KLASIŲ HIERARCHIJOS KŪRIMO MECHANIZMAI POLIMORFIZMAS IR VIRTUALIOS FUNKCIJOS

Paveldėjimas. Jau žinome, kad kuriant klasių hierarchiją, naujas klases galima kurti pridedant naujus laukus ir metodus prie jau egzistuojančių klasių. Klasė, kurios pagrindu konstruojama nauja klasė, vadinama

protėvis (bazinė arba *motininė klasė)*,

o naujai sukonstruota klasė –

palikuonis (*išvestinė klasė* arba *vaikas*).

Jei protėvio klasė viena, paveldėjimas vadinamas *paprastas* (*viengubas*), jei tokių klasių daugiau – *sudėtinis* (*daugybinis*, *daugkartinis*).

Tokiu būdu kuriama klasių hierarchija susideda iš vis sudėtingesnių palikuonių klasių. C++ kalboje paveldimi visų protėvių laukai ir metodai. Jų dubliavimui išvengti naudojamas *virtualus pavedėjimas*. Paveldėjimo mechanizmas leidžia kurti naujas programas iš standartinių dažnai pasitaikančių klasių bibliotekų, kas labai palengvina programuotojo darbą.

Polimorfizmas. Kuriant klasių hierarchija kai kurios paveldimos jų savybės turi vienodus vardus, bet jų esmė gali pasikeisti. Tokiu atveju, skirtinguose hierarchijos lygmenyse galima apibrėžti skirtingus veiksmus vienodo pavadinimo metodui. Tai vadinama paprastu polimorfizmu, o atitinkami metodai *statiniais polimorfiniais.* Konkrečios klasės polimorfinio metodo realizacija dažnai vadinama *aspektu.*

Polimorfizmo realizacijos mechanizmai programavimo kalbose.

Grynas polimorfizmas – funkcijos kodas gali būti skirtingai interpretuojamas priklausomai nuo argumentų tipo (LISP, Smalltalk).

Perkrovimas – apibrėžiamos kelios funkcijos su vienodais vardais. Tam tikroje programos vietoje reikalingos funkcijos pasirinkimas vyksta atsižvelgiant į:

- argumentų tipą (vadinamas *parametrinis perkrovimas*); C++ leidžia parametrinį perkrovimą klasių išorėje;
- vardų galiojimo sritį (modulio, klasės, failo ir pan. viduje).

Apibrėžimas iš naujo (paprastas polimorfizmas) – naudojamas kai klasių hierarchijoje metodai apibrėžti skirtingai. Konkretus metodas pasirenkamas (t.y., jo identifikatorius surišamas su atminties adresu) pagal objekto tipą kompiliacijos metu (*statinis polimorfizmas*, *ankstyvas*

sujungimas, angl. *early binding*), o patys tokie metodai vadinami *paprastais polimorfiniais*.

Polimorfiniai objektai (sudėtingas polimorfizmas) – polimorfiniais kintamaisiais (objektais) vadinami kintamieji, kuriems programos vykdymo metu gali būti priskirtą kito (skirtingo nuo apibrėžto kintamojo tipo) duomenų tipo reikšmė. Kadangi polimorfinio objekto tipas tampa žinomas tik programos vykdymo metu, jo metodo aspektas taip pat turi būti pasirenkamas vykdymo metu. Tam OOP kalbose įgyvendinamas *vėlyvo sujungimo (*angl. *late binding)* arba *dinaminio polimorfizmo* mechanizmas, leidžiantis nustatyti objekto tipą ir atitinkamą metodo aspektą programos vykdymo metu. Tokie metodai dar vadinami *virtualiais polimorfiniais*. Klasė, kurioje yra bent vienas virtualus metodas (funkcija), vadinama *polimorfine*.

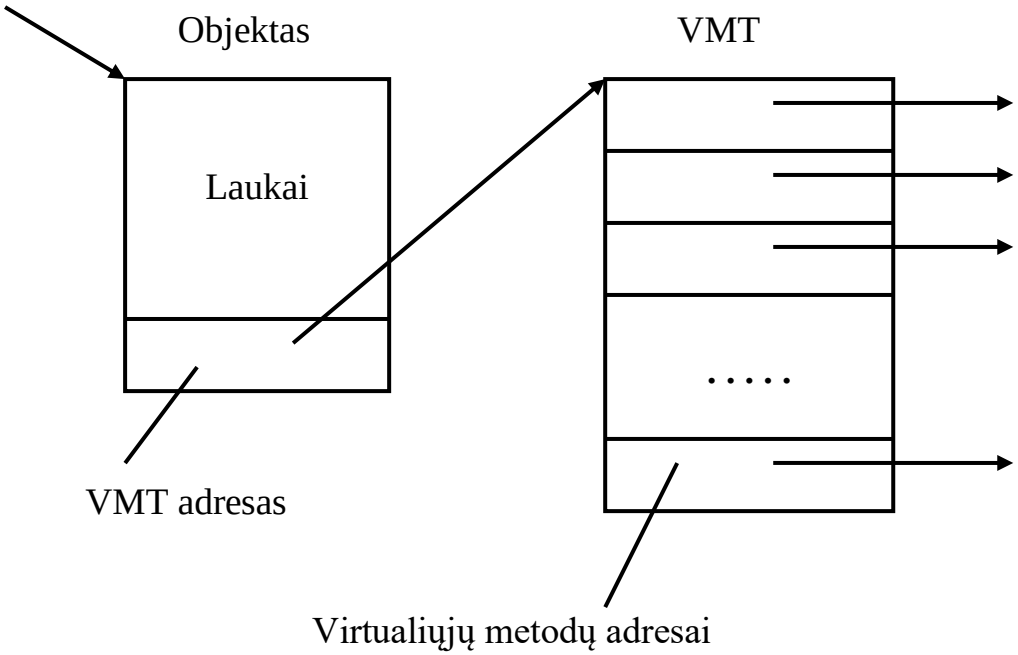
Iš tikrųjų, kai kompiliatorius (surišėjas) aptinka virtualų metodą, jis jį tik pažymi, atidedant jo identifikatoriaus surišimą su atminties adresu vėlesniam (vykdymo) laikui.

Apibendrintos funkcijos arba šablonai – naudojami parametrinių klasių realizacijai. Parametrinių klasių parametrai C++ kalboje – tai metodų argumentų duomenų tipai.

Polimorfizmas yra glaudžiai susijęs su paveldėjimu (paveldimumu). Prisiminkime, kad C++ leidžia apibrėžti rodyklę į bazinę klasę, kuri, iš tikrųjų, rodys ne tik į bazinės klasės objektą, bet į bet kurį objektą išvestinės klasės. Kompiliavimo metu dar nėra žinoma, kaip programuotojas pasinaudos šia rodykle į bazinę klasę ir kurios klasės objektą norės sukurti. Taigi, rodyklė bus surišta su savo objektu tik programos vykdymo metu (dinamiškai).

Kaip tai įgyvendinama? Kiekvienam polimorfiniam duomenų tipui kompiliatorius kuria vadinamąją *virtualiųjų funkcijų lentelę* (angl. *Virtual Function/Method Table*, *VFT* arba *VMT*), o į kiekvieną tokios klasės objektą įrašo rodyklę į ją. Šioje VMT saugomi atitinkamo objekto virtualiųjų funkcijų adresai (žr. 8.1 pav.).

VMT mechanizmo realizacija gali būti kiek skirtinga skirtinguose kompiliatoriuose. Taip *MS Visual C++* lentelė vadinama ***vftable***, o rodyklė į ją – ***vfptr***. Kompiliatorius automatiškai įterpia į polimorfinės klasės konstruktoriaus pradžią tokį kodą, kuris inicializuoja ***vfptr***. Jei kviečiama virtuali funkcija, šis kodas, pirma, suranda rodyklę į VMT ir toliau lentelėje suranda atitinkamos funkcijos adresą ir iškviečia ją (žr. 8.1 pav.) .



Pav. 8.1. Vėlyvo sujungimo įgyvendinimas naudojant VMT.

Prisiminkime, kad kuriant išvestinės klasės objektą, pirma, iškviečiamas bazinės klasės konstruktorius. Būtent tada ir kuriama VMT bei rodyklę į ją. Po išvestinės klasės konstruktoriaus iškvietimo, rodyklė nukreipiama į naujai apibrėžtos virtualios funkcijos variantą (jei toks yra atitinkamam objektui).

Apibendrinant, reikėtų pabrėžti, kad už lankstumą „reikia sumokėti“: objektų su virtualiomis funkcijomis naudojimas reikalauja daugiau atminties, tokios programos vykdomos lėčiau.

Virtualios funkcijos. Tai funkcijos, kurių realizacija negali būti apibrėžta bendrai, o tik konkrečiu atveju. Taip, jei išvestinė klasė apibrėžia iš naujo bazinės klasės virtualią funkciją, funkcija iš išvestinės klasės bus iškviečiama net tada, kai naudojama rodyklė (nuoroda) į bazinę klasę.

Pavyzdys 3.24. Virtualios funkcijos ir rodyklė į bazinę klasę.

```
#include <iostream.h>
class Bazine
{
        int x;
   public:
        Bazine (int _x) {x = _x;}
        virtual int Gauti_x() {return x;}
        virtual void Spausdinti_x() ;
};
```

```
void Bazine :: Spausdinti_x()
{
    cout << “Klaida. Ši funkcija neprieinama klasei Bazine ” << endl;
}

class Isvestine_1 : public Bazine
{
        public:
                Isvestine_1(int _x) : Bazine (_x) {}
                void Spausdinti_x() ;
};
void Isvestine_1 :: Spausdinti_x()
{
    cout << “Isvestine_1 :: x = ” << Gauti_x() << endl;
}

class Isvestine_2 : public Bazine
{
        public:
                Isvestine_2(int _x) : Bazine (_x) {}
                void Spausdinti_x() ;
};
void Isvestine_2 :: Spausdinti_x()
{
    cout << “Isvestine_2 :: x = ” << Gauti_x() << endl;
}

void main ()
{
        Isvestine_1 *pIsv1 = new Isvestine_1(10);
        Isvestine_2 *pIsv2 = new Isvestine_2(20);
        Bazine* pBaz = pIsv1;
        pBaz –> Spausdinti_x();
        pBaz = pIsv2;
        pBaz –> Spausdinti_x();
}
```

Išvestinių klasių funkcijos *Spausdinti_x()* yra virtualios, nes bazinėje klasėje ši funkcija paskelbta kaip virtuali. Tad kiekvienam objektui iškviečiama atitinkamai perkrauta bazinės klasės funkcija. Šios programos vykdymo rezultatai:

```
Isvestine_1 :: x = 10
Isvestine_2 :: x = 20
```

Jei būtų paskelbta klasė, kurioje ši funkcija nebūtų perkrauta, būtų kviečiama bazinės klasės funkcija *Spausdinti_x()*.

Bazinis žodis *virtual* gali būti naudojamas ir išvestinėse klasėse, tačiau jis nėra būtinas, nes perkrautos virtualios funkcijos lieka virtualiomis. Bazinių klasių virtualios funkcijos turi būti jose apibrėžtos. Išvestinės klasės funkcija perkraus bazinės klasės funkciją tik tada, kai turi tokį pat prototipą (turi tą patį vardą, tą patį ir tokių pat tipų parametrų skaičių) kaip ir bazinės klasės virtuali funkcija. Net jei išvestinės klasės funkcija skiriasi tik vieno parametro tipu, manoma, kad tai visiškai nauja funkcija ir perkrovimas nevyksta (žr. pavyzdį 3.25).

Pavyzdys 3.25. Virtualiųjų funkcijų perkrovimas.

```
#include <iostream.h>

class Bazine
{
        int x;
   public:
        virtual void Nustatyti(int  _x)
        {
          x = _x;
          cout << “Bazine :: x = ” << x << endl;
        }
        virtual void Rodyti (Bazine* pOb)
        {
          Nustatyti(10);
        }
};

class Isvestine : public Bazine
{
        int x, y;
   public:
        virtual void Nustatyti(int  _x, int _y)
        {
          x = _x;   y = _y;
          cout << “Isvestine :: x = ” << x << “Isvestine :: y = ” << y << endl;
        }
        virtual void Rodyti(Bazine* pOb)
        {
          Nustatyti(15, 20);
        }
```

```
}
int main ()
{
        Bazine* pOb1 = new Bazine;
        Bazine* pOb2 = new Isvestine;
        pOb1 –> Rodyti(pOb1);   //Virtuali funkcija kviečiama iš klasės Bazine
        pOb2 –> Rodyti(pOb1);   //Virtuali funkcija kviečiama iš klasės Isvestine
        pOb2 –> Rodyti(pOb2);   //Virtuali funkcija kviečiama iš klasės Isvestine
        return 0;
}
```

Šios programos vykdymo rezultatai:

```
Bazine :: x = 10
Isvestine :: x = 15  Isvestine :: y = 20
Isvestine :: x = 15  Isvestine :: y = 20
```

Čia, ir bazinėje, ir išvestinėje klasėje yra po dvi virtualios funkcijos su vienodais vardais. Tačiau, kompiliatorius skirtingai jas interpretuoja: funkcija *Nustatyti()* išvestinėje klasėje suprantama kaip visiškai nauja virtuali funkcija ir perkrovimas nevyksta, o bazinės klasės funkcija *Rodyti()* yra perkraunama išvestinėje klasėje.

Pavyzdys 3.26. Virtualiųjų ir nevirtualiųjų funkcijų palyginimas.

```
#include <iostream.h>

class Bazine
{
   public:
        virtual void VirtualiFunkcija()
        { cout << “Mes funkcijoje  Bazine::VirtualiFunkcija ” << endl; }
        void NevirtualiFunkcija()
        { cout << “Mes funkcijoje  Bazine::NevirtualiFunkcija ” << endl; }
};

class Isvestine : public Bazine  //Abi dvi funkcijos čia perkraunamos
{
   public:
        virtual void VirtualiFunkcija()
        { cout << “Mes funkcijoje  Isvestine::VirtualiFunkcija ” << endl; }
        void NevirtualiFunkcija()
        { cout << “Mes funkcijoje  Isvestine::NevirtualiFunkcija ” << endl; }
};
```

```
int main ()
{
        Bazine* pBase = new Isvestine;
        pBase–>VirtualiFunkcija();     //Kviečiama virtuali funkcija iš klasės Isvestine
        pBase–>NevirtualiFunkcija(); //Kviečiama nevirtuali funkcija iš klasės Bazine
        return 0;
}
```

Šios programos vykdymo rezultatai:

```
Mes funkcijoje  Isvestine::VirtualiFunkcija
Mes funkcijoje  Bazine::NevirtualiFunkcija
```

Rodyklė pBase tokiu būdu gali iškviesti atitinkamą nevirtualią funkciją iš bazinės klasės.

Vėlyvo surišimo mechanizmą galima išjungti, jei krepinyje į funkciją nurodyti pilną kreipinį su matomumo praplėtimo operacija :: (žr. pvz. 3.27).

Pavyzdys 3.27. Vėlyvas surišimas.

```
#include <iostream.h>
class Bazine
{
   public:
        virtual void VirtualiFunkcija()
        { cout << “Mes funkcijoje  Bazine::VirtualiFunkcija ” << endl; }
};
class Isvestine : public Bazine  //Abi dvi funkcijos čia perkraunamos
{
   public:
        virtual void VirtualiFunkcija()
        { cout << “Mes funkcijoje  Isvestine::VirtualiFunkcija ” << endl; }
};
int main ()
{
        Bazine* pBase = new Isvestine;
        pBase–>VirtualiFunkcija();     //Kviečiama virtuali funkcija iš klasės Isvestine
        //Kviečiama virtuali funkcija iš klasės Bazine, vėlyvas sujungimas atjungtas
        pBase–>Base::VirtualiFunkcija();
        return 0;
}
```

Šios programos vykdymo rezultatai:

Mes funkcijoje Isvestine::VirtualiFunkcija
Mes funkcijoje Bazine::VirtualiFunkcija

Tai yra todėl, kad paskutinis kreipinys
pBase–>Base::VirtualiFunkcija();
išjungia vėlyvo surišimo mechanizmą.

Virtualūs destruktoriai. Skirtingai nuo konstruktorių, destruktoriai gali būti virtualūs. Jei šalinamas išvestinės klasės objektas, į kurį rodo bazinės klasės rodyklė ir destruktorius paskelbtas kaip virtualus, tada bus iškviestas išvestinės klasės destruktorius ir tik po to bazinės klasės destruktorius. Tokiu atveju, objektas bus korektiškai pašalintas (visas). Priešingu atveju, būtų iškviestas tik bazinės klasės destruktorius ir atmintyje gali likti „šiukšlės“.
Pastabos. Jei bazinė klasė turi virtualių funkcijų, reikia naudoti virtualų destruktorių. Konstruktoriai negali būti virtualūs.

Abstrakčios klasės ir grynai virtualios funkcijos.

Abstrakčios klasės neturi savo atstovų – objektų. Jos egzistuoja tik tam, kad galima būtų kurti išvestinės klases, pasižyminčios bendromis savybėmis ir funkcionalumu. Abstrakčioji klasė turi turėti bent vieną *grynai virtualią funkciją*. Išvestinėse klasėse turi būti pateikta tokių grynai virtualių funkcijų realizacija, kitaip šios klasės irgi tampa abstrakčiomis. Grynai virtualios funkcijos apibrėžimo forma:

virtual <Funkcijos_vardas> (<parametrų_sąrašas>) = 0;

Paprastai, abstrakčios klasės objektai yra per bendri, kad būtų naudingi. Bet jie užtikrina bendras savybes klasių hierarchijoje. Pvz. 3.26 bet kuri išvestinė klasė turės funkcijos Spausdinti_x() realizaciją.

Pavyzdys 3.28. Abstrakti klasė.

```
class Bazine
{
        int x;
   public:
        Bazine (int _x) {x = _x;}
        virtual int Gauti_x() {return x;}
        virtual void Spausdinti_x() = 0; // Grynai virtuali funkcija
};
```

Abstrakčiųjų klasių naudojimo ypatumai:

- abstrakčioji klasė negali būti kitos klasės kintamuoju – nariu;
- abstrakčioji klasė negali būti naudojama kaip funkcijai perduodamų argumentų tipas;
- abstrakčioji klasė negali būti naudojama kaip funkcijos gražinamos reikšmės tipas;
- negalima skelbti abstrakčios klasės objektų;
- negalima atlikti tiesioginio objekto tipo transformavimo į abstrakčios klasės tipą;
- galima paskelbti rodyklę arba nuorodą į abstrakčią klasę.