Osa

V

18. oppitunti

Kehittynyt monimuotoisuus

Edellisissä luvuissa käsiteltiin virtuaalifunktioiden luomista johdetuissa luokissa. Tämä on monimuotoisuuden peruspilari: mahdollisuus sitoa tietyn, johdetun luokan oliot perusluokan osoittimiin ajon aikana. Tässä luvussa käsitellään seuraavia aiheita:

- ❏ Mitä muunnos tarkoittaa ja se saatetaan toteuttaa
- ❏ Mitä ovat abstraktit tietotyypit
- ❏ Mitä ovat puhtaat virtuaalifunktiot

Yksittäisen periytymisen ongelmat

Aiemmissa luvuissa puhuttiin johdettujen luokkien käsittelemisestä monimuotoisesti niiden perusluokissa. Näit, että, jos perusluokalla on metodi Speak(), joka korvataan johdetussa luokassa, peruskohteen osoitin,

joka sijoitetaan johdettuun kohteeseen, toimii oikein. Listaus 18.1 havainnollistaa tätä ajatusta.

Listaus 18.1. Virtuaalimetodit.

```
1:  // Listaus 18.1 - virtuaalimetodit
2:  #include <iostream.h>
3:  class Mammal
4:  {
5:  public:
6:    Mammal():itsAge(1) { cout << "Mammal constructor...\n"; }
7:    ~Mammal() { cout << "Mammal destructor...\n"; }
8:    virtual void Speak() const { cout << "Mammal speak!\n"; }
9:  protected:
10:   int itsAge;
11: };
12:
13: class Cat: public Mammal
14: {
15: public:
16:   Cat() { cout << "Cat constructor...\n"; }
17:   ~Cat() { cout << "Cat destructor...\n"; }
18:   void Speak()const { cout << "Meow\n”; }
19: };
20:
21: int main()
22: {
23:   Mammal *pCat = new Cat;
24:   pCat ->Speak();
25:   return 0;
26: }
```

Tulostus

```
Mammal constructor…
Cat constructor…
Meow!
```

Analyysi

Rivillä 8 esitellään Speak() virtuaalisena metodina; se korvataan riveillä 19 ja sitä käytetään rivillä 25. Huomaa, että pCat esitellään Mammal-osoittimena, mutta siihen sijoitetaan Cat-osoite. Kyseessä on, kuten todettiin luvussa 17, "Monimuotoisuus ja johdetut luokat", monimuotoisuuden ydin.

Mitä kuitenkin tapahtuu, jos haluat lisätä Cat-luokkaan metodin, joka ei sovi Mammal-luokkaan? Olettakaamme esimerkiksi, että haluat lisätä metodin Purr(). Kissa kehrää, mutta muut elukat eivät tee niin. Voisit esitellä luokkasi seuraavasti:

```
class Cat: public Mammal
{
public:
    Cat() cout << "Cat constructor…\n";}
    ~Cat() { cout << "Cat destructor…\n";}
    void Speak() const { cout << "meow\n";}
```

```
    void Purr()const { cout << "rrrrrrrrrrrrrrr\n";}
};
```

Ongelma on tämä: jos nyt osoittimella kutsutaan Purr()-metodia, antaa kääntäjä virheilmoituksen:

```
error C2039: 'Purr' : is not a member of 'Mammal'
```

Kun kääntäjä yrittää ottaa Purr()-metodin omasta Mammal-v-taulukostaan, ei siellä ole tietoa.

Voit viedä tämän metodin perusluokkaan, mutta se on huono ajatus. Vaikka se toimisikin, perusluokan kansoittaminen metodeille, jotka eivät kuulu yleisesti johdettuihin luokkiin, on huonoa ohjelmointia ja tuottaa vaikeasti ylläpidettävää koodia.

Itse asiassa koko tämä ongelma on huonon suunnittelun tulosta. Yleisesti puhuen, jos sinulla on perusluokkaan osoitin, johon sijoitetaan johdetun luokan olio, se tehdään siksi, että aiot käyttää tuota oliota monimuotoisesti eikä sinun tässä tapauksessa pitäisi edes yrittää käyttää metodeita, jotka kuuluvat johdettuun luokkaan.

Ja vielä tarkemmin: ongelma ei ole siinä, että sinulla on tiettyjä metodeita, vaan siinä, että yrität käsitellä niitä perusluokan osoittimella. Ihanteena on, ettet yrittäisi käsitellä noita metodeita sellaisella osoittimella.

Mutta todellisuudessa meillä on esimerkiksi joukko perusolioita, vaikkapa koko eläintarha täynnä Mammal-olioita. Sitten huomaat, että sinulla on Cat-olio, jonka tulisi kehrätä. Nyt meillä saattaa olla vain yksi asia tehtävänä: petkuttaminen.

Petkuttaminen tapahtuu seuraavasti: muunnat perusluokan osoittimesi johdetun luokan osoittimeksi. Kerrot siis kääntäjälle: "Minä satun tietämään, että kyseessä on todellakin kissa, joten tee niin kuin sanon". Sinä siis pakotat Cat-käytöksen esille Mammal-osoittimesta.

Tämän toteuttamiseen käytät uutta dynaamisen muunnoksen operaattoria. Tämä operaattori takaa, että muunnos tapahtuu turvallisesti. Lisäksi se helpottaa löytämään nopeasti nuo koodipaikat, joissa olet käyttänyt tuota piirrettä, jotta voit poistaa ne nopeasti. Se toimii seuraavasti.

Jos meillä on osoitin perusluokkaan, kuten Mammal-luokkaan ja sijoitat siihen osoittimen johdettuun luokkaan, kuten Cat-luokkaan, voi käyttää Mammal-osoitinta monimuotoisesti. Jos sinun on sitten voitava kutsua Cat-oliota, vaikkapa sen purr()-metodia, luot Cat-osoittimen ja käytät dynaamisen muuntamisen operaattoria tekemään muunnoksen.

Ajon aikana tutkitaan perusosoitin. Jos muunnos on sopiva, uusi Cat-osoitin on hieno. Jos muunnos ei ole sopiva, jos Cat-oliota ei olekaan, on uusi osoitin null. Listaus 18.2 havainnollistaa tätä.

Listaus 18.2. Dynaaminen muunnos.

```
1:  // Listais 18.2 - dynaaminen muunnos
2:
3:
4:  #include <iostream.h>
5:  class Mammal
6:  {
7:  public:
8:    Mammal():itsAge(1) { cout << "Mammal constructor...\n"; }
9:    ~Mammal() { cout << "Mammal destructor...\n"; }
10:   virtual void Speak() const { cout << "Mammal speak!\n"; }
11: protected:
12:   int itsAge;
13: };
14:
15: class Cat: public Mammal
16: {
17: public:
18:   Cat() { cout << "Cat constructor...\n"; }
19:   ~Cat() { cout << "Cat destructor...\n"; }
20:   void Speak()const { cout << "Meow\n"; }
21:   void Purr() const { cout << "rrrrrrrrrr\n"; }
22: };
23:
24: class Dog: public Mammal
25: {
26: public:
27:   Dog() { cout << "Dog Constructor...\n"; }
28:   ~Dog() { cout << "Dog destructor...\n"; }
29:   void Speak()const { cout << "Woof!\n"; }
30: };
31:
32:
33: int main()
34: {
35:   const int NumberMammals = 3;
36:   Mammal* Zoo[NumberMammals];
37:   Mammal* pMammal;
38:   int choice,i;
39:   for (i=0; i<NumberMammals; i++)
40:   {
41:     cout << "(1)Dog (2)Cat: ";
42:     cin >> choice;
43:     if (choice == 1)
44:       Mammal = new Dog;
45:     else
46:       pMammal = new Cat;
47:
48:     Zoo[i] = pMammal;
49:   }
50:
51:     cout << "\n";
52:
53:     for (i=0; i<NumberMammals; i++)
54:     {
55:       Zoo[i] ->Speak();
```

```
56:
57:     Cat *pRealCat = 0;
58:     pRealCat = dynamic_cast< Cat *> (Zoo[i]);
59:
60:     if (pRealCat)
61:       pRealCat->Purr();
62:     else
63:       cout << "Uh oh, not a cat!\n";
64:
65:     delete Zoo[i];
66:     cout << "\n";
67:   }
68:
69:   return 0;
70: }
```

Tulostus

```
(1)Dog (2)Cat: 1
Mammal constructor…
Dog constructor…
(1)Dog (2)Cat: 2
Mammal constructor…
Dog constructor…
(1)Dog (2)Cat: 1
Mammal constructor…
Dog constructor…

Woof!
Uh oh, not a cat!
Mammal constructor…

Meow!
rrrrrrrrrrrrrr
Mammal constructor…

Woof!
Uh oh, not a cat!
Mammal constructor…
```

Analyysi

Riveillä 39-49 pyydetään käyttäjää lisäämään joko Cat- tai Dog-olio Mammal-osoittimien taulukkoon. Rivi 53 käy läpi taulukon ja rivillä 55 kutsutaan kunkin olion virtuaalia speak()-metodia. Nämä metodit vastaavat monimuotoisesti: kissat naukuvat ja koirat sanovat woof!

Rivillä 61 halutaan Cat-olion kehräävän, mutta vastaavaa metodia ei haluta kutsua Dog-olioille. Nyt käytetään dynaamisen muunnoksen operaattoria rivillä 58 takaamaan, että kutsuttava olio, joka siis kehrää (purr()), on Cat-olio. Jos näin on, ei osoitin ole null ja se menee läpi rivin 60 testin.

Abstraktit tietotyypit

Usein luokista halutaan luoda hierarkia. Haluamme esimerkiksi luoda Shape-luokan (shape = kuvio) ja johtaa siitä Rectangle (suorakaide) ja Circle (ympyrä). Rectangle-tyypistä voidaan johtaa Square (neliö) suorakaiteen erikoistapauksena.

Kukin johdettu luokkaa korvaa Draw()-metodin, GetArea()-metodin, jne. Listaus 18.3 havainnollistaa Shape-luokan ja siitä johdettujen Circle- ja Rectangle-luokkien toteuttamista.

Listaus 18.3. Shape-luokat.

```
1:  //Listaus 18.3. Shape-luokat.
2:
3:  #include <iostream.h>
4:
5:  enum BOOL { FALSE, TRUE };
6:
7:  class Shape
8:  {
9:  public:
10:   Shape(){}
11:   ~Shape(){}
12:   virtual long GetArea() { return -1; } // error
13:   virtual long GetPerim() { return -1; }
14:   virtual void Draw() {}
15: private:
16: };
17:
18: class Circle : public Shape
19: {
20: public:
21:   Circle(int radius):itsRadius(radius){}
22:   ~Circle(){}
23:   long GetArea() { return 3 * itsRadius * itsRadius; }
24:   long GetPerim() { return 9 * itsRadius; }
25:   void Draw();
26: private:
27:   int itsRadius;
28:   int itsCircumference;
29: };
30:
31: void Circle::Draw()
32: {
33:   cout << "Circle drawing routine here!\n";
34: }
35:
36:
37: class Rectangle : public Shape
38: {
39: public:
40:   Rectangle(int len, int width):
41:   itsLength(len), itsWidth(width){}
42:   ~Rectangle(){}
43:   virtual long GetArea() { return itsLength * itsWidth; }
44:   virtual long GetPerim() {return 2*itsLength + 2*itsWidth; }
45:   virtual int GetLength() { return itsLength; }
46:   virtual int GetWidth() { return itsWidth; }
```

```
47:   virtual void Draw();
48: private:
49:   int itsWidth;
50:   int itsLength;
51: };
52:
53: void Rectangle::Draw()
54: {
55:   for (int i = 0; i<itsLength; i++)
56:   {
57:     for (int j = 0; j<itsWidth; j++)
58:       cout << "x ";
59:
60:     cout << "\n";
61:   }
62: }
63:
64: class Square : public Rectangle
65: {
66: public:
67:   Square(int len);
68:   Square(int len, int width);
69:   ~Square(){}
70:   long GetPerim() {return 4 * GetLength();}
71: };
72:
73: Square::Square(int len):
74: Rectangle(len,len)
75: {}
76:
77: Square::Square(int len, int width):
78: Rectangle(len,width)
79:
80: {
81:   if (GetLength() != GetWidth())
82:     cout << "Error, not a square... a Rectangle??\n";
83: }
84:
85: int main()
86: {
87:   int choice;
88:   BOOL fQuit = FALSE;
89:   Shape * sp;
90:
91:   while (1)
92:   {
93:     cout << "(1)Circle (2)Rectangle (3)Square (0)Quit: ";
94:     cin >> choice;
95:
96:     switch (choice)
97:     {
98:       case 1: sp = new Circle(5);
99:       break;
100:      case 2: sp = new Rectangle(4,6);
101:      break;
102:      case 3: sp = new Square(5);
103:      break;
104:      default: fQuit = TRUE;
105:      break;
106:    }
107:    if (fQuit)
```

```
108:       break;
109:
110:     sp->Draw();
111:     cout << "\n";
112:   }
113:   return 0;
114: }
```

Tulostus

```
(1)Circle (2)Rectangle (3)Square (0)Quit: 2
xxxxxx
xxxxxx
xxxxxx
xxxxxx

(1)Circle (2)Rectangle (3)Square (0)Quit: 3
xxxxxxxxxx
xxxxxxxxxx
xxxxxxxxxx
xxxxxxxxxx
xxxxxxxxxx

(1)Circle (2)Rectangle (3)Square (0)Quit: 0
```

Analyysi

Riveillä 7-16 esitellään Shape-luokka. Metodit GetArea() ja GetPerim() palauttavat virhearvon eikä Draw() toimi. Mitä loppujen lopuksi kuvion piirtäminen tarkoittaa? Kaikenlaisia kuvioita voidaan piirtää (ympyröitä, suorakaiteita jne), mutta kuviota abstraktina käsitteenä ei voida piirtää.

Circle johdetaan Shape-luokasta ja siinä korvataan kolme virtuaalimetodia. Huomaa, että ei ole mitään syytä lisätä sanaa virtuaalinen, koska se on osa virtuaalimetodien periytymistä. Mutta sen käyttämisestä ei ole harmiakaan, kuten esitetään Rectangle-luokassa riveillä 43, 44 ja 47. On hyvä sisällyttää sana virtual ikään kuin osaksi dokumentaatiota.

Square johdetaan Rectangle-luokasta ja se myöskin korvaa GetPerim()-metodin perien loput Rectangle-luokassa määritellyistä metodeista.

On kuitenkin ongelmallista, että asiakas saattaa yrittää luoda Shape-olion ja olisi mukava tehdä se mahdottomaksi. Shape-luokka on vain liittymän tarjoaja siitä johdetuille luokille ja siten se on abstrakti tietotyyppi (ADT, Abstract Data Type).

Abstrakti tietotyyppi edustaa käsitettä (kuten kuvio) eikä niinkään oliota (kuten ympyrä). C++ -kielessä on ADT aina muiden luokkien perusluokka eikä ole sopivaa luoda ADT-olio.

Puhtaat virtuaalifunktiot

C++ mahdollistaa abstraktien tietotyyppien luomisen puhtailla virtuaalifunktioilla. Virtuaalifunktio tehdään puhtaaksi alustamalla se nollalla:

```
virtual void Draw() = 0;
```

Jokainen luokka, jossa on yksi tai useampia puhtaita virtuaalifunktioita, on ADT ja on laitonta luoda ADT-luokan olioita. Sen yrittäminen aiheuttaa kääntämisvirheen. Puhtaan virtuaalifunktion sijoittaminen luokkaan viestittää kahta asiaa luokan asiakkaille:

- Älä luo oliota tästä luokasta, vaan johda siitä luokkia
- Varmista, että puhdas virtuaalifunktio tulee korvatuksi

Jokainen luokka, joka johdetaan ADT-luokasta, perii puhtaan virtuaalifunktion puhtaana ja sen tulee korvata kaikki virtuaalifunktiot voidaakseen luoda olioita. Siksi, jos Rectangle periytyy Shape-luokasta, jossa on kolme puhdasta virtuaalifunktiota, täytyy Rectangle-luokan korvata kaikki kolme tai se on myöskin ADT. Listaus 18.4 muuttaa Shape-luokan abstraktiksi. Tilan säästämiseksi ei muuta koodia ole mukana. Korvaa listauksen 18.3 Shape-esittely (rivit 7-16) listauksen 18.4 Shape-esittelyllä ja aja ohjelma uudelleen.

Listaus 18.4. Abstraktit tietotyypit.

```
1:  class Shape
2:  {
3:  public:
4:    Shape(){}
5:    ~Shape(){}
6:    virtual long GetArea() = 0;
7:    virtual long GetPerim()= 0;
8:    virtual void Draw() = 0;
9:  private:
10: };
```

Tulostus

```
(1)Circle (2)Rectangle (3)Square (0)Quit:2
xxxxxx
xxxxxx
xxxxxx
xxxxxx
(1)Circle (2)Rectangle (3)Square (0)Quit:3
xxxxxxx
xxxxxxx
xxxxxxx
xxxxxxx
(1)Circle (2)Rectangle (3)Square (0)Quit:0
```

Analyysi

Kuten voit nähdä, ohjelman toiminta ei muutu lainkaan. Ainoa ero on siinä, että nyt olisi mahdotonta muodostaa Shape-luokan olio.

Abstraktit tietotyypit

Luokka esitellään abstraktina sijoittamalla yksi tai useampia puhtaita virtuaalifunktioita luokan esittelyyn. Puhdas virtuaalifunktio esitellään kirjoittamalla = 0 funktion esittelyn jälkeen.

Esimerkiksi

```
class Shape
{
  virtual void Draw() = 0;   // puhdas virtuaalifunktio
};
```

Puhtaiden virtuaalifunktioiden toteuttaminen

Yleensä abstraktin luokan puhtaita virtuaalifunktioita ei koskaan toteuteta. Koska yhtään tuon tyyppistä oliota ei koskaan luoda, ei ole syytä antaa määrittelyäkään ja ADT toimii pelkästään käyttöliittymänä, josta johdetaan uusia olioita.

On kuitenkin mahdollista antaa puhtaan virtuaalifunktion määrittely. Sen jälkeen ADT-luokasta johdetut oliot voivat kutsua sitä ja ehkäpä korvata funktion halutulla toiminnallisuudella. Listaus 18.5 on muutettu listaus 18.3. Tällä kertaa listauksessa on Shape-luokka abstraktina ja puhdas virtuaalifunktio, Draw(), on toteutettu. Circle-luokka korvaa Draw()-metodin, kuten pitääkin, mutta sitten se hyödyntää perusluokan funktiota lisätoiminnallisuuden saavuttamiseksi.

Tässä esimerkissä on lisätoimintona yksinkertainen tulostus, mutta voit kuvitella, että perusluokka tarjoaa jaetun piirtomekanismin, asettaen ehkäpä ikkunan, jota kaikki johdetut luokat käyttävät.

Listaus 18.5. Puhtaiden virtuaalifunktioiden toteuttaminen.

```
1:  //Puhtaat virtuaalifunktiot
2:
3:  #include <iostream.h>
4:
5:  enum BOOL { FALSE, TRUE };
6:
7:  class Shape
8:  {
9:  public:
10:   Shape(){}
11:   ~Shape(){}
12:   virtual long GetArea() = 0; // error
13:   virtual long GetPerim()= 0;
14:   virtual void Draw() = 0;
15: private:
```

```
16: };
17:
18: void Shape::Draw()
19: {
20:   cout << "Abstract drawing mechanism!\n";
21: }
22:
23: class Circle : public Shape
24: {
25: public:
26:   Circle(int radius):itsRadius(radius){}
27:   ~Circle(){}
28:   long GetArea() { return 3 * itsRadius * itsRadius; }
29:   long GetPerim() { return 9 * itsRadius; }
30:   void Draw();
31: private:
32:   int itsRadius;
33:   int itsCircumference;
34: };
35:
36: void Circle::Draw()
37: {
38:   cout << "Circle drawing routine here!\n";
39:   Shape::Draw();
40: }
41:
42:
43: class Rectangle : public Shape
44: {
45: public:
46:   Rectangle(int len, int width):
47:   itsLength(len), itsWidth(width){}
48:   ~Rectangle(){}
49:   long GetArea() { return itsLength * itsWidth; }
50:   long GetPerim() {return 2*itsLength + 2*itsWidth; }
51:   virtual int GetLength() { return itsLength; }
52:   virtual int GetWidth() { return itsWidth; }
53:   void Draw();
54: private:
55:   int itsWidth;
56:   int itsLength;
57: };
58:
59: void Rectangle::Draw()
60: {
61:   for (int i = 0; i<itsLength; i++)
62:   {
63:     for (int j = 0; j<itsWidth; j++)
64:       cout << "x ";
65:
66:     cout << "\n";
67:   }
68:   Shape::Draw();
69: }
70:
71:
72: class Square : public Rectangle
73: {
74: public:
75:   Square(int len);
76:   Square(int len, int width);
```

```
77:    ~Square(){}
78:    long GetPerim() {return 4 * GetLength();}
79: };
80:
81: Square::Square(int len):
82: Rectangle(len,len)
83: {}
84:
85: Square::Square(int len, int width):
86: Rectangle(len,width)
87:
88: {
89:   if (GetLength() != GetWidth())
90:     cout << "Error, not a square... a Rectangle??\n";
91: }
92:
93: int main()
94: {
95:   int choice;
96:   BOOL fQuit = FALSE;
97:   Shape * sp;
98:
99:   while (1)
100:  {
101:    cout << "(1)Circle (2)Rectangle (3)Square (0)Quit: ";
102:    cin >> choice;
103:
104:    switch (choice)
105:    {
106:      case 1: sp = new Circle(5);
107:      break;
108:      case 2: sp = new Rectangle(4,6);
109:      break;
110:      case 3: sp = new Square (5);
111:      break;
112:      default: fQuit = TRUE;
113:      break;
114:    }
115:    if (fQuit)
116:      break;
117:
118:    sp->Draw();
119:    cout << "\n";
120:  }
121:  return 0;
122: }
```

Tulostus

```
(1)Circle (2)Rectangle (3)Square (0)Quit:2
XXXXXX
XXXXXX
XXXXXX
XXXXXX
Abstract drawing mechanism!
(1)Circle (2)Rectangle (3)Square (0)Quit:3
XXXXXXX
XXXXXXX
XXXXXXX
XXXXXXX
```

```
Abstract drawing mechanism!
(1)Circle (2)Rectangle (3)Square (0)Quit:0
```

Analyysi
Riveillä 7-16 esitellään abstrakti Shape-luokka, jonka kaikki kolme metodia ovat puhtaita virtuaalifunktioita. Huomaa, että tämä ei ole välttämätöntä. Jos yksikin metodi esitellään puhtaasti virtuaaliseksi, olisi luokka ADT.

Metodeja GetArea() ja GetPerim() ei toteuteta, mutta Draw() toteutetaan. Sekä Circle että Rectangle korvaavat Draw()-metodin ja ne käyttävät perusluokan metodia hyödyntääkseen perusluokan jaettua toiminnallisuutta.

Abstraktion monimutkaiset hierarkiat

Joskus halutaan johtaa ADT-luokkia toisista ADT-luokista. Ehkäpä siksi, että jostakin johdetusta puhtaasta virtuaalifunktiosta halutaan tehdä ei-puhdas ja jättää muut puhtaiksi.

Jos luot Animal-luokan, voit tehdä Eat()-, Sleep()-, Move()- ja Reproduce()-metodeista puhtaita virtuaalifunktioita. Ehkäpä johdat Animal-luokasta Mammal- ja Fish-luokat.

Tutkimusten jälkeen päätät, että jokainen Mammal tuottaa samalla tavalla, joten teet metodista Mammal::Reproduce ei-puhtaan, mutta annat muiden metodien olla puhtaita virtuaalifunktioita.

Johdat Mammal-luokasta Dog-luokan, jolloin Dogin on korvattava ja toteutettava kolme jäljelle jäänyttä puhdasta virtuaalifunktiota, jotta Dog-olioita voitaisiin luoda.

Olet siis luokan suunnittelijana aikaansaanut sen, että Animal- tai Mammal-olioita ei voida luoda, mutta kaikki Mammal-oliot voivat periä Reproduce()-metodin korvaamatta sitä.

Listaus 18.6 havainnollistaa tätä menettelyä käyttäen karsittuja luokkamäärittelyjä.

Listaus 18.6. ADT-luokkien johtaminen toisista ADT-luokista.

```
1:  // Listaus 18.6
2:  // Johdetaan ADT-luokkia toisista ADT-luokista
3:  #include <iostream.h>
4:
5:  enum COLOR { Red, Green, Blue, Yellow, White, Black, Brown } ;
6:  enum BOOL { FALSE, TRUE };
7:
8:  class Animal        // common base to both horse and bird
9:  {
10: public:
11:   Animal(int);
```

```
12:   virtual ~Animal() { cout << "Animal destructor...\n"; }
13:   virtual int GetAge() const { return itsAge; }
14:   virtual void SetAge(int age) { itsAge = age; }
15:   virtual void Sleep() const = 0;
16:   virtual void Eat() const = 0;
17:   virtual void Reproduce() const = 0;
18:   virtual void Move() const = 0;
19:   virtual void Speak() const = 0;
20: private:
21:   int itsAge;
22: };
23:
24: Animal::Animal(int age):
25: itsAge(age)
26: {
27:   cout << "Animal constructor...\n";
28: }
29:
30: class Mammal : public Animal
31: {
32: public:
33:   Mammal(int age):Animal(age){ cout << "Mammal constructor...\n";}
34:   ~Mammal() { cout << "Mammal destructor...\n";}
35:   virtual void Reproduce() const
36:     { cout << "Mammal reproduction depicted...\n"; }
37: };
38:
39: class Fish : public Animal
40: {
41: public:
42:   Fish(int age):Animal(age){ cout << "Fish constructor...\n";}
43:   virtual ~Fish() {cout << "Fish destructor...\n";  }
44:   virtual void Sleep() const { cout << "fish snoring...\n"; }
45:   virtual void Eat() const { cout << "fish feeding...\n"; }
46:   virtual void Reproduce() const { cout << "fish laying eggs...\n"; }
47:   virtual void Move() const { cout << "fish swimming...\n";   }
48:   virtual void Speak() const { }
49: };
50:
51: class Horse : public Mammal
52: {
53: public:
54:   Horse(int age, COLOR color ):
55:   Mammal(age), itsColor(color) { cout << "Horse constructor...\n"; }
56:   virtual ~Horse() { cout << "Horse destructor...\n"; }
57:   virtual void Speak()const { cout << "Whinny!... \n"; }
58:   virtual COLOR GetItsColor() const { return itsColor; }
59:   virtual void Sleep() const { cout << "Horse snoring...\n"; }
60:   virtual void Eat() const { cout << "Horse feeding...\n"; }
61:   virtual void Move() const { cout << "Horse running...\n";}
62:
63: protected:
64:   COLOR itsColor;
65: };
66:
67: class Dog : public Mammal
68: {
69: public:
70:   Dog(int age, COLOR color ):
71:   Mammal(age), itsColor(color) { cout << "Dog constructor...\n"; }
72:   virtual ~Dog() { cout << "Dog destructor...\n"; }
```

```
73:    virtual void Speak()const { cout << "Whoof!... \n"; }
74:    virtual void Sleep() const { cout << "Dog snoring...\n"; }
75:    virtual void Eat() const { cout << "Dog eating...\n"; }
76:    virtual void Move() const  { cout << "Dog running...\n"; }
77:    virtual void Reproduce() const { cout << "Dogs reproducing...\n"; }
78:
79:  protected:
80:    COLOR itsColor;
81:  };
82:
83:  int main()
84:  {
85:    Animal *pAnimal=0;
86:    int choice;
87:    BOOL fQuit = FALSE;
88:
89:    while (1)
90:    {
91:      cout << "(1)Dog (2)Horse (3)Fish (0)Quit: ";
92:      cin >> choice;
93:
94:      switch (choice)
95:      {
96:        case 1: pAnimal = new Dog(5,Brown);
97:        break;
98:        case 2: pAnimal = new Horse(4,Black);
99:        break;
100:       case 3: pAnimal = new Fish (5);
101:       break;
102:       default: fQuit = TRUE;
103:       break;
104:     }
105:     if (fQuit)
106:       break;
107:
108:     pAnimal->Speak();
109:     pAnimal->Eat();
110:     pAnimal->Reproduce();
111:     pAnimal->Move();
112:     pAnimal->Sleep();
113:     delete pAnimal;
114:     cout << "\n";
115:   }
116:
117:
118:
119:   return 0;
120: }
```

Tulostus

```
(1)Dog (2)Horse  (3)Bird  (0)Quit: 1
Animal constructor…
Dog constructor…
Whoof!
Dog eating…
Dog reproducing…
Dog running…
Dog snoring…
Dog destructor…
```

```
Mammal destructor…
Animal destructor…
(1)Dog (2)Horse  (3)Bird  (0)Quit: 0
```

Analyysi
Riveillä 8-22 esitellään abstrakti Animal-luokka. Luokassa ei ole yhtään ei-puhdasta metodia itsAge-muuttujalle, jonka kaikki Animal-oliot jakavat. Siinä on viisi puhdasta virtuaalifunktiota: Sleep(), Eat(), Reproduce(), Move() ja Speak().

Mammal johdetaan Animal-luokasta ja esitellään riveillä 30-36 mitään lisäämättä. Se korvaa kuitenkin Reproduce()-metodin tarjoten yleisen tuottamismuodon kaikille Mammal-olioille. Fish-luokan tulee korvata Reproduce(), koska Fish johdetaan suoraan Animal-luokasta eikä se voi hyödyntää Mammal-luokan Reproduce-metodia.

Mammal-luokat eivät enää korvaa Reproduce()-funktiota, minkä ne voisivat vapaasti tehdä, jos ne tekevät kuten Dog rivillä 77. Fish, Horse ja Dog korvaavat jäljellä olevat puhtaat virtuaalifunktiot, joten niiden tyyppisiä olioita voidaan luoda.

Ohjelman rungossa käytetään Animal-osoitinta osoittamaan vuoron perään johdettuihin olioihin. Virtuaalimetodeja kutsutaan ja oikeata metodia kutsutaan johdetussa luokassa osoittimen ajonaikaisen sidonnan perusteella.

Kääntämisvirhe syntyisi, jos yritettäisiin luoda Animal- tai Mammal-olio, koska molemmat ovat abstrakteja tyyppejä.

Mitkä tyypit ovat abstrakteja?

Yhdessä ohjelmassa Animal-luokka on abstrakti, toisessa taas ei. Mikä määrää sen, onko luokka abstrakti?

Vastaus kysymykseen syntyy ohjelman järkevyyden perusteella. Jos kirjoitat ohjelmaa, joka käsittelee maatilaa tai eläintarhaa, haluat varmaankin tehdä Animal-luokasta abstraktin, mutta Dog-luokasta haluat johtaa uusia olioita.

Toisaalta, jos luot animoidun kennelin, haluat ehkä laittaa Dog-luokan abstraktiksi ja luoda vain eri tyyppisiä koiria: noutajia, terrierejä, jne. Abstraktion taso määräytyy siitä, kuinka tarkkaan haluat erotella eri tyyppejäsi.

Tee/Älä tee

Käytä abstrakteja tyyppejä pohjana useille johdetuille tyypeille.
Korvaa kaikki puhtaat virtuaalifunktiot.
Tee jokainen korvattava funktio puhtaaksi virtuaalifunktioksi.
Älä yritä luoda oliota abstraktista tietotyypistä.

Yhteenveto

Tässä luvussa opit käyttämään dynamic_cast-operaattoria muuntamaan periytymis-hierarkiaa. Opit myös, miksi muuntaminen voi olla merkkinä huonosta luokkien suunnittelusta.

Näit, kuinka luodaan abstrakteja tietotyyppejä puhtaiden virtuaalifunktioiden avulla ja kuinka puhtaita virtuaalifunktioita toteutetaan niin, että johdetut luokat voivat käyttää niitä.

Kysymyksiä ja Vastauksia

K
Mitä toiminnallisuuden kiinnittäminen ylöspäin merkitsee?
V
Se viittaa ideaan, jossa jaettua toiminnallisuutta siirretään ylöspäin yleiseen perusluokkaan. Jos useampi kuin yksi luokka jakaa funktion, on toivottavaa löytää yhteinen perusluokka, johon tuo funktio voidaan tallentaa.

K
Onko edellä mainittu menettely aina hyvä asia?
V
Kyllä, jos viet toiminnallisuutta ylöspäin. Ei, jos siirrät vain käyttöliittymän. Jos siis kaikki johdetut luokat eivät käytä metodia, on virhe siirtää se ylöspäin yleiseen perusluokkaan. Jos teet niin, on sinun siirryttävä ajonaikaiseen oliotyyppiin ennen funktion kutsumisesta päättämistä.

K
Miksi dynaaminen muuntaminen on huono asia?
V
Virtuaalifunktioiden ytimenä on se, että virtuaalitaulukko pikemminkin kuin ohjelmoija päättää ajonaikaisen olion tyypin.

K
Miksi nähdä vaivaa abstrakteilla tietotyypeillä? Miksei tehdä kaikista niistä ei-abstrakteja, ja vältytä luomasta sen tyyppisiä olioita?
V
Monen C++ -käytännön tarkoituksena on hyödyntää kääntäjää virheiden hakemisessa, jotta virheet eivät pääsisi jakeluversioon, asiakkaalle asti. Luokan tekeminen abstraktiksi (puhtailla virtuaalifunktioilla) saa kääntäjän liputtamaan virheistä, jos tuosta tyypistä yritetään luoda olioita.