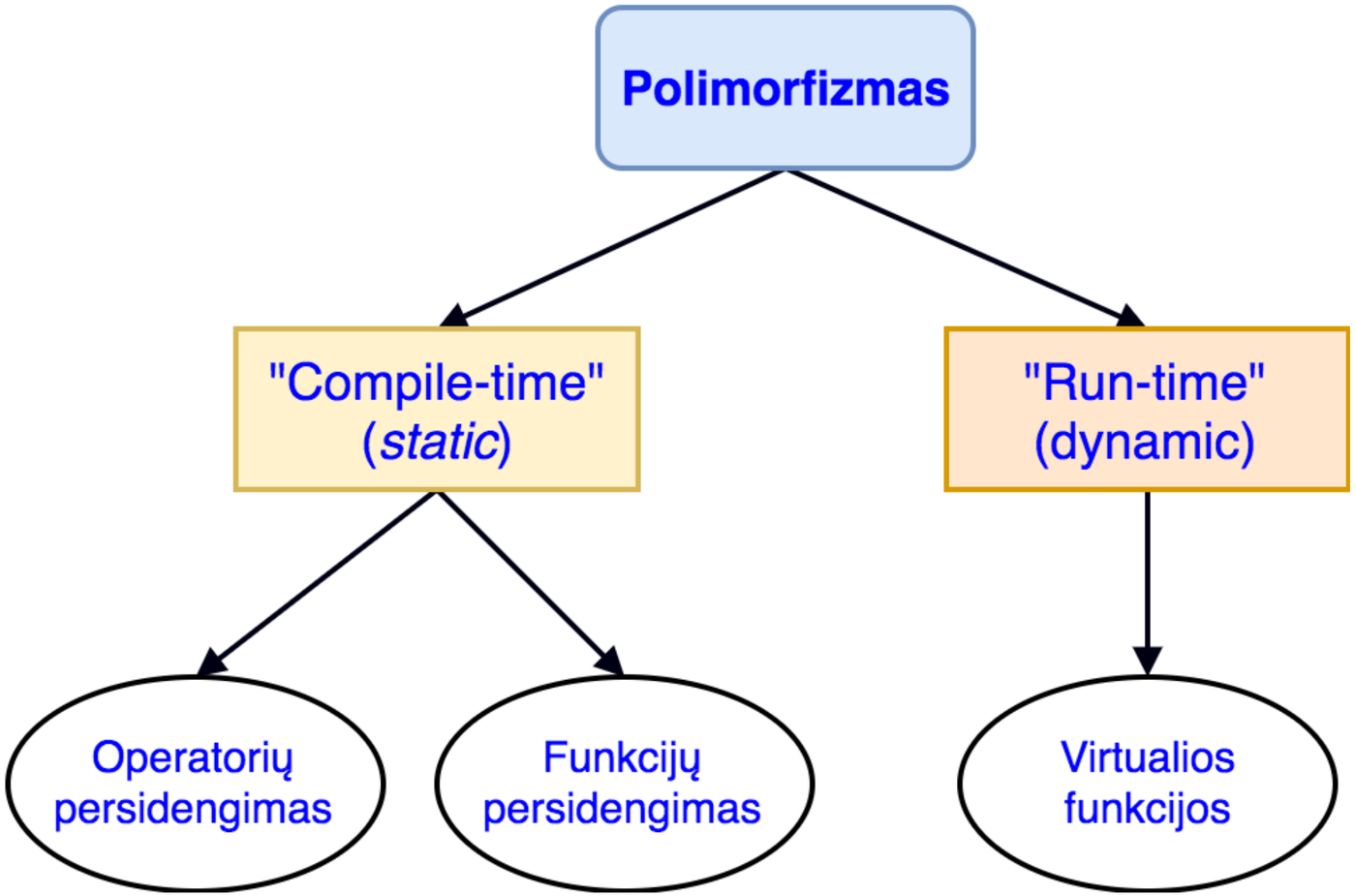
Ankstyvas ir vėlyvas susiejimas (bind**'ingas) (1)**

— **Binding'as** - procesas, kurio metu kintamųjų/funkcijų vardai konvertuojami į mašininius adresus.

— **Ankstyvas (statinis) binding'as** kai kompiliatorius/linkeris gali tiesiogiai funkcijos/kintamojo vardą 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)



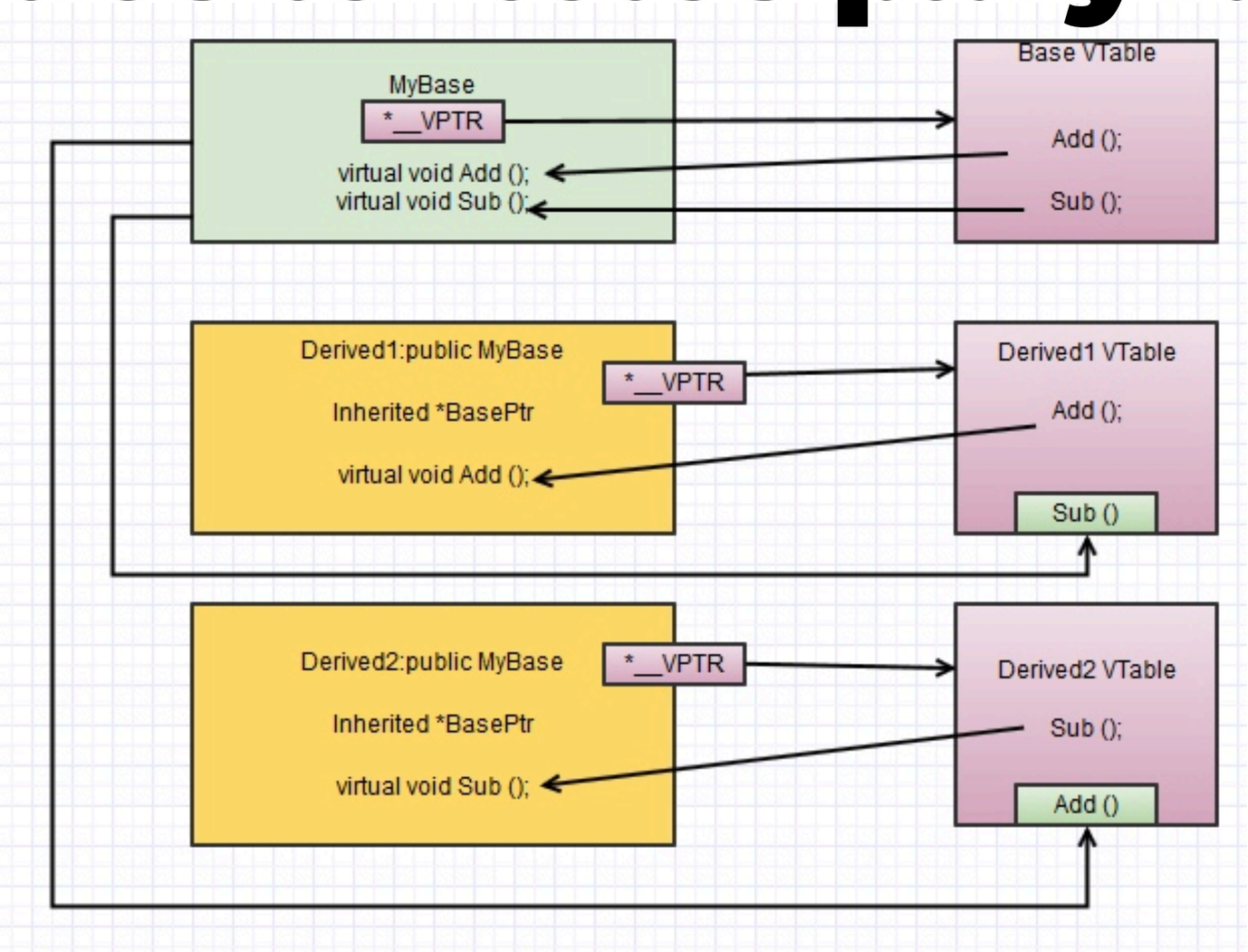
Virtuali lentelė (vtable**)**

— 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 iškvietimų dinaminiam-vėlyvam susiejimui.

— Virtualios lentelės dar vadinimos - vtable arba virtualių funkcijų lentelė.

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 vietos!)

— Rodyklę __VPTR paveldi kiekviena išvestinė klasė, tačiau ji nukreipti į atitinkamos klasės virtualią lentelę.

— Kiekvienai atitinkamai klasei virtualią lentelę (vtable) kompiliatorius sukuria automatiškai.

— vtable yra array sudarytas iš funkcijų rodyklių nukreiptų į virtualias funkcijas.

Abstraktūs tipai (1)

— Iki šiol visos mūsų kurtos `virtual`'ios funkcijos buvo apibrėžtos.

— C++ yra galimybė kurti ir **visiškai virtualias funkcijas (abstrakčias funkcijas) *, kurios yra visiškai *neapibrėžtos**.

— Išvestinės klasės turi realizuoti šias visiškai virtualias funkcijas arba jos taip pat bus bazinės abstrakčios klasės!

— Klasės turinčios abstrakčias funkcijas vadinamos **abstrakčiomis klasėmis**, t.y, negalima sukurti tų klasių tipo objektų!

Abstraktūs tipai (2)

```
// Abstrakti klasė
class Base {
protected:
    std::string vardas;
public:
    Base(std::string v = "") : vardas{v} { }
    virtual void whoAmI() = 0; // Visiškai virtuali funkcija
    virtual std::string getVardas() { return vardas; }; // virtuali f-ija
};

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

Abstraktūs tipai (3)

```
// Abstrakčią (virtualią) funkciją gauname priskyrę ją = 0
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;
    d.whoAmI(); // ką čia gausime?
}
```

Abstraktūs tipai (4)

— Sugrįžkime prie "**ką gyvūnai sako?**" realizacijos, kurioje bazinės klasės konstruktorių tyčia padarėme protected:

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

— Mes galime sukurti išvestines klases, kurios "užmiršta" realizuoti sako() funkciją.

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
 protected:
  std::string vardas;
  Gyvunas(std::string v) : vardas(v) {}
 public:
  std::string getVardas() { return vardas; }
  virtual std::string sako() = 0; // Visiškai virtuali funkcija
};

// Rekomendacinė realizacija
std::string Gyvunas::sako() { return "Bla bla bla"; }

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

Sąsajų (interface) klasės (1)

Abstraktus konteineris

```
// Interfeiso klasė - neturi narių kintamųjų ir visos funkcijos abstrakčios.
// Apibrėžia funkcionalumą, tačiau nerealizuoja jo.
class Container {
public:
  virtual double& operator[](int) = 0; // abstrakti virtuali funkcija
  virtual int size() const = 0;        // abstrakti virtuali funkcija
  virtual~Container() {}               // destruktorius
};

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

Sąsajų (interface) klasės (2)

```
#include<vector>

// VectorContainer realizuoja Container
class VectorContainer : public Container {
  std::vector<double> v; // čia gali būti ir mūsų 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