فصل
سیزدهم

برنامه‌نویسی شیی‌گرا: چندریختی

اهداف

- چند ریختی چیست و چگونه می‌تواند در برنامه‌نویسی موثر بکار گرفته شود.
- اعلان و استفاده از توابع virtual در موثرتر کردن چندریختی.
- وجه تمایز مابین کلاس های انتزاعی و مقید.
- اعلان توابع virtual محض برای ایجاد کلاس‌های انتزاعی.
- نحوه استفاده از اطلاعات نوع زمان اجرا (RTTI) به همراه تبدیل نوع dynamic_cast، typeid و type_info.
- نحوه پیاده‌سازی توابع virtual توسط C++.
- نحوه استفاده از نابود کننده‌های virtual برای حصول اطمینان از اجرای تمام نابود کننده‌های مورد نیاز بر روی یک شی.

www.yourshop.4kia.ir

رئوس مطالب

۱-۱۳ مقدمه
۲-۱۳ چند مثال از چندریختی
۳-۱۳ رابطه مابین شی‌ها در سلسله مراتب توارث
۱-۳-۱۳ احضار توابع کلاس مبنا از طریق شی‌های کلاس مشتق شده
۲-۳-۱۳ هدایت اشاره‌گرهای کلاس مشتق شده بطرف شی‌های کلاس مشتق شده
۳-۳-۱۳ فراخوانی تابع عضو کلاس مشتق شده از طریق اشاره‌گرهای کلاس مبنا
٤-۳-۱۳ توابع virtual
٥-۳-۱۳ تخصیص‌های قابل انجام مابین شی‌ها و اشاره‌گرهای کلاس مبنا و کلاس مشتق شده
٤-۱۳ عبارت switch
٥-۱۳ کلاس‌های انتزاعی و توابع virtual محض
٦-۱۳ مبحث آموزشی: سیستم پرداخت حقوق با استفاده از چند ریختی
١-٦-۱۳ ایجاد کلاس مبنای انتزاعی Employee
٢-٦-۱۳ ایجاد کلاس مشتق شده غیرانتزاعی SalariedEmployee
٣-٦-۱۳ ایجاد کلاس مشتق شده غیرانتزاعی HourlyEmplyee
٤-٦-۱۳ ایجاد کلاس مشتق شده غیرانتزاعی CommissinEmployee
٥-٦-۱۳ ایجاد غیرمستقیم مشتق شده غیرانتزاعی BasePlusCommssionEmployee
٦-٦-۱۳ شرح فرآیند چندریختی
۷-۱۳ چندریختی، توابع virtual و مقیدسازی دینامیکی
۸-۱۳ مبحث آموزشی: سیستم پرداخت حقوق با استفاده از چندریختی و اطلاعات نوع زمان اجرا با تبدیل dynamic_cast، typeid و type_info
۹-۱۳ نابود کننده virtual
۱۰-۱۳ مبحث آموزشی مهندسی نرم‌افزار: ارث‌بری در سیستم ATM

۱-۱۳ مقدمه

در فصل‌های ۱۲-۹ در ارتباط با مباحث کلیدی برنامه‌نویسی شی‌گرا و تکنولوژی‌های آن شامل کلاس‌ها، شی‌ها، کپسوله‌سازی، سربارگذاری عملگر و توارث صحبت کردیم. حال به آموزش OOP با توضیح و تفسیر مفهوم *چندریختی* (*polymorphism*) در سلسله مراتب توارث ادامه می‌دهیم. چندریختی امکان می‌دهد تا برنامه‌ها بجای اینکه «*برنامه خاصی*» باشند، حالت یک «*برنامه کلی*» داشته باشند.

در عمل، چند ریختی امکان می‌دهد تا برنامه‌هایی بنویسیم که مبادرت به پردازش شی‌ها از کلاس‌هایی کنند که بخشی از همان سلسله مراتب کلاس هستند، همچنانکه همگی آنها شی‌های از سلسله مراتب کلاس مبنا می‌باشند. همانطوری که بزودی خواهید دید، چند ریختی با *هندل‌های* (دستگیره‌های) اشاره‌گر کلاس مبنا و مراجعه‌های کلاس مبنا کاری ندارد و بر پایه نام هندل‌ها عمل می‌کند.

www.yourshop.4kia.ir

به مثالی در ارتباط با چند ریختی توجه کنید. فرض کنید می‌خواهیم برنامه‌ای بنویسیم که حرکت چند نوع حیوان را شبیه‌سازی کند. کلاس‌های Fish (ماهی)، Frog (قورباغه) و Bird (پرنده) نشان‌دهنده سه نوع حیوان تحت بررسی هستند. تصور کنید که هر یک از این کلاس‌ها از کلاس مبنای Animal ارث‌بری دارند، که حاوی یک تابع move بوده و موقعیت جاری حیوان را نگهداری می‌کند. هر کلاس مشتق شده تابع move را پیاده‌سازی می‌کند. برنامه مبادرت به نگهداری یک بردار (vector) از اشاره‌گرها به شی‌های از انواع کلاس‌های مشتق شده Animal می‌کند. برای شبیه‌سازی حرکت حیوانات، برنامه به هر شی در هر ثانیه یک پیغام بنام move ارسال می‌کند. با این وجود، هر نوع خاص از حیوان به این پیغام move (حرکت) به روش خود پاسخ می‌دهد، برای مثال ماهی قادر به شنا به میزان دو فوت، قورباغه قادر به پرش به میزان سه فوت و پرنده قادر به پرواز به میزان ده فوت است. برنامه بطور جامع یک پیغام (همان move) را به هر شی ارسال می‌کند، اما هر شی از نحوه اصلاح موقعیت خود براساس نوع حرکتی خود مطلع است و بر مبنای آن حرکت می‌کند. بر پایه اینکه هر شی از نحوه «انجام فعل صحیح» مطلع است، واکنش به فراخوانی تابع یکسان، مفهوم کلیدی چندریختی یا polymorphism است. پیغام یکسان (در این مورد move) که به انواع شی‌ها ارسال می‌شود، نتایج مختلفی بدنبال دارد و از اینرو نشاندهنده مفهوم چندریختی است.

به کمک چندریختی، می‌توانیم سیستم‌های را طراحی و پیاده‌سازی کنیم که گسترش و بسط‌پذیری آنها آسانتر است. کلاس‌های جدید می‌توانند با کمی تغییر یا اصلاح در بخش‌های عمومی برنامه، به آن افزوده شوند، مادامیکه کلاس‌های جدید بخشی از سلسله مراتب توارثی باشند که برنامه بطور جامع آنرا پردازش می‌کند. تنها بخش‌های از برنامه که باید برای تطبیق یافتن با کلاس‌های جدید تغییر داده شوند آنهایی هستند که نیاز دارند تا از وجود کلاس‌های جدید افزوده شده به سلسه مراتب مستقیماً مطلع گردند. برای مثال، اگر کلاس Tortoise (لاک‌پشت) را که از کلاس Animal ارث‌بری دارد را ایجاد کنیم (که می‌تواند به پیغام move به میزان یک اینچ حرکت یا خزیدن واکنش نشان دهد)، فقط نیاز است تا کلاس Tortoise و آن بخشی که یک نمونه از شی Tortoise را شبیه‌سازی می‌کند را بنویسیم.

با مطرح کردن مثال‌های سعی می‌کنیم تا درک مناسبی از مفهوم توابع **virtual** (مجازی) و *مقیدسازی دینامیکی* بوجود آوریم. که زیر ساخت‌های از تکنولوژی چند ریختی هستند. سپس به مطرح کردن یک مبحث آموزشی می‌پردازیم که در آن سلسله مراتب **Employee** از فصل دوازده بازبینی شده است. در مبحث آموزشی، یک «*واسط*» مشترک برای تمام کلاس‌های موجود در سلسله مراتب تعریف می‌کنیم. این واسط با قابلیت‌های مشترک در میان کارمندان، بعنوان کلاس مبنای انتزاعی **Employee** نامیده می‌شود، که از کلاس‌های **SalariedEmployee**، **HourlyEmployee** و **CommissionEmployee**

www.yourshop.4kia.ir

مستقیماً ارث‌بری دارد و کلاس **BasePlusCommissionEmployee** بصورت غیرمستقیم. بزودی شاهد این مطلب خواهید بود که چگونه کلاسی "انتزاعی" می‌شود و کلاسی "غیرانتزاعی".

در این سلسله مراتب، هر کارمندی دارای یک تابع **earnings** (حقوق) برای محاسبه حقوق هفتگی است. این توابع حقوق براساس نوع کارمند عمل می‌کنند، برای نمونه، کارمند **SalariedEmployee** یک حقوق هفتگی ثابت صرفنظر از ساعت کاری دریافت می‌کند، در حالیکه به کارمندی از نوع HourlyEmployee براساس ساعات کاری و اضافه کاری حقوق پرداخت می‌شود. نحوه پردازش هر کارمند را در حالت کلی نشان خواهیم داد که در آن از اشاره‌گرهای کلاس مبنا برای فراخوانی تابع **earnings** از میان چندین شی از کلاس مشتق شده استفاد می‌شود. در این روش، برنامه‌نویس نیاز به توجه به نوع فراخوانی تابع دارد، که می‌تواند برای اجرای چندین تابع مختلف براساس شی‌های اشاره شده توسط اشاره‌گرهای کلاس مبنا، بکار گرفته شود.

ویژگی کلیدی این فصل، بحث در ارتباط با چند ریختی، توابع **virtual** و مقیدسازی دینامیکی است، که در آن از یک دیاگرام برای توضیح اینکه چگونه چندریختی می‌تواند در C++ پیاده‌سازی شود، استفاده شده است.

از سلسله مراتب **Employee** برای شرح قابلیت‌های بنام *اطلاعات نوع زمان اجرا* (RTTI) و تبدیل دینامیکی استفاده مجدد خواهیم کرد که به برنامه امکان تعیین نوع یک شی در زمان اجرا را فراهم آورده و شی براساس آن عمل می‌کند.

۱۳-۲ چند مثال از چندریختی

در این بخش، در ارتباط با مثال‌های از چند ریختی بحث می‌کنیم. در چند ریختی، یک تابع می‌تواند اعمال متفاوتی را با توجه به نوع شی که تابع احضار می‌شود انجام دهد. اگر کلاس **Rectangle** (مستطیل) از کلاس **Quadrilateral** (چهارضلعی) است، پس یک شی مستطیل، نسخه بسیار خاصی از یک شی چهارضلعی است. بنابر این، هر عملیاتی (همانند محاسبه محیط یا مساحت) که می‌تواند بر روی یک شی از کلاس چهارضلعی بکار گرفته شود نیز می‌تواند بر روی شی از کلاس مستطیل پیاده گردد. البته چنین عملیاتی را می‌توان بر روی انواع چهارضلعی، همانند مربع‌ها، متوازی‌الاضلاع‌ها و ذوزنقه‌ها انجام داد. چند ریختی زمانی اتفاق می‌افتد که برنامه مبادرت به فراخوانی یک تابع **virtual** (مجازی) از طریق اشاره‌گر یا مراجعه کلاس مبنا (یعنی چهارضلعی) کند. C++ بصورت دینامیکی یا پویا (یعنی در زمان اجرا) تابع مناسب را برای کلاس انتخاب می‌کند (با توجه به شی که نمونه‌سازی شده است). در بخش ۳-۱۳ مثالی در این ارتباط به همراه کد آن آورده شده است.

www.yourshop.4kia.ir

بعنوان یک مثال دیگر، فرض کنید که یک بازی ویدئوی طراحی کرده‌ایم که شی‌های از نوع مختلف را نمونه‌سازی می‌کند، که در بر گیرنده شی‌های از کلاس‌های **Martian** (مریخی)، **Venutian** (ونوسی)، **Plutonian** (پلوتونی)، **SpaceShip** (سفینه فضایی) و **LaseBean** (پرتو لیزر) است. فرض کنید که هر کدامیک از این کلاس‌ها از کلاس مبنای مشترکی بنام **SpaceObject** ارث‌بری دارند، که حاوی تابع عضو **draw** است. هر کلاس مشتق شده، این تابع را به روش مقتضی برای آن کلاس پیاده‌سازی می‌کند. برنامه مدیریت صحنه مبادرت به نگهداری یک حامل (یک بردار یا **vector**) می‌کند که وظیفه آن حفظ اشاره‌گرهای **SpaceObject** به شی‌های از کلاس‌های مختلف است. برای نوسازی صحنه، مدیر صحنه در زمان‌های منظم به هر شی، پیغام یکسان **draw** را ارسال می‌کند. هر نوع از شی به یک طریق منحصر بفرد به این پیغام واکنش نشان می‌دهد. برای مثال، یک شی **Martian** می‌تواند خود را به رنگ قرمز با تعداد مشخصی آنتن ترسیم کند. یک شی **SpaceShip** می‌تواند خود را بصورت یک بشقاب پرنده نقره‌ای رنگ ترسیم نماید. یک شی **LaserBeam** می‌تواند خود را بصورت یک پرتو قرمز رنگ در امتداد صحنه ترسیم کند. مجدداً همان پیغام (در این مورد، **draw**) به انواع مختلفی از شی‌ها ارسال می‌شود و نتیجه آن بشکل‌های مختلف در می‌آید.

یک مدیر صحنه چند ریختی کار افزودن کلاس‌های جدید به یک سیستم را با حداقل تغییرات در کد فراهم می‌آورد. فرض کنید که می‌خواهیم شی‌های از کلاس **Mercurian** (عطارد) به بازی ویدئوی اضافه کنیم. برای انجام اینکار، بایستی یک کلاس **Mercurian** ایجاد کنیم که از **SpaceObject** ارث‌بری داشته باشد، اما تعریف متعلق بخود را از تابع عضو **draw** داشته باشد. سپس، زمانیکه اشاره‌گرهای به شی‌های از کلاس **Mercurian** در حامل ظاهر شوند، دیگر برنامه‌نویس نیازی به اصلاح کد مدیر صحنه نخواهد داشت. مدیر صحنه تابع عضو **draw** را برای هر شی در حامل فراخوانی می‌کند، صرفنظر از نوع شی، بنابر این شی‌های جدید **Mercurian** براحتی کار خود را انجام می‌دهند. از اینرو برنامه‌نویسان می‌توانند بدون هیچ تغییری در سیستم (بجز ایجاد و وارد کردن خود کلاس‌ها)، از چند ریختی برای تطبیق دادن کلاس‌های دیگر حتی آنهایی که در زمان ایجاد سیستم تصوری از آنها وجود نداشت، استفاده کنند.

۱۳-۳ رابطه مابین شی‌ها در سلسله مراتب توارث

در بخش ۴-۱۲ یک سلسله مراتب کلاس کارمند ایجاد کردیم که در آن کلاس **BasePlusCommissionEmployee** از کلاس **CommissinEmployee** ارث‌بری داشت. در فصل ۱۲ مثال‌های مطرح گردید که در آنها شی‌های **CommissinEmployee** و **BasePlusCommissionEmployee** با استفاده از اسامی شی‌ها مبادرت به احضار توابع عضو خود

www.yourshop.4kia.ir

می‌کردند. در این بخش به بررسی دقیق‌تر رابطه موجود در میان کلاس‌ها در سلسله مراتب می‌پردازیم. در چند بخش بعدی به معرفی دنباله‌ای از مثال‌ها خواهیم پرداخت که به بررسی عملکرد اشاره‌گرهای کلاس مبنا و کلاس مشتق شده می‌پردازند و همچنین نشان می‌دهند که چگونه می‌توان از این اشاره‌گرها در احضار توابع عضو استفاده کرد. در انتهای این بخش، به معرفی نحوه بدست گرفتن رفتار چند ریختی از اشاره‌گرهای کلاس مبنا که به شی‌های از کلاس مشتق شده اشاره دارند، خواهیم پرداخت.

۱-۳-۱۳ احضار توابع کلاس مبنا از طریق شی‌های کلاس مشتق شده

در مثال شکل‌های ۱-۱۳ الی ۵-۱۳ به بررسی سه روش در هدایت اشاره‌گرهای کلاس مبنا و اشاره‌گرهای کلاس مشتق شده بطرف شی‌های کلاس مبنا و شی‌های کلاس مشتق شده می‌پردازیم. دو روش اول بسیار سر راست هستند، یک اشاره‌گر کلاس مبنا را به طرف یک شی از کلاس مبنا هدایت کرده (و توابع کلاس مبنا احضار می‌شوند) و اشاره‌گر یک کلاس مشتق شده را به طرف یک شی کلاس مشتق شده (و توابع کلاس مشتق شده احضار می‌شوند) هدایت می‌کنیم. سپس، به بررسی رابطه موجود مابین کلاس‌های مشتق شده و کلاس‌های مبنا (یعنی رابطه is-a در سلسله مراتب) با هدایت یک اشاره‌گر کلاس مبنا بطرف یک شی کلاس مشتق شده می‌پردازیم (و نشان می‌دهیم که براستی قابلیت‌های کلاس مبنا در شی از کلاس مشتق شده وجود دارد).

از کلاس **CommissionEmployee** (شکل‌های ۱-۱۳ و ۲-۱۳) که در فصل ۱۲ توضیح دادیم استفاده می‌کنیم تا کارمندانی را معرفی کنیم که براساس درصدی از فروش به آنها حقوق پرداخت می‌شود. از کلاس **BasePlusCommissionEmployee** (شکل‌های ۳-۱۳ و ۴-۱۳) که در فصل ۱۲ توضیح داده‌ایم، به منظور معرفی کارمندانی که یک حقوق پایه به اضافه درصدی از فروش خود دریافت می‌کنند، استفاده می‌کنیم. هر شی **BasePlusCommissionEmployee** یک **CommissionEmployee** است (رابطه is-a) که دارای حقوق پایه نیز می‌باشد. کلاس **BasePlusCommissionEmployee** دارای تابع عضو **earnings** است (خطوط 35-32 از شکل ۴-۱۳) که تعریف مجددی از تابع عضو **earnings** از کلاس **CommissionEmployee** می‌باشد (خطوط 82-79 از شکل ۲-۱۳) که به آن حقوق پایه نیز افزوده گردیده است. تابع عضو **print** کلاس **BasePlusCommissionEmployee** (خطوط 46-38 از شکل ۴-۱۳) تعریف مجددی از تابع عضو **print** کلاس **CommissionEmployee** است (خطوط 92-85 از شکل ۲-۱۳) تا همان اطلاعات را به همراه حقوق پایه کارمند به نمایش در آورد.

```
1  // Fig. 13.1: CommissionEmployee.h
2  // CommissionEmployee class definition represents a commission employee.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
```

www.yourshop.4kia.ir

```
9  class CommissionEmployee
10 {
11 public:
12    CommissionEmployee( const string &, const string &, const string &,
13       double = 0.0, double = 0.0 );
14
15    void setFirstName( const string & ); // set first name
16    string getFirstName() const; // return first name
17
18    void setLastName( const string & ); // set last name
19    string getLastName() const; // return last name
20
21    void setSocialSecurityNumber( const string & ); // set SSN
22    string getSocialSecurityNumber() const; // return SSN
23
24    void setGrossSales( double ); // set gross sales amount
25    double getGrossSales() const; // return gross sales amount
26
27    void setCommissionRate( double ); // set commission rate
28    double getCommissionRate() const; // return commission rate
29
30    double earnings() const; // calculate earnings
31    void print() const; // print CommissionEmployee object
32 private:
33    string firstName;
34    string lastName;
35    string socialSecurityNumber;
36    double grossSales; // gross weekly sales
37    double commissionRate; // commission percentage
38 }; // end class CommissionEmployee
39
40 #endif
```

شکل ۱-۱۳ | فایل سرآیند کلاس CommissionEmployee.

```
1  // Fig. 13.2: CommissionEmployee.cpp
2  // Class CommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8  // constructor
9  CommissionEmployee::CommissionEmployee(
10    const string &first, const string &last, const string &ssn,
11    double sales, double rate )
12    : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
13 {
14    setGrossSales( sales ); // validate and store gross sales
15    setCommissionRate( rate ); // validate and store commission rate
16 } // end CommissionEmployee constructor
17
18 // set first name
19 void CommissionEmployee::setFirstName( const string &first )
20 {
21    firstName = first; // should validate
22 } // end function setFirstName
23
24 // return first name
25 string CommissionEmployee::getFirstName() const
26 {
27    return firstName;
28 } // end function getFirstName
29
30 // set last name
31 void CommissionEmployee::setLastName( const string &last )
32 {
33    lastName = last; // should validate
34 } // end function setLastName
35
36 // return last name
```

www.yourshop.4kia.ir

```
string CommissionEmployee::getLastName() const
{
   return lastName;
} // end function getLastName

// set social security number
void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
{
   socialSecurityNumber = ssn; // should validate
} // end function setSocialSecurityNumber

// return social security number
string CommissionEmployee::getSocialSecurityNumber() const
{
   return socialSecurityNumber;
} // end function getSocialSecurityNumber

// set gross sales amount
void CommissionEmployee::setGrossSales( double sales )
{
   grossSales = ( sales < 0.0 ) ? 0.0 : sales;
} // end function setGrossSales

// return gross sales amount
double CommissionEmployee::getGrossSales() const
{
   return grossSales;
} // end function getGrossSales

// set commission rate
void CommissionEmployee::setCommissionRate( double rate )
{
   commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
} // end function setCommissionRate

// return commission rate
double CommissionEmployee::getCommissionRate() const
{
   return commissionRate;
} // end function getCommissionRate

// calculate earnings
double CommissionEmployee::earnings() const
{
   return getCommissionRate() * getGrossSales();
} // end function earnings

// print CommissionEmployee object
void CommissionEmployee::print() const
{
   cout << "commission employee: "
      << getFirstName() << ' ' << getLastName()
      << "\nsocial security number: " << getSocialSecurityNumber()
      << "\ngross sales: " << getGrossSales()
      << "\ncommission rate: " << getCommissionRate();
} // end function print
```

شکل ۲-۱۳ | فایل پیاده‌سازی کلاس CommissionEmployee.

```
// Fig. 13.3: BasePlusCommissionEmployee.h
// BasePlusCommissionEmployee class derived from class
// CommissionEmployee.
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include <string> // C++ standard string class
using std::string;

#include "CommissionEmployee.h" // CommissionEmployee class declaration

class BasePlusCommissionEmployee : public CommissionEmployee
```

www.yourshop.4kia.ir

```
13 {
14 public:
15    BasePlusCommissionEmployee( const string &, const string &,
16       const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18    void setBaseSalary( double ); // set base salary
19    double getBaseSalary() const; // return base salary
20
21    double earnings() const; // calculate earnings
22    void print() const; // print BasePlusCommissionEmployee object
23 private:
24    double baseSalary; // base salary
25 }; // end class BasePlusCommissionEmployee
26
27 #endif
```

شکل ۳-۱۳ | فایل سرآیند کلاس BasePlusCommissionEmployee.

```
1  // Fig. 13.4: BasePlusCommissionEmployee.cpp
2  // Class BasePlusCommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  // BasePlusCommissionEmployee class definition
7  #include "BasePlusCommissionEmployee.h"
8
9  // constructor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11    const string &first, const string &last, const string &ssn,
12    double sales, double rate, double salary )
13    // explicitly call base-class constructor
14    : CommissionEmployee( first, last, ssn, sales, rate )
15 {
16    setBaseSalary( salary ); // validate and store base salary
17 } // end BasePlusCommissionEmployee constructor
18
19 // set base salary
20 void BasePlusCommissionEmployee::setBaseSalary( double salary )
21 {
22    baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
23 } // end function setBaseSalary
24
25 // return base salary
26 double BasePlusCommissionEmployee::getBaseSalary() const
27 {
28    return baseSalary;
29 } // end function getBaseSalary
30
31 // calculate earnings
32 double BasePlusCommissionEmployee::earnings() const
33 {
34    return getBaseSalary() + CommissionEmployee::earnings();
35 } // end function earnings
36
37 // print BasePlusCommissionEmployee object
38 void BasePlusCommissionEmployee::print() const
39 {
40    cout << "base-salaried ";
41
42    // invoke CommissionEmployee's print function
43    CommissionEmployee::print();
44
45    cout << "\nbase salary: " << getBaseSalary();
46 } // end function print
```

شکل ٤-۱۳ | فایل پیاده‌سازی کلاس BasePlusCommissionEmployee.

در شکل ۵-۱۳، خطوط 20-19 یک شی **CommissionEmployee** و در خط 23 یک اشاره‌گر به این شی و در خطوط 27-26 یک شی **BasePlusCommissionEmployee** و در خط 30 یک اشاره‌گر به

www.yourshop.4kia.ir

این شی ایجاد می‌شود. خطوط 37 و 39 از نام این شی‌ها برای احضار تابع عضو **print** هر یک از این شی‌ها استفاده می‌کنند. خط 42 آدرس کلاس مبنای شی **CommissionEmpolyee** را به اشاره‌گر کلاس مبنای **CommissionEmployeePtr** تخصیص می‌دهد، که خط 45 با استفاده از آن، تابع عضو **print** را برای شی **CommissionEmployee** احضار می‌نماید. با این عمل، نسخه **print** تعریف شده در کلاس مبنای **CommissionEmpolyee** احضار می‌شود. به همین ترتیب خط 48 آدرس شی کلاس مشتق شده **basePlusCommissionEmployee** را به اشاره‌گر کلاس مشتق شده **basePlusCommissionEmployeePtr** تخصیص می‌دهد، که خط 52 با استفاده از آن، تابع عضو print را بر روی شی **BasePlusCommissionEmployee** فراخوانی می‌کند. با این عمل، نسخه **print** تعریف شده در کلاس مشتق شده **BasePlusCommissionEmployee** فراخوانی می‌گردد. سپس خط **55** مبادرت به تخصیص آدرس شی کلاس مشتق شده **basePlusCommissionEmployee** به اشاره‌گر کلاس مبنا **CommissionEmployerPtr** می‌کند، که خط 59 با استفاده از آن تابع عضو **print** را احضار می‌نماید. کامپایلر C++ اجازه تغییر از یک حالت به حالت دیگر را می‌دهد، چرا که یک شی از یک کلاس مشتق شده، یک شی از کلاس مبنای خودش است (رابطه is-a). توجه کنید با وجود اینکه اشاره‌گر کلاس مبنا **CommissionEmployee** به یک کلاس مشتق شده **BasePlusCommissionEmployee** اشاره می‌کند، تابع عضو **print** کلاس مبنای **CommissionEmployee** احضار می‌شود (بجای تابع **print** کلاس **BasePlusCommissionEmployee**).

```
1  // Fig. 13.5: fig13_05.cpp
2  // Aiming base-class and derived-class pointers at base-class
3  // and derived-class objects, respectively.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7  using std::fixed;
8
9  #include <iomanip>
10 using std::setprecision;
11
12 // include class definitions
13 #include "CommissionEmployee.h"
14 #include "BasePlusCommissionEmployee.h"
15
16 int main()
17 {
18    // create base-class object
19    CommissionEmployee commissionEmployee(
20      "Sue", "Jones", "222-22-2222", 10000, .06 );
21
22    // create base-class pointer
23    CommissionEmployee *commissionEmployeePtr = 0;
24
25    // create derived-class object
26    BasePlusCommissionEmployee basePlusCommissionEmployee(
27       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
28
29    // create derived-class pointer
30    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
```

www.yourshop.4kia.ir

```
   // set floating-point output formatting
   cout << fixed << setprecision( 2 );

   // output objects commissionEmployee and basePlusCommissionEmployee
   cout << "Print base-class and derived-class objects:\n\n";
   commissionEmployee.print(); // invokes base-class print
   cout << "\n\n";
   basePlusCommissionEmployee.print(); // invokes derived-class print

   // aim base-class pointer at base-class object and print
   commissionEmployeePtr = &commissionEmployee; // perfectly natural
   cout << "\n\n\nCalling print with base-class pointer to "
      << "\nbase-class object invokes base-class print function:\n\n";
   commissionEmployeePtr->print(); // invokes base-class print

   // aim derived-class pointer at derived-class object and print
   basePlusCommissionEmployeePtr = &basePlusCommissionEmployee; // natural
   cout << "\n\n\nCalling print with derived-class pointer to "
      << "\nderived-class object invokes derived-class "
      << "print function:\n\n";
  basePlusCommissionEmployeePtr->print();// invokes derived-class print

   // aim base-class pointer at derived-class object and print
   commissionEmployeePtr = &basePlusCommissionEmployee;
   cout << "\n\n\nCalling print with base-class pointer to "
      << "derived-class object\ninvokes base-class print "
      << "function on that derived-class object:\n\n";
   commissionEmployeePtr->print(); // invokes base-class print
   cout << endl;
   return 0;
} // end main
```

```
Print base-class and derived-class objects:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary:300.00

calling print with base-class pointer to
base-class object invokes base-class print function:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

calling print with derived-class pointer to
derived-class object invokes derived-class print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary:300.00

calling print with base-class pointer to derived-class object
invokes base-class print function on that derived-class object:

commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
```

www.yourshop.4kia.ir

شکل ۵-۱۳ | تخصیص آدرس شی‌های کلاس مبنا و کلاس مشتق شده به اشاره‌گرهای کلاس مبنا و کلاس مشتق شده.

خروجی هر تابع عضو **print** احضار شده در این برنامه آشکار می‌کند که فراخوانی یا احضار تابع بستگی به نوع دستگیر یا هندلی دارد (یعنی نوع اشاره‌گر یا مراجعه) که در فراخوانی تابع بکار گرفته شده است، نه به نوع شی که هندل به آن اشاره می‌کند. در بخش ۴-۳-۱۳، زمانیکه به معرفی توابع **virtual** پرداختیم، نشان خواهیم داد که می‌توان بجای توجه به نوع هندل، مبادرت به فراخوانی تابع کرد. شاهد خواهید بود که اینحالت در پیاده‌سازی رفتار چند ریختی بسیار تعیین کننده است و یکی از مباحث کلیدی این فصل نیز می‌باشد.

۲-۳-۱۳ هدایت اشاره‌گرهای کلاس مشتق شده بطرف شی‌های کلاس مشتق شده

در بخش ۱-۳-۱۳، اقدام به تخصیص آدرس یک شی از کلاس مشتق شده به اشاره‌گر یک کلاس مبنا کردیم و توضیح دادیم که کامپایلر ++C اجازه انجام چنین تخصیصی را می‌دهد، چرا که یک شی از کلاس مشتق شده یک شی از کلاس مبنا است (رابطه is-a). در شکل ۶-۱۳ یک رویه مخالف اخذ کرده‌ایم و اشاره‌گر کلاس مشتق شده را به طرف یک شی کلاس مبنا هدایت می‌کنیم. [نکته: این برنامه از کلاس **CommissinEmployee** و **BasePlusCommissionEmployee** شکل‌های ۱-۱۳ الی ۴-۱۳ استفاده کرده است.] خطوط 9-8 از شکل ۶-۱۳ یک شی **CommissionEmployee** ایجاد می‌کنند و خط 10 یک اشاره‌گر **BasePlusCommissionEmployee** ایجاد می‌نماید خط 14 مبادرت به تخصیص آدرس شی کلاس مبنا **CommissionEmployee** به اشاره‌گر کلاس مشتق شده **basePlusCommissionEmployeePtr** می‌کند، اما کامپایلر ++C خطا تولید می‌کند. کامپایلر مانع انجام چنین تخصیصی می‌شود، چرا که یک **CommissionEmployee** یک **BasePlusCommissionEmployee** نیست. اگر کامپایلر اجازه چنین تخصیصی را می‌داد، اتفاقات زیر بدنبال هم رخ می‌دادند. از طریق اشاره‌گر **BasePlusCommissionEmployee** می‌توانستیم هر تابع **BasePlusCommissionEmployee** شامل **setBaseSalary** را برای شی که اشاره‌گر بر آن اشاره دارد فراخوانی کنیم (یعنی شی کلاس مبنا **CommissionEmployee**). با این وجود، شی **CommissionEmployee** دارای تابع عضو **setBaseSalary** نیست و نمی‌تواند عضو داده **baseSalary** را تنظیم کند (چرا که این نوع کارمند دارای حقوق پایه نیست و دستمزد خود را براساس درصدی از میزان فروش دریافت می‌کند). پس اینکار می‌تواند منجر به مشکلاتی گردد، برای اینکه تابع عضو **setBaseSalary** فرض می‌کند که در اینجا یک عضو داده **baseSalary** برای تنظیم وجود دارد. این حافظه متعلق به شی **CommissinEmployee** نبوده و از اینرو تابع عضو **setBaseSalary** می‌تواند بر روی

www.yourshop.4kia.ir

اطلاعات با ارزش دیگری را که در حافظه قرار دارند، بازنویسی کند، در صورتیکه این اطلاعات متعلق به شی دیگری هستند.

```
1  // Fig. 13.6: fig13_06.cpp
2  // Aiming a derived-class pointer at a base-class object.
3  #include "CommissionEmployee.h"
4  #include "BasePlusCommissionEmployee.h"
5
6  int main()
7  {
8     CommissionEmployee commissionEmployee(
9        "Sue", "Jones", "222-22-2222", 10000, .06 );
10    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
11
12    // aim derived-class pointer at base-class object
13    // Error: a CommissionEmployee is not a BasePlusCommissionEmployee
14    basePlusCommissionEmployeePtr = &commissionEmployee;
15    return 0;
16 } // end main
```

Borland C++ command-line compiler error message:

```
Error E2034 Fig13_06\fig13_06.cpp 14: Cannot convert 'CommissionEmployee
*'
   to 'BasePluseCommissionEmployee *' in function main()
```

GNU C++ compiler error message:

```
fig13_06.cpp:14: error: invalid conversion from 'CommissionEmployee*' to
   'BasePlusCommisssionEmployee*'
```

Microsoft Visual C++.NET compiler error message:

```
C:\cpphtp5_examples\ch13\Fig13_06\fig13_06.cpp(14): error C2440:
   '*': cannot convert from 'CommissionEmployee *__w64 ' to
   'BasePlusCommissionEmployee *'
        Cast from base to derived requires dynamic_cast or static_cast
```

شکل ٦-١٣ | هدایت اشاره‌گر کلاس مشتق شده بطرف یک شی کلاس مبنا.

۳-۳-۱۳ فراخوانی تابع عضو کلاس مشتق شده از طریق اشاره‌گرهای کلاس مبنا

بدون حضور اشاره‌گر کلاس مبنا، کامپایلر اجازه می‌دهد تا فقط توابع عضو کلاس مبنا را فراخوانی کنیم. از اینرو اگر اشاره‌گر کلاس مبنا بطرف یک شی کلاس مشتق شده هدایت گردد، و مبادرت به دسترسی به یک عضو از کلاس مشتق شده شود، خطای زمان کامپایل رخ خواهد داد.

برنامه شکل ۷-۱۳ پیآمد مبادرت به احضار یک تابع عضو کلاس مشتق شده از طریق اشاره‌گر کلاس مبنا را نشان می‌دهد. [نکته: مجدداً از کلاس‌های **CommissionEmployee** و **BasePlusCommissionEmployee** شکل‌های ۱-۱۳ الی ۴-۱۳ استفاده کرده‌ایم]. خط 9 مبادرت به ایجاد **CommissionEployeePtr** می‌کند، که یک اشاره‌گر به یک شی **CommissionEmployee** است، و خطوط 11-10 یک شی **BasePlusCommissionEmployee** ایجاد می‌کند. خط 14 اشاره‌گر را بطرف شی مشتق شده از کلاس **BasePlusCommissionEmployee** هدایت می‌نماید. از بخش ۱-۳-۱۳ بخاطر دارید که کامپایلر C++ اجازه انجام چنین کاری را می‌دهد، چرا که یک **BasePlusCommissionEmployee** یک **CommissionEmployee** است.

```
1  // Fig. 13.7: fig13_07.cpp
2  // Attempting to invoke derived-class-only member functions
3  // through a base-class pointer.
4  #include "CommissionEmployee.h"
5  #include "BasePlusCommissionEmployee.h"
```

www.yourshop.4kia.ir

```
6
7  int main()
8  {
9     CommissionEmployee *commissionEmployeePtr = 0; // base class
10    BasePlusCommissionEmployee basePlusCommissionEmployee(
11       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 ); // derived class
12
13    // aim base-class pointer at derived-class object
14    commissionEmployeePtr = &basePlusCommissionEmployee;
15
16    // invoke base-class member functions on derived-class
17    // object through base-class pointer
18    string firstName = commissionEmployeePtr->getFirstName();
19    string lastName = commissionEmployeePtr->getLastName();
20    string ssn = commissionEmployeePtr->getSocialSecurityNumber();
21    double grossSales = commissionEmployeePtr->getGrossSales();
22    double commissionRate = commissionEmployeePtr->getCommissionRate();
23
24    // attempt to invoke derived-class-only member functions
25    // on derived-class object through base-class pointer
26    double baseSalary = commissionEmployeePtr->getBaseSalary();
27    commissionEmployeePtr->setBaseSalary( 500 );
28    return 0;
29 } // end main
```

شکل ۷-۱۳ | اقدام به احضار فقط توابع کلاس مشتق شده از طریق اشاره‌گر کلاس مبنا.

خطوط 22-18 توابع عضو کلاس مبنا بنام‌های **getFirstName**، **getLastName**، **getSocialSecurityName**، **getGrossSales** و **getCommissionRate** از طریق اشاره‌گر کلاس مبنا فراخوانی می‌کنند. تمام این فراخوانی‌ها مشروع هستند، برای اینکه **BasePlusCommissionEmployee** این توابع عضو را از **CommissionEmployee** به ارث برده است. می‌دانیم که اشاره‌گر **CommissionEmployeePtr** بطرف یک شی **BasePlusCommissionEmployee** هدایت شده، از اینرو در خطوط 27-26 مبادرت به احضار توابع عضو **getBaseSalary** و **setBaseSalary** از **BasePlusCommissionEmployee** کرده‌ایم. کامپایلر C++ بر روی هر دو این خطوط خطا تولید می‌کند، به این دلیل که اینها توابع عضو کلاس **CommissionEmployee** نیستند.

۱۳-۳-٤ توابع virtual

در بخشی ۱-۳-۱۳ مبادرت به هدایت اشاره‌گر کلاس مبنا **CommissionEmployee** به طرف شی **BasePlusCommissionEmployee** از یک کلاس مشتق شده کردیم، سپس تابع عضو **print** را از طریق این اشاره‌گر فراخوانی نمودیم. بخاطر دارید که نوع دستگیره تعیین می‌کند که کدام تابع کلاس احضار شود. در این مورد، اشاره‌گر **CommissionEmployee** تابع عضو **print** متعلق به **CommissionEmployee** را بر روی شی **BasePlusCommissionEmployee** فراخوانی می‌کند، ولو اینکه اشاره‌گر به طرف شی **BasePlusCommissionEmployee** هدایت شده باشد که خود دارای تابع **print** خاص خودش است. به کمک توابع **virtual** (مجازی)، نوع شی اشاره شده و نه نوع دستگیره، تعیین می‌کند که کدام نسخه از یک تابع مجازی فراخوانی شود.

www.yourshop.4kia.ir

ابتدا به دلیل سودمند بودن توابع مجازی می‌پردازیم. فرض کنید که مجموعه‌ای از کلاس‌های شکل همانند **Circle** (دایره)، **Triangle** (مثلث)، **Rectangle** (مستطیل) و **Square** (مربع) داریم که همگی از کلاس مبنا **Shape** مشتق شده‌اند. هر کدامیک از کلاس‌ها می‌توانند از قابلیت ترسیم خود از طریق یک تابع عضو بنام **draw** برخوردار باشد. اگرچه هر کلاسی دارای تابع **draw** متعلق بخود است، عملکرد این تابع برای هر شکل با دیگری کاملاً متفاوت خواهد بود. در برنامه‌ای که مجموعه‌ای از شکل‌ها را ترسیم می‌کند، قابلیت تلقی کردن تمام شکل‌ها بصورت شی‌های از کلاس مبنا **Shape** سودمند خواهد بود. سپس، برای ترسیم هر شکلی، می‌توانیم به آسانی از یک اشاره‌گر **Shape** کلاس مبنا برای فراخوانی تابع **draw** استفاده کرده و به برنامه اجازه دهیم تا بصورت دینامیکی (یعنی در زمان اجرا) تعیین کند که کدام تابع **draw** کلاس مشتق شده برحسب نوع شی که اشاره‌گر **Shape** به آن اشاره می‌کند، بکار گرفته شود.

برای داشتن چنین رفتاری، ابتدا تابع **draw** را در کلاس مبنا بعنوان یک تابع **virtual** اعلان کرده و تابع **draw** در هر کلاس مشتق شده را برای ترسیم شکل مقتضی **override** می‌کنیم. از منظر پیاده‌سازی، **override** کردن یک تابع تفاوتی با تعریف مجدد آن ندارد (روشی که تا بدین جا از آن استفاده کرده‌ایم). یک تابع **override** شده در یک کلاس مشتق شده دارای همان امضاء و نوع برگشتی است (یعنی *نوع اولیه* یا *prototype*). اگر تابع کلاس مبنا را بصورت **virtual** اعلان نکنیم، می‌توانیم آن تابع را مجدداً تعریف کنیم. در مقابل اگر تابع کلاس مبنا را بصورت **virtual** اعلان کنیم، می‌توانیم آن تابع را **override** کرده تا از رفتار چند ریختی بهره‌مند گردیم.

می‌توانیم نمونه اولیه تابع فوق را با قرار دادن کلمه کلیدی **virtual** در کلاس مبنا، بصورت زیر بعنوان یک تابع **virtual** اعلان کنیم. برای مثال **virtual void draw() const** می‌تواند در کلاس مبنای **Shape** جای داده شود. در عبارت فوق تابع **draw** بصورت یک تابع **virtual** اعلان شده که هیچ آرگومانی دریافت نمی‌کند و چیزی هم برگشت نمی‌دهد. تابع بصورت **const** اعلان شده است چرا که تابع **draw** تغییری در شی **Shap** که برای ترسیم آن فراخوانی شده، بوجود نمی‌آورد. ضرورتاً توابع **virtual** مجبور نیستند تا بصورت **const** اعلان شوند.

اگر برنامه‌ای مبادرت به فراخوانی یک تابع **virtual** از طریق اشاره‌گر یک کلاس مبنا به یک شی از کلاس مشتق شده کند (مثلاً **ShapePtr->draw()**)، برنامه بصورت دینامیکی (یعنی در زمان اجرا) تابع صحیح **draw** را براساس نوع شی و نه نوع اشاره‌گر انتخاب خواهد کرد. انتخاب تابع مقتضی برای فراخوانی در زمان اجرا (بجای زمان کامپایل) بعنوان *مقیدسازی دینامیکی* (*dynamic binding*) یا *مقیدسازی تاخیری* (*late binding*) شناخته می‌شود.

www.yourshop.4kia.ir

زمانیکه یک تابع **virtual** توسط مراجعه‌ای به یک شی خاص توسط نام و استفاده از عملگر انتخاب عضو، نقطه (مثلاً **squarObject.draw()**) فراخوانی می‌شود، احضار تابع در زمان کامپایل مقرر می‌شود (که به اینحالت مقیدسازی استاتیک گفته می‌شود) و تابع **virtual** که فراخوانی شده یک تابع تعریف شده برای کلاسی از شی مشخص است، که این رفتار نشاندهنده چند ریختی نیست. از اینرو، مقیدسازی دینامیکی با توابع **virtual** فقط با دستگیره‌های اشاره‌گر (مراجعه) اتفاق می‌افتد.

حال اجازه دهید تا ببینیم چگونه توابع **virtual** می‌تواند نشاندهنده رفتار چند ریختی در سلسله مراتب کارمندی باشند. شکل‌های ۸-۱۳ و ۹-۱۳ فایلهای سرآیند برای کلاس‌های **CommissionEmployee** و **BasePlusCommissionEmployee** هستند. توجه کنید که تنها تفاوت موجود مابین این فایلها و آنهایی که در شکل‌های ۱-۱۳ و ۳-۱۳ قرار دارند در این است که توابع عضو **earnings** و **print** را بصورت **virtual** اعلان کرده‌ایم (خطوط 31-30 از شکل ۸-۱۳ و خطوط 22-21 از شکل ۹-۱۳). چون توابع **earnings** و **print** بصورت **virtual** در کلاس **CommissionEmployee** هستند، توابع **earnings** و **print** در **BasePlusCommissionEmployee** اقدام به **override** کردن کلاس **CommissionEmployee** می‌کنند. اکنون، اگر مبادرت به هدایت اشاره‌گر کلاس مبنای **CommissionEmployee** بطرف یک شی از کلاس مشتق شده **BasePlusCommissionEmployee** کنیم و برنامه از آن اشاره‌گر برای فراخوانی هر یک از دو تابع **earnings** یا **print** استفاده کند، تابع متناظر شی **BasePlusCommissionEmployee** فراخوانی خواهد شد. در اینجا هیچ تغییری در پیاده‌سازی تابع عضو از کلاس‌های **CommidssionEmployee** و **BasePlusCommissionEmployee** رخ نمی‌دهد، از اینرو استفاده مجددی از نسخه‌های شکل ۲-۱۳ و ۴-۱۳ می‌کنیم.

برای ایجاد برنامه شکل ۱۰-۱۳، تغییراتی در برنامه شکل ۵-۱۳ اعمال کرده‌ایم. خطوط 57-46 مجدداً نشان می‌دهند که می‌توان یک اشاره‌گر **CommissionEmployee** را بطرف یک شی **CommissionEmployee** هدایت کرده و توابع آنرا احضار کرد. اینحالت برای اشاره‌گر **BasePlusCommissionEmployee** نیز صادق است. در خط 60، اشاره‌گر کلاس مبنا **CommissionEmplyeePtr** بطرف شی کلاس مشتق شده **BasePlusCommissionEmployee** هدایت شده است. دقت کنید زمانیکه خط 67 مبادرت به احضار تابع عضو **print** از طریق اشاره‌گر کلاس مبنا می‌کند، تابع عضو **print** کلاس مشتق شده **BasePlusCommissionEmployee** احضار می‌گردد، از اینرو خط 67 متن متفاوتی از خط 59 در شکل ۵-۱۳ را در خروجی قرار می‌دهد (زمانیکه تابع عضو **print** بصورت **virtual** اعلان نشده بود).

```
1  // Fig. 13.8: CommissionEmployee.h
2  // CommissionEmployee class definition represents a commission employee.
3  #ifndef COMMISSION_H
```

www.yourshop.4kia.ir

```
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
9  class CommissionEmployee
10 {
11 public:
12    CommissionEmployee( const string &, const string &, const string &,
13       double = 0.0, double = 0.0 );
14
15    void setFirstName( const string & ); // set first name
16    string getFirstName() const; // return first name
17
18    void setLastName( const string & ); // set last name
19    string getLastName() const; // return last name
20
21    void setSocialSecurityNumber( const string & ); // set SSN
22    string getSocialSecurityNumber() const; // return SSN
23
24    void setGrossSales( double ); // set gross sales amount
25    double getGrossSales() const; // return gross sales amount
26
27    void setCommissionRate( double ); // set commission rate
28    double getCommissionRate() const; // return commission rate
29
30    virtual double earnings() const; // calculate earnings
31    virtual void print() const; // print CommissionEmployee object
32 private:
33    string firstName;
34    string lastName;
35    string socialSecurityNumber;
36    double grossSales; // gross weekly sales
37    double commissionRate; // commission percentage
38 }; // end class CommissionEmployee
39
40 #endif
```

شکل ۸-۱۳ | فایل سرآیند کلاس CommissionEmployee که در آن توابع earnings و print بعنوان virtual اعلان شده‌اند.

```
1  // Fig. 13.9: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from class
3  // CommissionEmployee.
4  #ifndef BASEPLUS_H
5  #define BASEPLUS_H
6
7  #include <string> // C++ standard string class
8  using std::string;
9
10 #include "CommissionEmployee.h" // CommissionEmployee class declaration
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15    BasePlusCommissionEmployee( const string &, const string &,
16       const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18    void setBaseSalary( double ); // set base salary
19    double getBaseSalary() const; // return base salary
20
21    virtual double earnings() const; // calculate earnings
22    virtual void print() const; // print BasePlusCommissionEmployee object
23 private:
24    double baseSalary; // base salary
25 }; // end class BasePlusCommissionEmployee
26
27 #endif
```

www.yourshop.4kia.ir

شکل ۹-۱۳ | فایل سرآیند کلاس BasePlusCommissionEmployee که در آن توابع earnings و print بعنوان virtual اعلان شده‌اند.

مشاهده کردید که اعلان یک تابع عضو **virtual** سبب می‌شود تا برنامه بصورت دینامیکی تعیین کند که کدام تابع براساس نوع شی که دستگیره به آن اشاره می‌کند، فراخوانی گردد (بجای توجه به نوع دستگیره). تصمیم در مورد اینکه کدام تابع فراخوانی شود، مثالی از چند ریختی است.

مجدداً توجه کنید زمانیکه **CommissionEmployeePtr** به یک شی **CommissionEmployee** اشاره می‌کند (خط 46) تابع **print** کلاس **CommissionEmployee** احضار شده و زمانیکه **CommissionEmployePtr** به یک شی **BasePlusCommissionEmployee** اشاره می‌کند، تابع **print** کلاس **BasePlusCommissionEmployee** احضار می‌گردد. از اینرو، پیغام یکسان – در این مورد، **print** – به انواع مختلفی از شی‌ها ارسال می‌شود که رابطه ارث‌بری با کلاس مبنا دارند، و وارد فرم‌های متعدد نشده که نشاندهنده رفتار چند ریختی است.

```
1  // Fig. 13.10: fig13_10.cpp
2  // Introducing polymorphism, virtual functions and dynamic binding.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::fixed;
7
8  #include <iomanip>
9  using std::setprecision;
10
11 // include class definitions
12 #include "CommissionEmployee.h"
13 #include "BasePlusCommissionEmployee.h"
14
15 int main()
16 {
17    // create base-class object
18    CommissionEmployee commissionEmployee(
19       "Sue", "Jones", "222-22-2222", 10000, .06 );
20
21    // create base-class pointer
22    CommissionEmployee *commissionEmployeePtr = 0;
23
24    // create derived-class object
25    BasePlusCommissionEmployee basePlusCommissionEmployee(
26       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
27
28    // create derived-class pointer
29    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
30
31    // set floating-point output formatting
32    cout << fixed << setprecision( 2 );
33
34    // output objects using static binding
35    cout << "Invoking print function on base-class and derived-class "
36       << "\nobjects with static binding\n\n";
37    commissionEmployee.print(); // static binding
38    cout << "\n\n";
39    basePlusCommissionEmployee.print(); // static binding
40
41    // output objects using dynamic binding
42    cout << "\n\n\nInvoking print function on base-class and "
43       << "derived-class \nobjects with dynamic binding";
44
```

www.yourshop.4kia.ir

```
   // aim base-class pointer at base-class object and print
   commissionEmployeePtr = &commissionEmployee;
   cout << "\n\nCalling virtual function print with base-class pointer"
      << "\nto base-class object invokes base-class "
      << "print function:\n\n";
   commissionEmployeePtr->print(); // invokes base-class print

   // aim derived-class pointer at derived-class object and print
   basePlusCommissionEmployeePtr = &basePlusCommissionEmployee;
   cout << "\n\nCalling virtual function print with derived-class "
      << "pointer\nto derived-class object invokes derived-class "
      << "print function:\n\n";
   basePlusCommissionEmployeePtr->print(); // invokes derived-class print

   // aim base-class pointer at derived-class object and print
   commissionEmployeePtr = &basePlusCommissionEmployee;
   cout << "\n\nCalling virtual function print with base-class pointer"
      << "\nto derived-class object invokes derived-class "
      << "print function:\n\n";

   // polymorphism; invokes BasePlusCommissionEmployee's print;
   // base-class pointer to derived-class object
   commissionEmployeePtr->print();
   cout << endl;
   return 0;
} // end main
```

```
Invoking print function on base-class and derived-class
Objects with static binding

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary:300.00

Invoking print function on base-class and derived-class
Objects with dynamic binding

calling virtual function print with base-class pointer to
base-class object invokes base-class print function:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

calling virtual function print with derived-class pointer to
derived-class object invokes derived-class print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary:300.00

calling virtual function print with base-class pointer to derived-class
object invokes derived-class print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

www.yourshop.4kia.ir

شکل ۱۰-۱۳ | توصیف چند ریختی با فراخوانی یک تابع virtual کلاس مشتق شده از طریق اشاره‌گر کلاس مبنا به یک کلاس مشتق شده.

۵-۳-۱۳ تخصیص‌های قابل انجام مابین شی‌ها و اشاره‌گرهای کلاس مبنا و کلاس مشتق شده

اکنون که شاهد یک برنامه کامل که مبادرت به پردازش شی‌های مشتق شده به روش چند ریختی بودید، بطور خلاصه مواردی را که می‌توانید با شی‌های کلاس مشتق شده، مبنا و اشاره‌گر انجام دهید و آنهایی که نمی‌توانید انجام دهید، مطرح می‌کنیم.

اگرچه یک شی کلاس مشتق شده یک شی از یک کلاس مبنا است، اما این دو شی با هم تفاوت‌های دارند. همانطوری که قبلاً هم بحث شدهٔ، شی‌های کلاس مشتق شده در صورتیکه شی‌های از کلاس مبنا باشند می‌توانند بصورت کلاس مبنا در نظر گرفته شوند. این یک رابطه منطقی است، چرا که کلاس مشتق شده حاوی تمام اعضا از کلاس مبنا است. با این وجود نمی‌توان با شی‌های کلاس مبنا مثل اینکه شی‌های از کلاس مشتق شده هستند رفتار کرد. به همین دلیل، هدایت اشاره‌گر کلاس مشتق شده بطرف یک شی کلاس مبنا، بدون یک تبدیل صریح امکان‌پذیر نمی‌باشد. عمل تبدیل به کامپایلر کمک می‌کند تا از صدور پیغام خطا اجتناب کند. در چنین حالتی، با استفاده از تبدیل می‌گوید که «من از خطری کاری که انجام می‌دهم مطلع هستم و مسئولیت تمام کارها را برعهده می‌گیرم.»

در بخش جاری و فصل دوازدهم، به بررسی چهار روش در هدایت اشاره‌گرهای کلاس مبنا و اشاره‌گرهای کلاس مشتق شده بطرف شی‌های کلاس مبنا و کلاس مشتق شده پرداختیم:

۱- هدایت اشاره‌گر کلاس مبنا بطرف شی از کلاس مبنا کار سرراستی است. با این عمل اشاره‌گر مبادرت به فراخوانی کلاس مبنا می‌کند.

۲- هدایت اشاره‌گر کلاس مشتق شده بطرف شی از کلاس مشتق شده کار سرراستی است. با این عمل اشاره‌گر مبادرت به فراخوانی کلاس مشتق شده می‌کند.

۳- هدایت اشاره‌گر یک کلاس مبنا بطرف یک شی از کلاس مشتق شده، خطری ندارد، چرا که شی از کلاس مشتق شده یک شی از کلاس مبنای خودش است. با این وجود، از این اشاره‌گر می‌توان فقط در احضار توابع عضو کلاس مبنا استفاده کرد. اگر برنامه‌نویس از طریق اشاره‌گر کلاس مبنا مبادرت به اشاره به یک عضو فقط کلاس مشتق شده کند، کامپایلر خطا گزارش خواهد کرد.

برای اجتناب از این خطا، برنامه‌نویس بایستی اشاره‌گر کلاس مبنا را تبدیل به اشاره‌گر کلاس مشتق شده نماید. سپس می‌توان از اشاره‌گر کلاس مشتق شده برای فراخوانی کل توابع شی از کلاس مشتق شده استفاده کرد. با این همه این روش خطراتی نیز دارد که بخش ۸-۱۳ به بررسی و رفع آن خواهیم پرداخت.

www.yourshop.4kia.ir

۴- هدایت اشاره‌گر کلاس مشتق شده بطرف شی از کلاس مبنا، خطای کامپایل تولید می‌کند. رابطه is-a (است–یک) فقط بر روی یک کلاس مشتق شده بصورت مستقیم یا غیرمستقیم از کلاس مبنا خود معتبر است و عکس آن صادق نیست. یک شی از کلاس مبنا حاوی عضوهای خاص کلاس مشتق شده نیست که بتواند اشاره‌گر کلاس مشتق شده را احضار نماید.

۱۳-٤ عبارت switch

یکی از روش‌های تعیین نوع یک شی در یک برنامه بزرگ استفاده از عبارت **switch** است. این عبارت امکان می‌دهد تا مابین انواع شی‌ها تفاوت قائل شده، سپس عمل مقتضی را بر روی آن شی مشخص انجام دهیم. برای مثال در سلسله مراتب شکل‌ها که در آن هر شی دارای صفت **shaptType** است، یک عبارت **switch** می‌تواند به بررسی **shapeType** شی پرداخته و تعیین کند که کدام تابع **print** فراخوانی گردد.

با این همه، استفاده از **switch** سبب می‌شود تا منطق برنامه در معرض دید قرار گرفته و مهیا برای مشکلات شود. برای مثال، امکان دارد برنامه‌نویس انجام تستی بر روی نوع خاصی را یا قرار دادن تمام حالات ممکنه را در عبارت **switch** را فراموش کند. به هنگام اصلاح یک سیستم مبتنی بر **switch** که با افزودن نوع‌های جدید همراه است، امکان دارد برنامه‌نویس وارد کردن حالات جدید را در تمام عبارات **switch** وابسته فراموش نماید. هر افزودن یا حذف کلاسی مستلزم اصلاح در کلیه عبارات **switch** است، بررسی چنین عباراتی می‌تواند زمانبر بوده و مستعد و زمینه‌ساز خطا است.

۱۳-۵ کلاس‌های انتزاعی و توابع virtual محض

هنگامی که در مورد نوع یک کلاس فکر می‌کنیم، فرض ما بر این است که برنامه‌ها اقدام به ایجاد شیی‌ها از نوع تعیین شده خواهند کرد. با این همه، گاهی اوقات فقط کلاس‌ها تعریف می‌شوند و برنامه‌نویسان هرگز قصد ندارند هیچ شیی را نمونه‌سازی کنند. چنین کلاس‌هایی، کلاس‌های *انتزاعی* نامیده می‌شوند. چرا که چنین کلاس‌هایی معمولاً بعنوان کلاس‌های مبنا در سلسله مراتب توارث بکار گرفته می‌شوند، از اینرو معمولاً با عنوان کلاس‌های *مبنای انتزاعی* شناخته می‌شوند. این کلاس‌ها نمی‌توانند برای نمونه‌سازی شیی‌ها بکار گرفته شوند. کلاس‌های انتزاعی کامل نیستند. کلاس‌های مشتق شده بایستی بعنوان بخش‌های مفقود شده تعریف شوند.

منظور از یک کلاس انتزاعی فراهم آوردن یک کلاس مبنای مقتضی از سایر کلاس‌ها است که ممکن است به ارث برسند. کلاس‌های که از چنین شی‌هایی نمونه‌سازی می‌شوند، کلاس‌های *مقید* نام دارند. چنین کلاس‌هایی تدارک بیننده هر تابع عضو هستند که تعریف شده‌اند. برای مثال می‌توانیم یک کلاس مبنای انتزاعی بنام **TwoDimensionalObject** و کلاس‌های مشتق شده مقید همانند **Squar**،

www.yourshop.4kia.ir

Circle و **Triangle** داشته باشیم. همچنین می‌توانیم یک کلاس مبنای انتزاعی بنام **ThreeDimensionalObject** و کلاس‌های مشتق شده مقید همانند **Cube**، **Sphere** و **Cylinder** داشته باشیم. کلاس‌های مبنای انتزاعی برای تعریف شیی‌های واقعی بسیار کلی هستند، از اینرو قبل از اینکه شیی را نمونه‌سازی کنیم باید با دقت در مورد آن فکر کنیم. برای مثال، اگر شخصی به شما بگوید "شکلی دو بعدی ترسیم کنید"، باید پرسید چه شکلی؟

یک سلسله مراتب توارث نیازی به کلاس‌های انتزاعی ندارد، اما همانطوری که مشاهده خواهید کرد، بسیاری از سیستم‌های خوب شی‌گرا، دارای سلسله مراتب کلاسی مناسب با کلاس‌های مبنا انتزاعی هستند. در برخی از موارد کلاس‌های انتزاعی، چند سطح فوقانی را در سلسله مراتب تشکیل می‌دهند. یک مثال خوب در این زمینه، سلسله مراتب شکل‌ها در شکل ۳-۱۲ است که با کلاس مبنای انتزاعی **Shape** شروع می‌شود. در سطح بعدی سلسله مراتب دو کلاس مبنای انتزاعی دیگر بنام‌های **TwoDimensionalShape** (شکل‌های دوبعدی) و **ThreeDimensinalShape** (شکل‌های سه بعدی) داریم. سطح بعدی سلسله مراتب کلاس‌های غیرانتزاعی برای شکل‌های دوبعدی را تعریف کرده است (بنام‌های **Circle**، **Square** و **Triangle**) و برای شکل‌های سه بعدی بنام‌های **Sphere**، **Cube** و **Tetrahedron** (چهارسطحی).

با اعلان یک یا چند تابع **virtual** یک کلاس بصورت محض، آن کلاس بصورت یک کلاس انتزاعی ایجاد می‌شود. با قرار دادن "**0** =" در اعلان یک تابع **virtual** آن تابع بصورت یک تابع **virtual** محض مشخص می شود، بصورت

```
virtual void draw() const = 0; //pure virtual function
```

"**0=**" بعنوان تصریح کننده محض شناخته می‌شود. توابع **virtual** محض دارای پیاده‌سازی نیستند. هر کلاس غیرانتزاعی بایستی تمام توابع **virtual** محض کلاس مبنا را **override** کند با پیاده‌سازی غیرانتزاعی توابع آنها. تفاوت موجود مابین یک تابع **virtual** و یک تابع **virutal** محض در این است که تابع **virtual** دارای پیاده‌سازی بوده و به کلاس مشتق شده گزینه‌ای برای **override** (لغو کردن) تابع اعطا می‌کند، در مقابل، یک تابع **virtual** محض دارای پیاده‌سازی نبوده و کلاس مشتق شده را ملزم به **oveeride** کردن تابع می‌نماید (به همین دلیل است که کلاس مشتق شده غیرانتزاعی می‌شود، در غیر اینصورت کلاس مشتق شده، انتزاعی باقی می‌ماند).

از توابع **virtual** محض زمانی استفاده می‌شود که احساس شود کلاس مبنا نیازی به پیاده‌سازی یک تابع ندارد، اما برنامه‌نویس مایل است تمام کلاس‌های مشتق شده غیرانتزاعی را در پیاده‌سازی تابع داشته باشد. اگر به مثال فضایی خود در ابتدای فصل باز گردیم، متوجه می‌شوید که کلاس مبنا **SpaceObject** دارای

www.yourshop.4kia.ir

پیاده‌سازی برای تابع **draw** نبود. مثالی از یک تابع که می‌تواند بعنوان یک تابع **virtual** (و نه یک **virtual** محض) تعریف شود آن است که نامی برای شی برگشت دهد.

اگرچه نمی‌توانیم نمونه‌های از شی‌های یک کلاس مبنا انتزاعی ایجاد کنیم، اما می‌توانیم از کلاس مبنای انتزاعی بمنظور اعلان اشاره‌گرها و مراجعه‌های که می‌توانند به شی‌های از هر کلاس غیرانتزاعی مشتق شده از کلاس‌های انتزاعی مراجعه کند، ایجاد کنیم. معمولاً برنامه‌ها از چنین اشاره‌گر و مراجعه‌های برای کار با شی‌های کلاس مشتق شده به روش چندریختی استفاده می‌کنند.

چند ریختی نقش ویژه‌ای در پیاده‌سازی لایه‌های مختلف در سیستم‌های نرم‌افزاری دارد. برای مثال، در سیستم‌های عامل، هر نوع، دستگاه فیزیکی می‌تواند بطور کاملاً متفاوتی در کنار دستگاه‌های دیگر بکار بپردازد. حتی دستورات خواندن و نوشتن داده‌ها از دستگاه‌ها می‌توانند بطور کلی با یکدیگر متفاوت باشند. اجازه دهید تا به بررسی کاربرد دیگری از چند ریختی بپردازیم. مدیر صفحه نیاز به نمایش انواع مختلفی از شی‌ها شامل انواع شی‌های جدیدی دارد که برنامه‌نویس پس از نوشتن مدیر صحنه به آن اضافه خواهد کرد. همچنین سیستم می‌تواند نیازمند به نمایش انواع مختلفی از شکل‌ها همانند دایره‌ها، مثلث‌ها یا مستطیل‌ها شود که از کلاس مبنای انتزاعی **Shape** مشتق شده‌اند. مدیر صحنه از اشاره‌گرهای **Shape** برای مدیریت شی‌های که به نمایش در می‌آیند استفاده می‌کند. برای ترسیم هر شی (صرفنظر از سطحی که کلاس شی در سلسله مراتب توارث قرار دارد)، مدیر صحنه از یک اشاره‌گر کلاس مبنا استفاده می‌کند تا تابع **draw** برای آن شی را فراخوانی کند، که یک تابع **virtual** محض در کلاس مبنای **Shape** است، بنابر این هر کلاس غیرانتزاعی مشتق شده باید تابع **draw** را پیاده‌سازی کند. هر شی **Shape** در سلسله مراتب توارث از چگونگی ترسیم خود مطلع است. نیازی نیست که مدیر صحنه نگران نوع هر شی بوده یا اینکه نگران آن باشد که قبلاً با آن شی مواجه شده است یا نه.

غالباً در برنامه‌نویسی شیءگرا، یک کلاس *تکرار شونده* (iterator) تعریف می‌کنند که می‌تواند در میان تمام شیءهای موجود در یک حامل (همانند یک آرایه) حرکت کند. برای مثال، برنامه می‌تواند لیستی از شیءهای موجود در یک لیست پیوندی را با ایجاد یک شیء تکرار شونده به چاپ در آورده و سپس با فراخوانی مجدد تکرار شونده، به عنصر بعدی در لیست دست یابد. معمولاً از تکرار شونده‌ها در برنامه‌نویسی پولی‌مورفیک برای پیمایش یک آرایه یا لیست پیوندی از سطح‌های مختلف یک سلسله مراتب استفاده می‌شود. اشاره‌گرها در چنین لیستی تماماً اشاره‌گرهای کلاس مبنا هستند. برای مثال، لیستی از شیءهایی کلاس مبنا **TwoDimensionalShape** می‌تواند، حاوی شیءهایی از کلاس‌های **Square**،

www.yourshop.4kia.ir

Circle، **Triangle** و غیره باشد. با استفاده از پلی‌مورفیزم یک پیغام **draw** به هر شیی در لیست ارسال می‌شود و آن شیی بدرستی بر روی صفحه ترسیم می‌گردد.

٦-۱۳ مبحث آموزشی: سیستم پرداخت حقوق با استفاده از چند ریختی

در این بخش به بررسی مجدد سلسله مراتب **CommissionEmployee-BasePlusCommissionEmployee** که در بخش ۴-۱۲ به معرفی آن اقدام کردیم، می‌پردازیم. در این مثال، از کلاس انتزاعی و چند ریختی برای انجام محاسبات پرداخت حقوق برحسب نوع کارمند استفاده می‌کنیم. سلسله مراتب کارمندی که در این بخش ایجاد می‌کنیم قادر به حل مسئله زیر است:

شرکتی به کارمندان خود بطور هفتگی حقوق پرداخت می‌کند. کارمندان به چهار دسته تقسیم شده‌اند: کارمندانی که یک حقوق ثابت صرفنظر از ساعات کاری در هفته دریافت می‌کنند، کارمندانی که براساس ساعت کاری و اضافه کاری در طول هفته مازاد بر 40 ساعت حقوق دریافت می‌کنند، کارمندانی که براساس فروش حقوق دریافت می‌کنند و کارمندانی که علاوه بر حقوق ثابت درصدی از فروش نیز کمیسیون به آنها تعلق می‌گیرد. شرکت تصمیم دارد که تا پرداخت حقوق‌های جاری به کارمندانی که حقوق پایه همراه با کمیسیون از فروش دریافت می‌کنند، ۱۰ درصد به میزان فروش آنها پاداش اضافه نماید. شرکت مایل به پیاده‌سازی یک برنامه‌ای است تا محاسبات پرداخت حقوق را به روش چند ریختی انجام دهد.

از کلاس انتزاعی **Employee** برای عرضه مفهوم کلی یک کارمند استفاده می‌کنیم. کلاس‌های که مستقیماً از **Employee** مشتق می‌شوند عبارتند از **SalariedEmployee**، **CommissionEmployee** و **HourlyEmployee**. کلاس **BasePlusCommissionEmployee** از کلاس **CommissionEmploye** مشتق شده و نشاندهنده آخرین نوع کارمند است. دیاگرام UML این کلاس در شکل ۱۱-۱۳ بنمایش درآمده و سلسله مراتب توارث را برای برنامه پرداخت حقوق به روش چندریختی را نشان می‌دهد. دقت کنید که نام کلاس **Employee** بصورت ایتالیک (کج) نوشته شده که قراردادی در UML می‌باشد.

شکل ۱۱-۳ | دیاگرام UML کلاس سلسله مراتب Employee.

کلاس مبنای انتزاعی **Employee** اعلان کننده یک «*واسط*» یا "*interface*" برای سلسله مراتب است، که مجموعه‌ای از توابع عضو می‌باشد که برنامه می‌تواند بر روی تمام شی‌های **Employee** فراخوانی کند. هر کارمندی صرفنظر از روش محاسبه حقوق وی، دارای نام، نام خانوادگی و شماره تامین اجتماعی بوده از اینرو اعضای داده خصوصی عبارتند از: **firstName**، **lastName** و **socialSecurityNumber** که در کلاس مبنای انتزاعی **Employee** ظاهر می‌شوند.

در بخش‌های زیر اقدام به پیاده‌سازی سلسله مراتب کلاس **Employee** خواهیم کرد. در پنج بخش اول یک کلاس انتزاعی یا غیرانتزاعی ایجاد می‌کنیم. در بخش پایانی یک برنامه تست پیاده‌سازی می‌نمائیم که شی‌های از تمام این کلاس‌ها ایجاد کرده و آنها را به روش چند ریختی پردازش می‌کند.

۱-٦-۱۳ ایجاد کلاس مبنای انتزاعی Employee

www.yourshop.4kia.ir

کلاس **Employee** (شکل‌های ۱۳-۱۳ و ۱۴-۱۳ که در بخش‌های بعدی توضیح داده خواهند شد) توابع **earnings** و **print** به همراه توابع متعدد *get* و *set* که اعضای داده **Employee** را فراهم می‌آورند، تدارک دیده است. تابع **earnings** بطور مشخصی بر روی تمام کارمندان اعمال می‌شود. اما محاسبه هر حقوق بستگی به کلاس کارمند دارد. از اینرو **earnings** را بصورت **virtual** محض در کلاس مبنای **employee** اعلان کرده‌ایم چرا که در پیاده‌سازی پیش‌فرض راضی کننده نیست یعنی اطلاعات کافی برای تعیین میزان حقوق پرداختی که باید برگشت داده شود وجود ندارد. هر کلاس مشتق شده‌ای تابع **earnings** را با پیاده‌سازی مقتضی بکار می‌گیرد. برای محاسبه حقوق یک کارمند برنامه آدرس شی کارمند را به اشاره‌گر کلاس مبنا تخصیص می‌دهد، سپس تابع **earnings** بر روی آن شی فراخوانی می‌گردد.

برای نگهداری اشاره‌گرهای **Employee** از یک **vector** استفاده کرده‌ایم که هر کدام به یک شی **Employee** اشاره کند (البته، آنها نمی‌توانند شی‌های **Employee** باشند چرا که **Employee** یک کلاس انتزاعی است، با این وجد بدلیل توارث، هر شی از تمام کلاس‌های مشتق شده از **Employee** شی‌های از آن محسوب می‌شوند). برنامه در میان **vector** حرکت کرده وتابع **earnings** را برای هر شی کارمند فراخوانی می‌کند. C++ این فراخوانی‌های تابع را به روش چند ریختی پردازش می‌کند. با توجه به اینکه **earnings** بصورت یک تابع **virtual** محض در **Employee** اعلان شده، هر کلاس که مستقیماً از **Employee** مشتق شود که می‌خواهد بصورت یک کلاس غیرانتزاعی باشد سبب **override** (انتخاب تابع مناسب) شدن **earnings** می‌شود. با اینکار طراح کلاس سلسله مراتب خواهد توانست برای هر کلاس مشتق شده محاسبه مقتضی را داشته باشد، در صورتیکه کلاس مشتق شده براستی غیرانتزاعی باشد.

تابع **print** در کلاس **Employee** مبادرت به نمایش نام، نام خانوادگی و شماره تامین اجتماعی کارمند می‌کند. همانطوری که مشاهده خواهید کرد، هر کلاس مشتق شده از **Employee** تابع مناسب **print** را انتخاب کرده و نوع کارمند را در خروجی چاپ می‌کند (مانند "salaried employee:") و به دنبال آن مابقی اطلاعات کارمند را قرار می‌دهد.

دیاگرام شکل ۱۲-۱۳ نمایشی از پنج کلاس موجود در سلسله مراتب است که در سمت چپ آن و توابع **earnings** و **print** در سرستون‌ها قرار گرفته‌اند. برای هر کلاس، دیاگرام نتیجه دلخواه هر تابع را نشان می‌دهد. دقت کنید که کلاس **Employee** با **"0="** برای تابع **earnings** همراه شده و نشان می‌دهد که این تابع یک تابع **virtual** محض است. هر کلاس مشتق شده تابع مناسب خود را برای انجام مقاصد مقتضی انتخاب می‌کند (یعنی تابع **earnings** را **override** می‌کند). در این دیاگرام توابع **get** و **set**

www.yourshop.4kia.ir

کلاس مبنای **Employee** را لیست نکرده‌ایم، چرا که آنها هیچ یک از توابع در کلاس‌های مشتق شده را **override** نمی‌کنند.

اجازه دهید تا به بررسی فایل سرآیند **Employee** (شکل ۱۳-۱۳) بپردازیم. توابع عضو **public** شامل یک سازنده (که نام، نام خانوادگی و شماره تامین اجتماعی را بعنوان آرگومان دریافت کرده (خط 12))، توابع *set* (که تنظیم کننده نام، نام خانوادگی و شماره تامین اجتماعی است (خطوط 17,14 و 20))، توابع *get* (که نام، نام خانوادگی و شماره تامین اجتماعی را برگشت می‌دهند (خطوط 18,15 و 21))، تابع **virtual** محض **earnings** (خط 21) و تابع **print** که **virtual** است، می‌باشند (خط 25).

بخاطر دارید که تابع **earnings** را بصورت یک تابع **virtual** محض اعلان کرده‌ایم، چرا که بایستی ابتدا از نوع کارمند مطلع شویم تا بتوانیم محاسبه مناسب برای حقوق آن نوع کارمند را تعیین نمائیم. اعلان این تابع بعنوان **virtual** محض بر این نکته دلالت دارد که هر کلاس مشتق شده غیرانتزاعی بایستی یک پیاده‌سازی مقتضی از **earnings** تدارک دیده و برنامه بتواند از اشاره‌گرهای **Employee** برای فراخوانی تابع **earnings** به روش چند ریختی برای هر نوع کارمند استفاده کند.

شکل ۱۴-۱۳ حاوی پیاده‌سازی تابع عضو برای کلاس **Employee** است. هیچ پیاده‌سازی برای تابع **earnings** که **virtual** است تدارک دیده نشده است. به سازنده **Employee** (خطوط 15-10) توجه کنید که مبادرت به اعتبارسنجی شماره تامین اجتماعی نمی‌کند. معمولاً بایستی چنین اعتبارسنجی در نظر گرفته شود.

	earnings	print
Employee	**=0**	*firstName lastName* **social security number:** *SSN*
Salaried-Employee	**weeklySalary**	**salaried employee:** *firstName lastName* **social security number:** *SSN* **weekly salary:** *weeklysalary*
Hourly-Employee	*if hours <= 40* **wage * hours** *if hours > 40* **(40 * wage) +** **((hours – 40)** *** wage * 1.5)**	**hourly employee:** *firstName lastName* **social security number:** *SSN* **hourly wage:** *wage*; **hours worked:** *hours*
Commission-Employee	**commissionRate * grossSales**	**commission employee:** *firstName lastName* **social security number:** *SSN* **gross sales:** *grossSales*; **commission rate:** *commissionRate*
BasePlus-Commission-Employee	**baseSalary + (commissionRate * grossSales)**	**base salaried commission employee:** *firstName lastName* **social security number:** *SSN* **gross sales:** *grossSales*; **commission rate:** *commissionRate;*

www.yourshop.4kia.ir

		base salary: *baseSalary*

شکل ۱۲-۱۳ | واسط چند ریختی برای سلسله مراتب کلاس‌های Employee.

```
1  // Fig. 13.13: Employee.h
2  // Employee abstract base class.
3  #ifndef EMPLOYEE_H
4  #define EMPLOYEE_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
9  class Employee
10 {
11 public:
12    Employee( const string &, const string &, const string & );
13
14    void setFirstName( const string & ); // set first name
15    string getFirstName() const; // return first name
16
17    void setLastName( const string & ); // set last name
18    string getLastName() const; // return last name
19
20    void setSocialSecurityNumber( const string & ); // set SSN
21    string getSocialSecurityNumber() const; // return SSN
22
23    // pure virtual function makes Employee abstract base class
24    virtual double earnings() const = 0; // pure virtual
25    virtual void print() const; // virtual
26 private:
27    string firstName;
28    string lastName;
29    string socialSecurityNumber;
30 }; // end class Employee
31
32 #endif // EMPLOYEE_H
```

شکل ۱۳-۱۳| فایل سرآیند کلاس Employee.

```
1  // Fig. 13.14: Employee.cpp
2  // Abstract-base-class Employee member-function definitions.
3  // Note: No definitions are given for pure virtual functions.
4  #include <iostream>
5  using std::cout;
6
7  #include "Employee.h" // Employee class definition
8
9  // constructor
10 Employee::Employee( const string &first, const string &last,
11    const string &ssn )
12    : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
13 {
14    // empty body
15 } // end Employee constructor
16
17 // set first name
18 void Employee::setFirstName( const string &first )
19 {
20    firstName = first;
21 } // end function setFirstName
22
23 // return first name
24 string Employee::getFirstName() const
25 {
26    return firstName;
27 } // end function getFirstName
28
29 // set last name
30 void Employee::setLastName( const string &last )
31 {
32    lastName = last;
33 } // end function setLastName
```

www.yourshop.4kia.ir

```
34
35 // return last name
36 string Employee::getLastName() const
37 {
38    return lastName;
39 } // end function getLastName
40
41 // set social security number
42 void Employee::setSocialSecurityNumber( const string &ssn )
43 {
44    socialSecurityNumber = ssn; // should validate
45 } // end function setSocialSecurityNumber
46
47 // return social security number
48 string Employee::getSocialSecurityNumber() const
49 {
50    return socialSecurityNumber;
51 } // end function getSocialSecurityNumber
52
53 // print Employee's information (virtual, but not pure virtual)
54 void Employee::print() const
55 {
56    cout << getFirstName() << ' ' << getLastName()
57       << "\nsocial security number: " << getSocialSecurityNumber();
58 } // end function print
```

شکل ١٤-١٣| فایل پیاده‌سازی کلاس Employee.

به تابع **print** که **virtual** است (شکل ۱۴-۱۳، خطوط 58-54) توجه نمائید که دارای پیاده‌سازی بوده و در هر کلاس مشتق شده override می‌شود (یعنی به کنار گذاشته شده و تابع متناسب برای آن کلاس انتخاب و اجرا می‌گردد). با این همه، هر یک از این توابع از **print** نسخه کلاس انتزاعی برای چاپ اطلاعات مشترک در تمام کلاس‌های موجود در سلسله مراتب **Employee** استفاده می‌کنند.

۲-٦-۱۳ ایجاد کلاس مشتق شده غیرانتزاعی SalariedEmployee

کلاس **SalariedEmployee** (شکل‌های ۱۵-۱۳ و ۱۶-۱۳) از کلاس **Employee** مشتق شده است (خط 8 از شکل ۱۵-۱۳). توابع عضو سراسری (public) شامل سازنده‌ای هستند که نام، نام خانوادگی، شماره تامین اجتماعی و حقوق هفتگی را بعنوان آرگومان می‌پذیرد (خطوط 12-11)، یک تابع *set* برای تخصیص مقادیر جدید غیرمنفی به عضو داده **weeklySalary** (خط 14)، یک تابع *get* برای برگشت دادن مقدار **weeklySalary** (خط 15)، یک تابع **earnings** که **virtual** بوده و حقوق کارمندی از نوع **SalariedEmployee** را محاسبه می‌کند (خط 18) و یک تابع **print** که **virtual** بوده که نوع کارمند را یعنی: "salaried employee" و بدنبال آن اطلاعات خاص آن کارمند را که توسط تابع **print** کلاس مبنای **Employee** و تابع **getWeeklySalary** کلاس **SalariedEmployee** تهیه شده است، چاپ می‌کند.

برنامه شکل ۱۶-۱۳ حاوی پیاده‌سازی تابع عضو برای **SalariedEmployee** است. سازنده کلاس مبادرت به ارسال نام، نام خانوادگی و شماره تامین اجتماعی به سازنده **Employee** می‌کند (خط 11) تا اعضای داده **private** را که از کلاس مبنا به ارث برده شده، اما در دسترس کلاس مشتق شده نمی‌باشند،

www.yourshop.4kia.ir

مقداردهی اولیه شوند. تابع **earnings** در خطوط 33-30 مبادرت به توقف تابع **earnings** در **Employee** می‌کند که **virtual** محض است تا پیاده‌سازی غیرانتزاعی تدارک دیده شده برای **SalariedEmployee** حقوق هفتگی را برگشت دهد. اگر **earnings** را پیاده‌سازی نمی‌کردیم، کلاس **SalariedEmployee** می‌خواست که یک کلاس انتزاعی باشد و نتیجه هر عملی برای نمونه‌سازی یک شی از کلاس، خطای کامپایل بود.

```
1  // Fig. 13.15: SalariedEmployee.h
2  // SalariedEmployee class derived from Employee.
3  #ifndef SALARIED_H
4  #define SALARIED_H
5
6  #include "Employee.h" // Employee class definition
7
8  class SalariedEmployee : public Employee
9  {
10 public:
11    SalariedEmployee( const string &, const string &,
12       const string &, double = 0.0 );
13
14    void setWeeklySalary( double ); // set weekly salary
15    double getWeeklySalary() const; // return weekly salary
16
17    // keyword virtual signals intent to override
18    virtual double earnings() const; // calculate earnings
19    virtual void print() const; // print SalariedEmployee object
20 private:
21    double weeklySalary; // salary per week
22 }; // end class SalariedEmployee
23
24 #endif // SALARIED_H
```

شکل ۱۵-۱۳ | فایل سرآیند کلاس SalariedEmployee.

```
1  // Fig. 13.16: SalariedEmployee.cpp
2  // SalariedEmployee class member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "SalariedEmployee.h" // SalariedEmployee class definition
7
8  // constructor
9  SalariedEmployee::SalariedEmployee( const string &first,
10    const string &last, const string &ssn, double salary )
11    : Employee( first, last, ssn )
12 {
13    setWeeklySalary( salary );
14 } // end SalariedEmployee constructor
15
16 // set salary
17 void SalariedEmployee::setWeeklySalary( double salary )
18 {
19    weeklySalary = ( salary < 0.0 ) ? 0.0 : salary;
20 } // end function setWeeklySalary
21
22 // return salary
23 double SalariedEmployee::getWeeklySalary() const
24 {
25    return weeklySalary;
26 } // end function getWeeklySalary
27
28 // calculate earnings;
29 // override pure virtual function earnings in Employee
30 double SalariedEmployee::earnings() const
31 {
32    return getWeeklySalary();
```

www.yourshop.4kia.ir

```
33 } // end function earnings
34
35 // print SalariedEmployee's information
36 void SalariedEmployee::print() const
37 {
38    cout << "salaried employee: ";
39    Employee::print(); // reuse abstract base-class print function
40    cout << "\nweekly salary: " << getWeeklySalary();
41 } // end function print
```

شکل ۱۶-۱۳ | فایل پیاده‌سازی کلاس SalariedEmployee.

به فایل سرآیند در کلاس **SalariedEmployee** توجه کنید که در آن توابع عضو **earnings** و **print** را بصورت **virtual** اعلان کرده‌ایم (خطوط 19-18 از شکل ۱۵-۱۳)، در واقع قرار دادن کلمه کلیدی **virtual** قبل از این توابع عضو اضافی است. ما آنها را بعنوان **virtual** در کلاس مبنای **Employee** اعلان کرده‌ایم، از اینرو آنها در کل سلسله مراتب کلاس بصورت توابع **virtual** باقی خواهد ماند.

تابع **print** از کلاس **SalariedEmployee** (خطوط 41-36 از شکل ۱۶-۱۳) مبادرت به متوقف ساختن تابع **print** از کلاس مبنای **Employee** می‌کند. اگر کلاس **SalaredEmployee** این تابع **print** را متوقف یا **override** نمی‌کرد، این کلاس، نسخه **print** از کلاس **Employee** را به ارث می‌برد. در چنین وضعی، تابع **print** کلاس **SalariedEmployee** فقط نام کامل و شماره تامین اجتماعی کارمند را برگشت می‌داد که نشاندهنده اطلاعات کافی در مورد این نوع کارمند نیست. برای چاپ اطلاعات کامل کارمندی از نوع **SalariedEmployee**، تابع **print** کلاس مشتق شده، عبارت "Salaried employee:" را چاپ و بدنبال آن اطلاعات خاص کلاس مبنای **Employee** (یعنی نام، نام خانوادگی و شماره تامین اجتماعی) را با فراخوانی تابع **print** کلاس مبنا با استفاده از عملگر تفکیک قلمرو (خط **39**) قرار می‌دهد. خروجی تولید شده توسط تابع **print** کلاس **SalariedEmployee** حاوی حقوق هفتگی کارمند است که با فراخوانی تابع **getWeeklySalary** تهیه شده است.

۳-۶-۱۳ ایجاد کلاس مشتق شده غیرانتزاعی HourlyEmployee

کلاس **HourlyEmployee** (شکل‌های ۱۷-۱۳ و ۱۸-۱۳) نیز از کلاس **Employee** مشتق شده است (خط 8 از شکل ۱۷-۱۳) توابع عضو سراسری شامل یک سازنده (خطوط 12-11) هستند که آرگومان‌های بعنوان نام، نام خانوادگی، شماره تامین اجتماعی، دستمزد ساعتی و تعداد ساعات کارکرد در هفته را دریافت می‌کند، توابع *set* که مقادیر جدید را به اعضای داده **wage** و **hours** تخصیص می‌دهند (خطوط 17 و 14)، توابع *get* که مبادرت به برگشت دادن مقادیر **wage** و **hours** می‌کنند (خطوط 15 و 18)، تابع **earnings** که **virtual** بوده و حقوق یک کارمند از نوع **HourlyEmployee** را محاسبه می‌کند (خط 21) و تابع **print** که آن هم **virtual** است و جمله: "hourly employee:"و اطلاعات خاص کارمند را چاپ می‌کند (خط 22).

```
1  // Fig. 13.17: HourlyEmployee.h
2  // HourlyEmployee class definition.
```

www.yourshop.4kia.ir

```
3  #ifndef HOURLY_H
4  #define HOURLY_H
5
6  #include "Employee.h" // Employee class definition
7
8  class HourlyEmployee : public Employee
9  {
10 public:
11    HourlyEmployee( const string &, const string &,
12       const string &, double = 0.0, double = 0.0 );
13
14    void setWage( double ); // set hourly wage
15    double getWage() const; // return hourly wage
16
17    void setHours( double ); // set hours worked
18    double getHours() const; // return hours worked
19
20    // keyword virtual signals intent to override
21    virtual double earnings() const; // calculate earnings
22    virtual void print() const; // print HourlyEmployee object
23 private:
24    double wage; // wage per hour
25    double hours; // hours worked for week
26 }; // end class HourlyEmployee
27
28 #endif // HOURLY_H
```

شکل ۱۷-۱۳ | فایل سرآیند کلاس HourlyEmployee.

```
1  // Fig. 13.18: HourlyEmployee.cpp
2  // HourlyEmployee class member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "HourlyEmployee.h" // HourlyEmployee class definition
7
8  // constructor
9  HourlyEmployee::HourlyEmployee( const string &first, const string &last,
10    const string &ssn, double hourlyWage, double hoursWorked )
11    : Employee( first, last, ssn )
12 {
13    setWage( hourlyWage ); // validate hourly wage
14    setHours( hoursWorked ); // validate hours worked
15 } // end HourlyEmployee constructor
16
17 // set wage
18 void HourlyEmployee::setWage( double hourlyWage )
19 {
20    wage = ( hourlyWage < 0.0 ? 0.0 : hourlyWage );
21 } // end function setWage
22
23 // return wage
24 double HourlyEmployee::getWage() const
25 {
26    return wage;
27 } // end function getWage
28
29 // set hours worked
30 void HourlyEmployee::setHours( double hoursWorked )
31 {
32    hours = ( ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) ) ?
33       hoursWorked : 0.0 );
34 } // end function setHours
35
36 // return hours worked
37 double HourlyEmployee::getHours() const
38 {
39    return hours;
40 } // end function getHours
41
42 // calculate earnings;
```

www.yourshop.4kia.ir

```
43 // override pure virtual function earnings in Employee
44 double HourlyEmployee::earnings() const
45 {
46    if ( getHours() <= 40 ) // no overtime
47       return getWage() * getHours();
48    else
49       return 40 * getWage() + ( (getHours() - 40) * getWage() * 1.5 );
50 } // end function earnings
51
52 // print HourlyEmployee's information
53 void HourlyEmployee::print() const
54 {
55    cout << "hourly employee: ";
56    Employee::print(); // code reuse
57    cout << "\nhourly wage: " << getWage() <<
58       "; hours worked: " << getHours();
59 } // end function print
```

شکل ۱۸-۱۳ | فایل پیاده‌سازی کلاس HourlyEmployee.

برنامه شکل ۱۸-۱۳ حاوی پیاده‌سازی تابع عضو برای کلاس **HourlyEmployee** است. خطوط 21-18 و 34-30 تعریف کننده توابع *set* هستند که مقادیر جدید را به اعضای داده **wage** و **hours** تخصیص می‌دهند. تابع **setWage** در خطوط 21-18 ما را مطمئن می‌سازد که **wage** یک مقدار غیرمنفی است و تابع **setHours** در خطوط 34-30 هم ما را مطمئن می‌کند که عضو داده **hours** مابین 0 و 168 (مجموع کل ساعات کارکرد در یک هفته) قرار دارد. توابع *get* در خطوط 27-24 و 40-37 پیاده‌سازی شده‌اند. این توابع را بصورت **virtual** اعلان نکرده‌ایم، از اینرو کلاس‌های مشتق شده از کلاس **HourlyEmployee** نمی‌توانند آنها را override کن‌ند. به سازنده **HourlyEmployee** توجه کنید، همانند سازنده **SalariedEmployee**، مبادرت به ارسال نام، نام خانوادگی و شماره تامین اجتماعی به سازنده کلاس مبنا **Employee** می‌کند (خط 11) تا اعضای داده **private** ارث برده شده و اعلان شده در کلاس مبنا مقداردهی اولیه گردد. علاوه بر این، تابع **print** این کلاس مبادرت به فراخوانی تابع **print** کلاس مبنا (خط 56) می‌کند تا اطلاعات خاص کارمند (یعنی نام و نام خانوادگی و شماره تامین اجتماعی) چاپ شود.

۱۳-٦-٤ ایجاد کلاس مشتق شده غیرانتزاعی CommissionEmployee

کلاس **CommissionEmployee** (شکل‌های ۱۹-۱۳ و ۲۰-۱۳) از کلاس **Employee** (خط 8 از شکل ۱۹-۱۳) مشتق شده است. پیاده‌سازی تابع عضو (شکل ۲۰-۱۳) شامل یک سازنده در خطوط 15-9 است که نام، نام خانوادگی، شماره تامین اجتماعی، میزان حقوق و نرخ کمیسیون را اخذ می‌کند، توابع *set* (خطوط 21-18 و 33-30) برای تخصیص مقادیر جدید به اعضای **CommissionRate** و **grossSales** بکار گرفته شده‌اند، توابع *get* (خطوط 27-24 و 39-36) که مقادیر این اعضای داده را بازیابی می‌کنند، تابع **earnings** (خطوط 46-43) برای محاسبه حقوق کارمندی از نوع **CommissionEmployee**، و تابع **print** در خطوط 55-49 که نوع کارمند را بصورت "commission employee:" و اطلاعات خاص

www.yourshop.4kia.ir

کارمند را چاپ می‌کند. همچنین سازنده **CommissionEmployee** نام، نام خانوادگی و شماره تامین اجتماعی را به سازنده **Employee** در خط 11 ارسال می‌کند تا با اعضاء داده **private** کلاس **Employee** مقداردهی اولیه شوند. تابع **print**، تابع **print** کلاس مبنا را فراخوانی می‌کند (خط 52) تا اطلاعات خاص **Employee** به نمایش در آید (نام، نام خانوادگی و شماره تامین اجتماعی).

```
1  // Fig. 13.19: CommissionEmployee.h
2  // CommissionEmployee class derived from Employee.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include "Employee.h" // Employee class definition
7
8  class CommissionEmployee : public Employee
9  {
10 public:
11    CommissionEmployee( const string &, const string &,
12       const string &, double = 0.0, double = 0.0 );
13
14    void setCommissionRate( double ); // set commission rate
15    double getCommissionRate() const; // return commission rate
16
17    void setGrossSales( double ); // set gross sales amount
18    double getGrossSales() const; // return gross sales amount
19
20    // keyword virtual signals intent to override
21    virtual double earnings() const; // calculate earnings
22    virtual void print() const; // print CommissionEmployee object
23 private:
24    double grossSales; // gross weekly sales
25    double commissionRate; // commission percentage
26 }; // end class CommissionEmployee
27
28 #endif // COMMISSION_H
```

شکل ۱۹-۱۳ |فایل سرآیند کلاس CommissionEmploye.

```
1  // Fig. 13.20: CommissionEmployee.cpp
2  // CommissionEmployee class member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8  // constructor
9  CommissionEmployee::CommissionEmployee( const string &first,
10    const string &last, const string &ssn, double sales, double rate )
11    : Employee( first, last, ssn )
12 {
13    setGrossSales( sales );
14    setCommissionRate( rate );
15 } // end CommissionEmployee constructor
16
17 // set commission rate
18 void CommissionEmployee::setCommissionRate( double rate )
19 {
20     commissionRate = ( ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0 );
21 } // end function setCommissionRate
22
23 // return commission rate
24 double CommissionEmployee::getCommissionRate() const
25 {
26     return commissionRate;
27 } // end function getCommissionRate
28
29 // set gross sales amount
30 void CommissionEmployee::setGrossSales( double sales )
```

www.yourshop.4kia.ir

```
31 {
32    grossSales = ( ( sales < 0.0 ) ? 0.0 : sales );
33 } // end function setGrossSales
34
35 // return gross sales amount
36 double CommissionEmployee::getGrossSales() const
37 {
38     return grossSales;
39 } // end function getGrossSales
40
41 // calculate earnings;
42 // override pure virtual function earnings in Employee
43 double CommissionEmployee::earnings() const
44 {
45    return getCommissionRate() * getGrossSales();
46 } // end function earnings
47
48 // print CommissionEmployee's information
49 void CommissionEmployee::print() const
50 {
51    cout << "commission employee: ";
52    Employee::print(); // code reuse
53    cout << "\ngross sales: " << getGrossSales()
54       << "; commission rate: " << getCommissionRate();
55 } // end function print
```

شکل ۲۰-۱۳ | فایل پیاده‌سازی کلاس CommssionEmployee.

۵-٦-۱۳ ایجاد غیرمستقیم کلاس مشتق شده غیرانتزاعی BasePlusCommissionEmployee

کلاس **BasePlusCommissionEmployee** (شکل‌های ۲۱-۱۳ و ۲۲-۱۳) بطور مستقیم از کلاس **CommissionEmployee** (خط 8 از شکل ۲۱-۱۳) ارث‌بری دارد و بنابر این یک کلاس مشتق شده غیرمستقیم از کلاس **Employee** است. پیاده‌سازی تابع عضو کلاس **BasePlusCommissionEmployee** شامل یک سازنده (خط 16-10 از شکل ۲۲-۱۳) است که آرگومان‌های بعنوان نام، نام خانوادگی. شماره تامین اجتماعی، میزان حقوق، نرخ کمیسیون و حقوق پایه دریافت می‌کند. سپس نام، نام خانوادگی، شماره تامین اجتماعی، میزان حقوق و نرخ کمیسیون را به سازنده **CommssionEmployee** ارسال می‌کند (خط 13) تا اعضای به ارث رفته مقداردهی اولیه شوند. همچنین کلاس **BasePlusCommissionEmployee** حاوی یک تابع *set* (خطوط 22-19) برای تخصیص مقدار جدید به عضو داده **baseSalary** و یک تابع *get* (خطوط 28-25) برای برگشت دادن مقدار **baseSalary** است. تابع **earnings** در خطوط 35-32 حقوق کارمندی از این نوع را محاسبه می‌کند. توجه کنید که خط 34 در تابع **earnings** تابع **earnings** کلاس مبنای **CommissionEmployee** را برای محاسبه آن بخش از حقوق را که از کمیسیون تامین می‌شود، فراخوانی می‌کند. تابع **print** کلاس **BasePlusCommissionEmployee** (خطوط 43-38) جمله "base-salaried" و بدنبال آن خروجی تابع **print** کلاس مبنای **CommissionEmployee** را چاپ می‌کند. در نتیجه خروجی شامل جمله: "base-salaried commission employee:" و بدنبال آن مابقی اطلاعات **BasePlusCommissionEmployee** خواهد بود.

```
1  // Fig. 13.21: BasePlusCommissionEmployee.h
```

www.yourshop.4kia.ir

```
// BasePlusCommissionEmployee class derived from Employee.
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include "CommissionEmployee.h" // CommissionEmployee class definition

class BasePlusCommissionEmployee : public CommissionEmployee
{
public:
   BasePlusCommissionEmployee( const string &, const string &,
      const string &, double = 0.0, double = 0.0, double = 0.0 );

   void setBaseSalary( double ); // set base salary
   double getBaseSalary() const; // return base salary

   // keyword virtual signals intent to override
   virtual double earnings() const; // calculate earnings
   virtual void print() const; //print BasePlusCommissionEmployee object
private:
   double baseSalary; // base salary per week
}; // end class BasePlusCommissionEmployee

#endif // BASEPLUS_H
```

شکل ۲۱-۱۳ | فایل سرآیند کلاس BasePlusCommissionEmployee.

```
// Fig. 13.22: BasePlusCommissionEmployee.cpp
// BasePlusCommissionEmployee member-function definitions.
#include <iostream>
using std::cout;

// BasePlusCommissionEmployee class definition
#include "BasePlusCommissionEmployee.h"

// constructor
BasePlusCommissionEmployee::BasePlusCommissionEmployee(
   const string &first, const string &last, const string &ssn,
   double sales, double rate, double salary )
   : CommissionEmployee( first, last, ssn, sales, rate )
{
   setBaseSalary( salary ); // validate and store base salary
} // end BasePlusCommissionEmployee constructor

// set base salary
void BasePlusCommissionEmployee::setBaseSalary( double salary )
{
   baseSalary = ( ( salary < 0.0 ) ? 0.0 : salary );
} // end function setBaseSalary

// return base salary
double BasePlusCommissionEmployee::getBaseSalary() const
{
    return baseSalary;
} // end function getBaseSalary

// calculate earnings;
// override pure virtual function earnings in Employee
double BasePlusCommissionEmployee::earnings() const
{
    return getBaseSalary() + CommissionEmployee::earnings();
} // end function earnings

// print BasePlusCommissionEmployee's information
void BasePlusCommissionEmployee::print() const
{
   cout << "base-salaried ";
   CommissionEmployee::print(); // code reuse
   cout << "; base salary: " << getBaseSalary();
} // end function print
```

شکل ۲۲-۱۳ | فایل پیاده‌سازی کلاس BasePlusCommissionEmployee.

www.yourshop.4kia.ir

بخاطر دارید که تابع **print** کلاس **CommissionEmployee** مبادرت به نمایش نام، نام خانوادگی و شماره تامین اجتماعی کارمند با فراخوانی تابع **print** از کلاس مبنای خود (یعنی **Employee**) می‌کرد. دقت کنید که تابع **print** کلاس **BasePlusCommissionEmployee** باعث راه‌اندازی زنجیره‌ای از فراخوانی‌های توابع می‌شود که در هر سه سطح سلسله مراتب **Employee** گسترش می‌یابد.

۶-۶-۱۳ شرح فرآیند چند ریختی

برای تست سلسله مراتب **Employee**، برنامه موجود در شکل ۲۳-۱۳ مبادرت به ایجاد یک شی از هر چهار شکل غیرانتزاعی بنام‌های **SalariedEmployee**، **HourlyEmployee**، **CommissionEmployee** و **BasePlusCommissionEmployee** می‌کند. برنامه با این شی‌ها کار می‌کند، ابتدا به روش مقیدسازی استاتیک، سپس چندریختی، با استفاده از برداری از اشاره‌گرهای **Employee**. خطوط 38-31 شی‌های از چهار کلاس غیرانتزاعی مشتق شده از کلاس **Employee** ایجاد می‌کنند. خطوط 51-43 اطلاعات و حقوق هر کارمند را در خروجی به نمایش در می‌آورند.

```
1  // Fig. 13.23: fig13_23.cpp
2  // Processing Employee derived-class objects individually
3  // and polymorphically using dynamic binding.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7  using std::fixed;
8
9  #include <iomanip>
10 using std::setprecision;
11
12 #include <vector>
13 using std::vector;
14
15 // include definitions of classes in Employee hierarchy
16 #include "Employee.h"
17 #include "SalariedEmployee.h"
18 #include "HourlyEmployee.h"
19 #include "CommissionEmployee.h"
20 #include "BasePlusCommissionEmployee.h"
21
22 void virtualViaPointer( const Employee * const ); // prototype
23 void virtualViaReference( const Employee & ); // prototype
24
25 int main()
26 {
27    // set floating-point output formatting
28    cout << fixed << setprecision( 2 );
29
30    // create derived-class objects
31    SalariedEmployee salariedEmployee(
32       "John", "Smith", "111-11-1111", 800 );
33    HourlyEmployee hourlyEmployee(
34       "Karen", "Price", "222-22-2222", 16.75, 40 );
35    CommissionEmployee commissionEmployee(
36       "Sue", "Jones", "333-33-3333", 10000, .06 );
37    BasePlusCommissionEmployee basePlusCommissionEmployee(
38       "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
39
40    cout << "Employees processed individually using static binding:\n\n";
41
42    // output each Employee's information and earnings using static binding
43    salariedEmployee.print();
```

www.yourshop.4kia.ir

```
44    cout << "\nearned $" << salariedEmployee.earnings() << "\n\n";
45    hourlyEmployee.print();
46    cout << "\nearned $" << hourlyEmployee.earnings() << "\n\n";
47    commissionEmployee.print();
48    cout << "\nearned $" << commissionEmployee.earnings() << "\n\n";
49    basePlusCommissionEmployee.print();
50    cout << "\nearned $" << basePlusCommissionEmployee.earnings()
51       << "\n\n";
52
53    // create vector of four base-class pointers
54    vector < Employee * > employees( 4 );
55
56    // initialize vector with Employees
57    employees[ 0 ] = &salariedEmployee;
58    employees[ 1 ] = &hourlyEmployee;
59    employees[ 2 ] = &commissionEmployee;
60    employees[ 3 ] = &basePlusCommissionEmployee;
61
62    cout<< "Employees processed polymorphically via dynamic binding:\n\n";
63
64    // call virtualViaPointer to print each Employee's information
65    // and earnings using dynamic binding
66    cout << "Virtual function calls made off base-class pointers:\n\n";
67
68    for ( size_t i = 0; i < employees.size(); i++ )
69       virtualViaPointer( employees[ i ] );
70
71    // call virtualViaReference to print each Employee's information
72    // and earnings using dynamic binding
73    cout << "Virtual function calls made off base-class references:\n\n";
74
75    for ( size_t i = 0; i < employees.size(); i++ )
76       virtualViaReference( *employees[ i ] ); // note dereferencing
77
78    return 0;
79 } // end main
80
81 // call Employee virtual functions print and earnings off a
82 // base-class pointer using dynamic binding
83 void virtualViaPointer( const Employee * const baseClassPtr )
84 {
85    baseClassPtr->print();
86    cout << "\nearned $" << baseClassPtr->earnings() << "\n\n";
87 } // end function virtualViaPointer
88
89 // call Employee virtual functions print and earnings off a
90 // base-class reference using dynamic binding
91 void virtualViaReference( const Employee &baseClassRef )
92 {
93    baseClassRef.print();
94    cout << "\nearned $" << baseClassRef.earnings() << "\n\n";
95 } // end function virtualViaReference
```

```
Employee processed individually using ststic binding:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary:300.00
```

www.yourshop.4kia.ir

```
earned $500.00

Employee processed polymorphically using dynamic binding:

Virtual function calls made off base-class pointers:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary:300.00
earned $500.00

Virtual function calls made off base-class references:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary:300.00
earned $500.00
```

شکل ۲۳-۱۳ | برنامه راه‌انداز سلسله مراتب کلاس Employee.

هر تابع عضو احض‌ار شده در خطوط 51-43 مثالی از مقیدسازی استاتیک در زمان کامپایل است، چرا که از اسامی دستگیرها (و نه اشاره‌گرها یا مراجعه‌ها) استفاده کرده‌ایم. کامپایلر قادر به شناسایی نوع هر شی بود و تعیین می‌کند که کدام تابع **print** و **earnings** فراخوانی شده است.

در خط 54 بردار **employee** اخذ شده که حاوی چهار اشاره‌گر **Employees** است. خط 57 مبادرت به هدایت **employees[0]** بطرف شی **SalariedEmployee** می‌کند. خط 58 مبادرت به هدایت **employees[1]** بطرف شی **hourlyEmployee**، خط 59 مبادرت به هدایت **employees[2]** بطرف شی **CommissionEmployee** و خط 60 مبادرت به هدایت **employees[3]** بطرف شی **BasePlusCommissionEmployee** کرده است. کامپایلر اجازه چنین تخصیص‌هایی را می‌دهد، چرا که

www.yourshop.4kia.ir

SalariedEmployee یک **Employee** است، **HourlyEmployee** یک **Employee** است، یک **CommissionEmployee** یک **Employee** است و یک **BasePlusCommissionEmployee** نیز یک **Employee** می‌باشد. بنابر این، می‌توانیم آدرس‌های **SalariedEmployee**، **HourlyEmployee**، **CommissionEmployee** و **BasePlusCommissionEmployee** را به اشاره‌گرهای کلاس مبنا **Employee** تخصیص دهیم.

عبارت **for** در خطوط 69-68 بردار **employees** را پیمایش کرده و تابع **virtualViaPointer** (خطوط 87-83) را برای هر عنصر موجود در **employees** فراخوانی می‌کند. تابع **virtualViaPointer** در پارامتر **baseClassPtr** (از نوع **const Employee * const**) آدرس ذخیره شده در یک عنصر **employees** را دریافت می‌کند. در هر بار فراخوانی، این تابع از **baseClassPtr** برای فراخوانی توابع **print** و **earnings** که مجازی یا **virtual** هستند استفاد می‌کند (خطوط 85 و 86). توجه کنید که تابع **virtualViaPointer** حاوی هیچ داده‌ای در ارتباط با نوع **SalariedEmployee**، **HourlyEmployee**، **CommissionEmployee** یا **BasePlusCommissionEmployee** نیست. تابع فقط اطلاعاتی در مورد نوع کلاس مبنا یعنی **Employee** دارد. بنابر این، در زمان کامپایل، کامپایلر نمی‌داند که کدام توابع کلاس غیرانتزاعی را از طریق **baseClassPtr** فراخوانی نماید. با اینحال در زمان اجرا، هر احضار تابع مجازی مبادرت به فراخوانی تابع بر روی شی می‌کند که **baseClassPtr** در آن زمان به آن اشاره دارد. خروجی برنامه نشان می‌دهد که براستی توابع مقتضی برای هر کلاس احضار شده و اطلاعات صحیح هر شی بنمایش درآمده است. برای نمونه، حقوق هفتگی برای کلاس **SalaryEmployee** بنمایش در آمده و ناخالص فروش برای **CommissionEmployee** و **BasePlusCommissionEmployee** نشان داده شده است. همچنین به محاسبه حقوق هر کارمند به روش چند ریختی در خط 86 توجه کنید که همان نتایج تولیدی به روش مقیدسازی استاتیکی است که در خطوط 44، 46 و 50 بدست آمده است.

در پایان، از یک عبارت **for** دیگر (خطوط 76-75) برای پیمایش **employees** و احضار تابع **virtualViaReference** بر روی هر عنصر در بردار استفاده شده است (خطوط 95-91). تابع **virtualViaReference** در پارامتر **baseClassRef** خود (از نوع **const Employee &**) یک مراجعه بفرم *dereference* کردن اشاره‌گر (یعنی دستیابی به اطلاعات از طریق آدرس موجود در یک اشاره‌گر) ذخیره شده در هر عنصر **employees** می‌پردازد (خط 76). در هر بار فراخوانی **virtualViaReference** توابع **print** و **earnings** که **virtual** هستند (خطوط 94-93) از طریق مراجعه **baseClassRef** احضار می‌شوند تا ثابت شود که پردازش چند ریختی با مراجعه‌های کلاس مبنا نیز بخوبی اتفاق می‌افتد. با احضار هر تابع **virtual**، تابعی بر روی شی که **baseClassRef** در زمان اجرا به آن اشاره دارد، فراخوانی می‌شود.

www.yourshop.4kia.ir

این مثالی دیگر از مقیدسازی دینامیکی است. خروجی تولید شده توسط مراجعه‌های کلاس مبنا با خروجی تولید شده توسط اشاره‌گرهای کلاس مبنا یکسان است.

۱۳-۷ چند ریختی، توابع virtual و مقیدسازی دینامیکی

++C بکارگیری روش چندریختی در برنامه‌ها را آسانتر کرده است. بطور کاملاً مشخص امکان استفاده از روش چندریختی در زبان‌های غیر شی‌گرا همانند C وجود دارد، اما انجام اینکار مستلزم پیچیدگی کار با اشاره‌گرها است که خود خطرات بالقوه‌ای برای برنامه می‌تواند داشته باشد.

در این بخش به بررسی نحوه عملکرد داخلی ++C در پیاده‌سازی چند ریختی، توابع **virtual** و مقیدسازی دینامیکی می‌پردازیم. با مطالعه این بخش اطلاعات اولیه و خوبی از نحوه کار این قابلیت‌ها بدست خواهید آورد. از همه مهمتر، این بخش به شما کمک می‌کند از کاری که چندریختی برایتان (منظور هزینه چندریختی است) در مصرف حافظه و زمان پردازنده انجام می‌دهد. مطلع شوید. همچنین به شما کمک می‌کند تا مشخص نمائید در چه مواقعی از چند ریختی استفاده کنید و در چه مواقعی آنرا به کنار بگذارید.

ابتدا به توضیح ساختمان داده‌ای می‌پردازیم که کامپایلر ++C در زمان اجرا به منظور پشتیبانی از چندریختی در زمان اجرا، تهیه می‌کند. مشاهده خواهید کرد که چند ریختی از طریق سه سطح اشاره‌گر ("سه‌گانه غیرمستقیم") صورت می‌گیرد. سپس نشان خواهیم داد که چگونه در زمان اجرا از این ساختارهای داده برای اجرای توابع **virtual** و انجام مقیدسازی دینامیکی مرتبط با چند ریختی استفاده می‌کند. توجه کنید که بحث ما یکی از روش‌های ممکنه پیاده‌سازی است و نه جزء اصول پیاده‌سازی زبان.

زمانیکه ++C مبادرت به کامپایل کلاسی می‌کند که دارای یک یا چندین تابع **virtual** است، یک *جدول تابع مجازی* یا (virtual function table) *vtable* برای آن کلاس ایجاد می‌کند. برنامه در هر بار اجرا، از این vtable برای انتخاب تابع صحیح، هر بار که یک تابع **virtual** برای آن کلاس فراخوانی می‌شود، استفاده می‌کند. سمت چپ‌ترین ستون در شکل ۱۳-۲۴ نشاندهنده vtable برای کلاس‌های **Employee**، **SalariedEmployee**، **HourlyEmployee**، **CommissionEmployee** و **BasePlusCommissionEmployee** است.

در vtable کلاس **Employee**، اشاره‌گر تابع اول با صفر تنظیم شده است (یعنی اشاره‌گر null). اینکار به این دلیل صورت گرفته که تابع **earnings** یک تابع **virtual** محض است و بنابر این فاقد پیاده‌سازی می‌باشد. اشاره‌گر تابع دوم به تابع **print** اشاره دارد، که نام کامل و شماره تامین اجتماعی کارمند را به نمایش در می‌آورد. هر کلاسی که دارای یک یا چند اشاره‌گر **null** در جدول vtable خود است یک کلاس انتزاعی می‌باشد. کلاس‌های که فاقد null در vtable هستند (همانند **SalareiedEmployee**،

www.yourshop.4kia.ir

HourlyEmployee، **CommissionEmployee** و **BasePlusCommissionEmployee**) کلاس‌های غیرانتزاعی می‌باشند.

کلاس **SalariedEmployee** مبادرت به override کردن تابع **earnings** می‌کند تا حقوق هفتگی کارمند را برگشت دهد، از اینرو اشاره‌گر تابع به تابع **earnings** از کلاس **SalariedEmployee** اشاره می‌کند. همچنین این کلاس مبادرت به override کردن تابع **print** می‌کند تا اشاره‌گر تابع متناظر به تابع عضو **SalariedEmployee** اشاره کند که جمله "salaried employee:" و بدنبال آن نام کارمند، شماره تامین اجتماعی و حقوق هفتگی چاپ شود.

اشاره‌گر تابع **earnings** در vtable کلاس **HourlyEmployee** به تابع **earnings** کلاس **HourlyEmployee** اشاره دارد که حاصلضرب دستمزد (wage) کارمند به تعداد ساعات (hours) کار را برگشت می‌دهد. اشاره‌گر تابع **print** به نسخه **HourlyEmployee** تابع اشاره دارد که جمله: "hourly employee:"، نام کارمند، شماره تامین اجتماعی، دستمزد ساعتی و ساعات کارکرد را چاپ می‌کند. هر دو مبادرت به override کردن توابع در کلاس **Employee** می‌کنند.

اشاره‌گر تابع **earnings** در vtable برای کلاس **CommissionEmployee** به تابع **earnings** این کلاس اشاره می‌کند که حاصلضرب فروش ناخالص در نرخ کمیسیون را برگشت می‌دهد. اشاره‌گر تابع **print** به نسخه **CommissionEmployee** تابع اشاره دارد که نوع کارمند، نام، شماره تامین اجتماعی، نرخ کمیسیون و فروش ناخالص را چاپ می‌کند. همانند کلاس **HourlyEmployee** هر دو تابع مبادرت به override کردن توابع در کلاس **Employee** می‌کنند.

اشاره‌گر تابع **earnings** در vtable برای کلاس **BasePlusCommissionEmployee** به تابع **earnings** این کلاس اشاره می‌کند که حاصل حقوق پایه به همراه فروش ناخالص ضرب شده در نرخ کمیسیون را برگشت می‌دهد. اشاره‌گر تابع **print** به نسخه **BasePlusCommissionEmployee** تابع اشاره دارد که نوع این کارمند، نام، شماره تامین اجتماعی، نرخ کمیسیون و فروش ناخالص را چاپ می‌کند. هر دو تابع مبادرت به override توابع در کلاس **CommissionEmployee** می‌کنند.

شکل ۲۴-۱۳ | نحوه عملکرد فراخوانی تابع virtual.

اگر به مبحث آموزشی **Empolyee** توجه کنید متوجه می‌شوید که هر کلاس غیرانتزاعی دارای پیاده‌سازی متعلق بخود برای توابع **virtual** بنام‌های **print** و **earnings** است. تا بدین جا آموخته‌اید که هر کلاسی که مستقیماً از کلاس مبنای انتزاعی **Employee** ارث‌بری دارد، بایستی تابع **earnings** را به نحوی پیاده کند که یک کلاس غیرانتزاعی گردد، چرا که **earnings** یک تابع **virtual** محض می‌باشد. این کلاس‌ها نیازی ندارند تا تابع **print** را پیاده‌سازی کنند، با این همه، با توجه به غیرانتزاعی بودن آنها، تابع یک تابع

www.yourshop.4kia.ir

virtual محض نیست و کلاس‌های مشتق شده می‌توانند پیاده‌سازی **print** در کلاس **Employee** را به ارث ببرند. از این گذشته، **BasePlusCommissionEmployee** مجبور نیست تا تابع **print** یا **earnings** را پیاده‌سازی کند، پیاده‌سازی هر دو تابع را می‌تواند از کلاس **CommissionEmployee** به ارث ببرد. اگر کلاسی در سلسله مراتب ما از پیاده‌سازی توابع به این روش ارث‌بری داشته باشد، اشاره‌گرهای vtable برای این توابع می‌توانند بسادگی به پیاده‌سازی تابعی اشاره داشته باشند که به ارث برده شده است. برای مثال، اگر **BasePlusCommissionEmployee** مبادرت به override کردن تابع **earnings** نکند، اشاره‌گر تابع **earnings** در vtable کلاس **BasePlusCommissionEmployee** به همان تابع **earnings** که در vtable کلاس **CommissionEmployee** به آن اشاره می‌کند، اشاره خواهد کرد.

چند ریختی از طریق یک ساختمان داده کارا و با سه سطح از اشاره‌گر انجام می‌شود. در ارتباط با یک سطح صحبت می‌کنیم، اشاره‌گرهای تابع در vtable. اینها به توابع واقعی اشاره دارند که به هنگام احضار یک تابع **virtual** اجرا می‌شوند.

حال به دومین سطح از اشاره‌گرها می‌پردازیم. هنگامی که یک شی از یک کلاس با یک یا چند تابع **virtual** معرفی می‌شود، کامپایلر مبادرت به الصاق یک اشاره‌گر به شی از جدول vtable برای آن کلاس می‌کند. معمولاً این اشاره‌گر جلوتر از شی قرار می‌گیرد، اما برای پیاده‌سازی به این روش ضرورتی ندارد. در شکل ۲۴-۱۳ این اشاره‌گرها مرتبط با شی‌های ایجاد شده در شکل ۲۳-۱۳ هستند (یک شی برای هر نوع **SalariedEmployee**، **HourlyEmployee**، **CommissionEmployee** و **BasePlusCommissionEmployee**).

دقت کنید که دیاگرام، مقادیر هر عضو داده شی را به نمایش در آورده است. برای مثال، شی **SalariedEmployee** حاوی یک اشاره‌گر به vtable این کلاس بوده، همچنین این شی حاوی مقادیر John Smith، 111-11-1111 و $800.00 است.

سطح سوم اشاره‌گرها فقط حاوی دستگیره‌ها به شی‌های است که فراخوانی تابع **virtual** را دریافت می‌کنند. دستگیره‌ها در این سطح می‌توانند مراجعه باشند. دقت کنید که در شکل ۲۴-۱۳ بردار **employess** رسم شده حاوی اشاره‌گرهای **Employee** می‌باشد. حال اجازه دهید تا به بررسی نحوه اجرای یک تابع **virtual** بپردازیم. به فراخوانی **baseClassPtr->print()** در تابع **virtualViaPointer** در خط 85 از شکل ۲۳-۱۳ توجه کنید. فرض کنید که **baseClassPtr** حاوی **employees[1]** است (یعنی آدرس شی **hourlyEmployee** در **employees**). زمانیکه کامپایلر این عبارت را کامپایل می‌کند، تعیین می‌کند که براستی فراخوانی صورت گرفته از طریق اشاره‌گر کلاس مبنا بوده و اینکه **print** یک تابع **virtual** است.

www.yourshop.4kia.ir

کامپایلر تعیین می‌کند که **print**، دومین ورودی یا چیز ثبت شده در vtable است. برای تعیین محل این ورودی، کامپایلر متوجه می‌شود که نیاز دارد تا از ورودی اول پرش کند. بنابر این، کامپایلر مبادرت به کامپایل یک افست یا جابجایی به میزان چهار بایت (چهار بایت برای هر اشاره‌گر بر روی اکثر کامپیوترهای 32 بیتی) در جدول اشاره‌گرهای کد شی زبان ماشین می‌کند تا کدی که فراخوانی تابع **virtual** را اجرا خواهد کرد بدست آید.

کد تولیدی توسط کامپایلر عملیات‌های زیر را انجام می‌دهد [نکته: اعداد بکار رفته در لیست متناظر با اعداد موجود در دوایر شکل ۲۴-۱۳ هستند].

۱- انتخاب *i*th ورودی از **employees** (در این مورد آدرس شی **hourlyEmployee**)، و ارسال آن بعنوان یک آرگومان به تابع **virtualViaPointer**. این کار سبب تنظیم پارامتر **baseClassPtr** برای اشاره به **hourlyEmployee** می‌شود.

۲- دستیابی به اطلاعات از طریق آدرس موجود در اشاره‌گر برای بدست آوردن شی **hourlyEmployee**.

۳- دستیابی به اطلاعات اشاره‌گر **hourlyEmployee** در جدول vtable برای رسیدن به vtable **HourlyEmployee**.

۴- جابجایی به میزان چهار بایت برای انتخاب اشاره‌گر تابع **print**.

۵- دستیابی به اطلاعات آدرس اشاره‌گر تابع **print** بفرم نام تابع اصلی که اجرا خواهد شد و استفاده از عملگر فراخوانی تابع ( ) برای اجرای تابع **print** مقتضی که در این مورد چاپ نوع کارمند، نام، شماره تامین اجتماعی، دستمزد ساعتی و ساعت کارکرد در هفته است. امکان دارد ساختمان داده بکار رفته در شکل ۲۴-۱۳ کمی پیچیده بنظر برسد، اما این پیچیدگی توسط کامپایلر مدیریت شده و از دید شما که سرگرم برنامه‌نویسی چندریختی هستید، پنهان است. عملیات دستیابی به اطلاعات از طریق آدرس موجود در اشاره‌گر و دسترسی به حافظه که در هر فراخوانی تابع **virtual** صورت می‌گیرد، مستلزم صرف زمان اضافی در اجرا است. vtable و اشاره‌گرهای vtable که به شی‌ها افزوده می‌شوند هم نیازمند حافظه هستند. با توجه به این موارد می‌توانید مشخص نمائید که آیا استفاده از توابع **virtual** در برنامه‌ها به نفع شما هست یا خیر.

۸-۱۳ مبحث آموزشی: سیستم پرداخت حقوق با استفاده از چند ریختی و اطلاعات نوع زمان اجرا با تبدیل dynamic_cast, typeid و type_info

به صورت مسئله مطرح شده در ابتدای بخش ۶-۱۳ مجدداً توجه کنید که در آن برای یک دوره پرداخت حقوق، شرکت تصمیم گرفته به حقوق پایه کارمندان نوع **BasePlusCommissionEmployee**، ده درصد اضافه کند. در زمان پردازش شی‌های **Employee** به روش چند ریختی در بخش ۶-۶-۱۳، نگران حالت

www.yourshop.4kia.ir

خاص نبودیم. با این وجود، هم اکنون برای تعدیل کردن حقوق پایه کارمندان **BasePlusCommissionEmployee**، مجبور هستیم تا نوع خاص هر شی **Employee** را در زمان اجرا مشخص کرده، سپس بر اساس آن عمل کنیم. در این بخش به بررسی قابلیت قدرتمندی بنام اطلاعات نوع زمان اجرا یا RTTI[1] و تبدیل دینامیکی خواهیم پرداخت که به برنامه امکان می‌دهند تا نوع یک شی را در زمان اجرا مشخص کرده و مطابق آن شی عمل شود.

برخی از کامپایلرها همانند Microsoft Visual C++ .NET، مستلزم این هستندکه RTTI قبل از اینکه بتواند در برنامه بکار گرفته شود، فعال گردد. می‌توانید با مراجعه به مستندات کامپایلر خود، با نحوه انجام اینکار آشنا شوید. برای فعال کردن RTTI در Visual C++ .NET، از منوی **Project** گزینه خصوصیات یا **Properties** را برای پروژه جاری انتخاب کنید. در کادر تبادلی **Property Pages**، گزینه **Configuration Properties>C/C++>Language** را انتخاب نمائید سپس از جعبه کامبو که در کنار **Enable Run-Time Type Info** قرار گرفته گزینه **Yes(/GR)** را انتخاب کرده و در پایان بر روی دکمه Ok کلیک کنید تا تغیرات صورت گرفته ذخیره شود.

در برنامه شکل ۲۵-۱۳ از سلسله مراتب **Employee** که در بخش ۶-۱۳ توسعه یافته استفاده کرده و ده درصد به حقوق پایه هر **BasePlusCommissionEmployee** اضافه می‌کنیم.

```
// Fig. 13.25: fig13_25.cpp
// Demonstrating downcasting and run-time type information.
// NOTE: For this example to run in Visual C++ .NET,
// you need to enable RTTI (Run-Time Type Info) for the project.
#include <iostream>
using std::cout;
using std::endl;
using std::fixed;

#include <iomanip>
using std::setprecision;

#include <vector>
using std::vector;

#include <typeinfo>

// include definitions of classes in Employee hierarchy
#include "Employee.h"
#include "SalariedEmployee.h"
#include "HourlyEmployee.h"
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"

int main()
{
   // set floating-point output formatting
   cout << fixed << setprecision( 2 );

   // create vector of four base-class pointers
   vector < Employee * > employees( 4 );

```

[1] - Run-Time Type Information

www.yourshop.4kia.ir

```
   // initialize vector with various kinds of Employees
   employees[ 0 ] = new SalariedEmployee(
      "John", "Smith", "111-11-1111", 800 );
   employees[ 1 ] = new HourlyEmployee(
      "Karen", "Price", "222-22-2222", 16.75, 40 );
   employees[ 2 ] = new CommissionEmployee(
      "Sue", "Jones", "333-33-3333", 10000, .06 );
   employees[ 3 ] = new BasePlusCommissionEmployee(
      "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );

   // polymorphically process each element in vector employees
   for ( size_t i = 0; i < employees.size(); i++ )
   {
      employees[ i ]->print(); // output employee information
      cout << endl;

      // downcast pointer
      BasePlusCommissionEmployee *derivedPtr =
         dynamic_cast < BasePlusCommissionEmployee * >
            ( employees[ i ] );

      // determine whether element points to base-salaried
      // commission employee
      if ( derivedPtr != 0 ) // 0 if not a BasePlusCommissionEmployee
      {
         double oldBaseSalary = derivedPtr->getBaseSalary();
         cout << "old base salary: $" << oldBaseSalary << endl;
         derivedPtr->setBaseSalary( 1.10 * oldBaseSalary );
         cout << "new base salary with 10% increase is: $"
            << derivedPtr->getBaseSalary() << endl;
      } // end if

      cout << "earned $" << employees[ i ]->earnings() << "\n\n";
   } // end for

   // release objects pointed to by vector's elements
   for ( size_t j = 0; j < employees.size(); j++ )
   {
      // output class name
      cout << "deleting object of "
         << typeid( *employees[ j ] ).name() << endl;

      delete employees[ j ];
   } // end for

   return 0;
} // end main
```

```
salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary:300.00
old base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

deleting object of class SalariedEmployee
```

www.yourshop.4kia.ir

```
deleting object of class HourlyEmployee
deleting object of class CommissionEmployee
deleting object of class BasePlusCommissionEmployee
```

شکل ۲۵-۱۳ | توضیح روش تبدیل و اطلاعات نوع زمان اجرا.

خط 31 چهار عنصر برداری **employees** اعلان کرده که اشاره‌گرهایی به شی‌های **Employee** را ذخیره می‌سازد. خطوط 41-34 بردار را با آدرس‌های اخد شده دینامیکی شی‌ها از کلاس **SalariedEmployee**(شکل‌های ۱۵-۱۳و۱۶-۱۳)، **HourlyEmployee**(شکل‌های ۱۷-۱۳و۱۸-۱۳)، **CommissionEmployee** (شکل‌های ۱۹-۱۳و ۲۰-۱۳)و **BasePlusCommissionEmployee**(شکل‌های ۲۱-۱۳ و ۲۲-۱۳) پر می‌کند.

عبارت **for** در خطوط 66-44 در میان بردار **employees** حرکت کرده و اطلاعات هر کارمند را با احضار تابع عضو **print** (خط 46) به نمایش در می‌آورد. بخاطر دارید که چون **print** بصورت **virtual** در کلاس مبنا **Employee** اعلان شده، سیستم تابع **print** مناسب با شی کلاس مشتق شده را فراخوانی می‌کند.

در این مثال با شی‌های **BasePlusCommissionEmployee** مواجه شده‌ایم و می‌خواهیم به حقوق پایه آنها ده درصد اضافه کنیم. از آنجائی که پردازش کارمندان را بصورت چند ریختی انجام داده‌ایم، نمی‌توانیم (با تکنیکی که آموخته‌ایم) بطور مشخص نوع **Employee** را برای کار در هر زمان تعیین کنیم. این حالت مشکل‌ساز می‌شود، چرا که کارمندان **BasePlusCommissionEmployee** بایستی در هنگام مواجه شدن با آنها تشخیص داده شوند تا بتوان ده درصد به حقوق آنها اضافه کرد. برای انجام اینکار، از عملگر **dynamic_cast** در خط 51 برای تعیین اینکه آیا نوع شی یک **BasePlusCommissionEmployee** است یا خیر، استفاده کرده‌ایم. در بخش ۳-۳-۱۳ از این نوع تبدیل یاد شده است. خطوط 52-50 بصورت دینامیکی **employees[1]** را از نوع **Employee *** به نوع **BasePlusCommissionEmployee *** تبدیل می‌کند. اگر عنصر **vector** به شیی اشاره کند که یک شی **BasePlusCommissionEmployee** است، پس آدرس آن شی به **commissionPtr** تخصیص داده می‌شود، در غیر اینصورت، صفر به اشاره‌گر **derivedPtr** کلاس مشتق شده تخصیص داده خواهد شد.

اگر مقدار برگشتی توسط عملگر **dynamic_cast** در خطوط 52-50 برابر صفر نباشد، شی از نوع صحیح بوده و عبارت **if** (خطوط 63-56) پردازش خاصی که مورد نیاز شی **BasePlusCommissionEmployee** است انجام می‌دهد. خطوط 58، 60 و 62 توابع **getBaseSalary** و **setBaseSalary** را برای بازیابی و به روز کردن حقوق کارمند فراخوانی می‌کنند.

www.yourshop.4kia.ir

خط 65 تابع عضو **earnings** را بر روی شی که **employees[1]** به آن اشاره می‌کند، فراخوانی می‌نماید. بخاطر دارید که **earnings** بصورت **virtual** در کلاس مبنا اعلان شده است، از اینرو برنامه تابع **earnings** شی از کلاس مشتق شده را فراخوانی می‌کند. اینحالت نمونه‌ای از مقیدسازی دینامیکی است.

حلقه **for** در خطوط 76-69 نوع هر کارمند را نشان داده و از عملگر **delete** برای آزادسازی یا بازپس‌گیری حافظه دینامیکی که هر عنصر **vector** به آن اشاره می‌کند، استفاده کرده است. عملگر **typeid** در خط 73 یک مراجعه به یک شی از کلاس **type_info** برگشت می‌دهد که حاوی اطلاعاتی در ارتباط با نوع عملوند آن است که شامل نام آن نوع می‌باشد. زمانیکه فراخوانی می‌شود، تابع عضو **type_info** بنام **name** (خط 73) یک رشته مبتنی بر اشاره‌گر برگشت می‌دهد که حاوی نام نوع (مثلاً "*class BasePlusCommissionEmployee*") از آرگومان ارسالی به **typied** است. برای استفاده از **typeid**، برنامه باید دارای فایل سرآیند <**typeinfo**> باشد (خط 6).

توجه کنید با تبدیل اشاره‌گر **Employee** به یک اشاره‌گر **BasePlusCommissionEmployee** (خطوط 52-50) از وقوع چند خطای کامپایل جلوگیری کرده‌ایم. اگر **dynamic_cast** را از خط 51 حذف کرده و مبادرت به تخصیص مستقیم اشاره‌گر جاری **Employee** به اشاره‌گر **commissionPtr** کلاس **BasePlusCommissionEmployee** کنیم، با خطای کامپایلر مواجه خواهیم شد. C++ به برنامه اجازه نمی‌دهد تا اشاره‌گر کلاس مبنا را به اشاره‌گر کلاس مشتق شده تخصیص دهید چرا که در اینحالت رابطه *is-a* نقض می‌شود، **CommissionEmployee** یک **BasePlusCommissionEmployee** نیست. رابطه is-a فقط مابین کلاس مشتق شده و کلاس‌های مبنای آن صادق است و عکس آن برقرار نمی‌باشد.

۹-۱۳ نابود کننده‌های virtual

به هنگام استفاده از روش چند ریختی در پردازش شی‌های اخذ شده دینامیکی از سلسله مراتب، احتمال رخ دادن مشکل وجود دارد. تا بدین جا شاهد نابود کننده‌های غیر virtual بودید، نابود کننده‌های که با کلمه کلیدی virtual اعلان نشده‌اند. اگر یک شی از کلاس مشتق شده با یک نابود کننده غیر virtual بطور صریح نابود شود، با اعمال عملگر delete بر روی اشاره‌گر کلاس مبنا، اینحالت در C++ استاندارد تعریف نشده است.

ساده‌ترین راه حل این مشکل ایجاد یک نابود کننده virtual (یعنی نابود کننده‌ای که با کلمه کلیدی virtual اعلان شده است) در کلاس مبنا است. با اینکار تمام نابود کننده‌های کلاس مشتق شده virtual خواهند شد حتی اگر نام یکسان با نابود کننده کلاس مبنا نداشته باشند. حال اگر یک شی در سلسله مراتب بصورت صریح و با استفاده از عملگر delete بر روی اشاره‌گر کلاس مبنا نابود شود، نابود کننده مقتضی کلاس براساس شی که اشاره‌گر کلاس مبنا به آن اشاره می‌کند، فراخوانی خواهد شد. بخاطر داشته باشید

www.yourshop.4kia.ir

که به هنگام نابود شدن یک شی از کلاس مشتق شده، بخشی از کلاس مبنا از کلاس مشتق شده نیز از بین می‌رود، از اینرو اجرای هر دو نابود کننده کلاس مشتق شده و مبنا از اهمیت برخوردار است. نابود کننده کلاس مبنا بصورت اتوماتیک پس از اجرای نابود کننده کلاس مشتق شده، اجرا می‌گردد.

خطای برنامه‌نویسی

سازنده‌ها نمی‌توانند **virtual** *باشند. اعلان سازنده بصورت* **virtual** *خطای کامپایل بدنبال خواهد داشت.*

۱۰-۱۳ مبحث آموزشی مهندسی نرم‌افزار: ارث‌بری در سیستم ATM

در این بخش مجدداً به سراغ طراحی سیستم ATM می‌رویم تا ببینیم چگونه می‌توان از مزایای ارث‌بری در این سیستم استفاده کرد. برای اعمال توارث ابتدا، نگاهی به نقاط مشترک مابین کلاس‌ها در سیستم می‌اندازیم. یک سلسله مراتب توارث برای مدل کردن کلاس‌ها در فرآیند چندریختی ایجاد می‌کنیم.

سپس دیاگرام کلاس را برای پیوستن روابط ارث‌بری جدید اصلاح می‌کنیم. در پایان، به بررسی نحوه به روز کردن طراحی خود در تبدیل به فایلهای سرآیند C++ می‌پردازیم.

در بخش ۱۱-۳، با مشکلی مواجه شدیم که در ارتباط با ارائه تراکنش مالی در سیستم بود. بجای ایجاد یک کلاس برای ارائه کلیه تعاملات صورت گرفته، تصمیم گرفتیم تا سه کلاس تراکنشی مجزا بنام‌های **BalanceInquiry**، **Withdrawal** و **Deposit** ایجاد کنیم تا نشاندهنده تعامل‌های باشند که سیستم **ATM** می‌تواند انجام دهد. شکل ۲۶-۱۳ نشاندهنده صفات و عملیات این کلاس‌ها است. توجه کنید که آنها در یک صفت **(accountNumber)** و یک عملیات **(execute)** مشترک هستند. هر کلاسی مستلزم صفت **accountNumber** برای مشخص کردن شماره حساب است تا تراکنش بر آن اعمال شود. هر کلاسی حاوی عملیات **execute** است که **ATM** با فراخوانی آن تراکنش را انجام می‌دهد. واضح است که **BalanceInquiry**، **Withdrawal** و **Deposit** ارائه کننده انواع تراکنش‌ها هستند. شکل ۲۶-۱۳ نقاط مشترک در میان تراکنش کلاس‌ها را آشکار کرده است، از اینرو بنظر می‌رسد استفاده از توارث برای فاکتورگیری از ویژگیهای مشترک، در طراحی این کلاس امکان‌پذیر باشد.

از اینرو عامل مشترک را در کلاس مبنا **Transaction** قرار داده و کلاس **BalanceInquiry**، **Withdrawal** و **Deposit** را از **Transaction** مشتق می کنیم (شکل ۲۷-۱۳).

شکل ۲۶-۱۳ | صفات و عملیات کلاس‌های BalanceInquiry، Withdrawal و Deposit.

UML تصریح کننده یک رابطه بنام *تعمیم* یا *generalization* برای مدل کردن توارث است. شکل ۲۷-۱۳ دیاگرام کلاسی است که رابطه توارث مابین کلاس مبنا **Transactiion** و سه کلاس مشتق شده از آنرا را مدل کرده است. فلش‌های با مثلث‌های توخالی بر این نکته دلالت دارند که کلاس‌های **BalanceInquiry**، **Withdrawal** و **Deposit** از کلاس **Transaction** مشتق شده‌اند. گفته می‌شود

www.yourshop.4kia.ir

کلاس **Transactiion** به کلاس‌های مشتق شده خودش تعمیم یافته است. گفته می‌شود کلاس‌های مشتق شده ویژه از کلاس **Transaction** هستند.

کلاس‌های **BalanceInquiry**، **Withdrawal** و **Deposit** صفت **accountNumber** از نوع صحیح را به اشتراک می‌گذارند، بنابر این از این صفت مشترک فاکتور گرفته و آنرا در کلاس مبنا **Transaction** جای می‌دهیم. دیگر، در بخش دوم کلاس‌های مشتق شده، **accountNumber** دیده نمی‌شود، چرا که هر سه کلاس مشتق شده این صفت را از **Transaction** ارث می‌برند.

با این همه، بخاطر داشته باشید که کلاس‌های مشتق شده نمی‌توانند به صفات **private** یک کلاس مبنا دسترسی داشته باشند. بنابر این تابع عضو سراسری **getAccountNumber** را در کلاس **Transaction** وارد کرده‌ایم. هر کلاس مشتق شده این تابع عضو را به ارث برده، و می‌تواند به **accountNumber** خود در صورت نیاز به اجرای یک تراکنش دسترسی پیدا کند.

مطابق شکل ۲۶-۱۳، کلاس‌های **BalanceInquiry**، **Withdrawal** و **Deposit** عملیات **execute** را هم به اشتراک گذاشته‌اند، از اینرو کلاس مبنا **Transaction** باید حاوی تابع عضو سراسری **execute** باشد. با این همه، پیاده‌سازی مشترک **execute** در کلاس **Transaction** وجود ندارد چرا که عملکرد این تابع عضو بستگی به نوع خاصی از تراکنش دارد. بنابر این تابع عضو **execute** بعنوان یک تابع **virtual** محض در کلاس مبنا **Transaction** اعلان شده است. با اینکار **Transaction** یک کلاس انتزاعی شده و هر کلاس مشتق شده از **Transaction** بایستی یک کلاس غیرانتزاعی باشد. UML مستلزم این است که اسامی کلاس انتزاعی (و توابع **virtual** محض–عملیات انتزاعی در UML) بصورت ایتالیک نشان داده شوند، از اینرو **Transaction** و تابع عضو آن **execute** بصورت ایتالیک در شکل ۲۷-۱۳ نشان داده شده‌اند. توجه کنید که عملیات **execute** در کلاس‌های مشتق شده **BalanceInquiry**، **Withdrawal** و **Deposit** بصورت ایتالیک نوشته نشده است. هر کلاس مشتق شده مبادرت به override کردن تابع عضو **execurte** کلاس **Transaction** با پیاده‌سازی مقتضی می‌کند. دقت کنید که در شکل ۲۷-۱۳ عملیات **execute** در بخش سوم کلاس‌های مشتق شده جای گرفته است، به این دلیل که هر کلاس دارای یک پیاده‌سازی غیرانتزاعی متفاوت از تابع عضو override شده است.

شکل ۲۷-۱۳ |دیاگرام کلاس مدل کننده رابطه تعمیم مابین کلاس مبنا Transaction و کلاس‌های مشتق شده.

همانطوری که در این فصل آموختید، یک کلاس مشتق شده می‌تواند واسط یا پیاده‌سازی را از کلاس مبنا به ارث ببرد. در مقایسه سلسله مراتب طراحی شده برای پیاده‌سازی توارث، یکی از طرح‌های ارث‌بری واسط متمایل به داشتن عاملیت خود در مراتب پایین سلسله مراتب است. کلاس مبنا دلالت بر یک یا چند

www.yourshop.4kia.ir

تابع دارد که بایستی توسط هر کلاس در سلسله مراتب تعریف شوند، اما کلاس‌های مشتق شده مجزا از هم پیاده‌سازی متعلق به خود را در ارتباط با توابع تدارک می‌بینند.

طراحی سلسله مراتب توارث صورت گرفته برای سیستم ATM از مزایای این نوع از ارث‌بری استفاده می‌کند و برای ATM روش مناسبی برای اجرای تمام تراکنش‌ها فراهم می‌آورد. هر کلاس مشتق شده از **Transaction** برخی از جزئیات پیاده‌سازی را به ارث می‌برد (مانند عضو داده **accountNumber**)، اما مزیت اصلی همکاری ارث‌بری در سیستم این است که کلاس‌های مشتق شده یک واسط مشترک را به اشتراک می‌گذارند (برای مثال، تابع عضو **execute** که **virtual** است). سیستم ATM می‌تواند اشاره‌گر **Transaction** را بطرف هر تراکنشی هدایت کرده و زمانیکه ATM تابع **execute** را از طریق این اشاره‌گر احضار نماید، نسخه مناسب **execute** برای آن تراکنش بصورت اتوماتیک اجرا می‌شود. برای مثال فرض کنید کاربر گزینه مشاهده میزان موجودی را انتخاب کند ATM اشاره‌گر **Transaction** را بطرف یک شی جدید از کلاس BalanceInquiry هدایت می‌کند، که کامپایلر ++C اجازه آنرا می‌دهد، چرا که **BalanceInquiry** یک شی از **Transaction** می‌باشد (رابطه is-a). زمانیکه ATM از این اشاره‌گر برای احضار **execute** استفاده می‌کند، نسخه **execute** کلاس **BalanceInquiry** فراخوانی می‌گردد.

این روش چند ریختی می‌تواند گسترش و بسط‌پذیری سیستم را به آسانی میسر نماید. برای مثال ممکن است مایل به افزودن یک نوع تراکنش جدید باشیم (برای مثال انتقال وجه یا پرداخت صورتحساب)، کاری که باید انجام دهیم ایجاد یک کلاس مشتق شده از **Transaction** است که مبادرت به override کردن تابع عضو **execute** با نسخه مقتضی برای تراکنش جدید می‌کند. در اینحالت کد سیستم به کمترین تغییر نیاز خواهد داشت.

همانطوری که در ابتدای فصل آموختید، یک کلاس انتزاعی همانند **Transaction** کلاسی است که برنامه‌نویس هرگز قصد نمونه‌سازی از آنرا ندارد. یک کلاس انتزاعی فقط صفات و رفتار مشترک را برای کلاس‌های مشتق شده از خود را در سلسله مراتب توارث اعلان می‌کند. کلاس **Transaction** مفهومی از تراکنش رخ داده بر روی شماره حساب و اجرای تراکنش تعریف می‌کند. ممکن است تعجب کنید که چرا زحمت وارد کردن تابع عضو **execute** را که **virtual** محض می‌باشد به کلاس **Transaction** بخود داده‌ایم، در صورتیکه **execute** فاقد یک پیاده‌سازی غیرانتزاعی است. به لحاظ مفهومی ما این تابع عضو را وارد کرده‌ایم، به این دلیل که این تابع تعریف کننده رفتاری از تمام تراکنش‌ها یعنی اجرا شدن است. به لحاظ تکنیکی، بایستی تابع عضو **execute** را در کلاس مبنای **Transaction** قرار دهیم، برای اینکه

www.yourshop.4kia.ir

ATM (یا هر کلاس دیگری) بتواند بصورت چندریختی نسخه override شده این تابع را از طریق اشاره‌گر با مراجعه **Transaction** فراخوانی کند.

کلاس‌های مشتق شده **BalanceInquiry**، **Withdrawal** و **Deposit** صفت **accountNumber** را از کلاس مبنای **Transaction** به ارث می‌برند، اما کلاس‌های **Withdrawal** و **Deposit** حاوی صفت دیگری بنام **amount** هستند که وجه تمایز آنها از کلاس **BalanceInquiry** است. کلاس‌های **Widthdrawal** و **Deposit** نیازمند این صفت اضافی هستند تا بتوانند میزان پولی که کاربر می‌خواهد از حساب برداشت کند یا پس‌انداز نماید، ذخیره کنند. کلاس **BalanceInqiury** نیازی به این صفت ندارد و فقط نیازمند یک شماره حساب برای اجرا شدن دارد. با اینکه دو تا از سه کلاس مشتق شده **Transaction** این صفت را به اشتراک می‌گذارند، ما آنرا در کلاس مبنا **Transaction** جای نداده‌ایم. فقط ویژگیهای مشترک در میان تمام کلاس‌های مشتق شده را در کلاس مبنا جای می‌دهیم، از اینرو کلاس‌های مشتق شده صفات غیرضروری (و عملیات) را به ارث نمی‌برند.

شکل ۲۸-۱۳ تصویری از دیاگرام کلاس به روز شده از مدلی است که توارث در آن وارد و به معرفی کلاس **Transaction** پرداخته شده است. یک پیوستگی مابین کلاس **ATM** و کلاس **Transaction** را مدل کرده‌ایم تا نشان دهیم که در هر لحظه، تراکنشی در ATM اجرا می‌شود یا خیر (یعنی، صفر یا یک شی از نوع **Transaction** در یک زمان در سیستم وجود دارد).

شکل ۲۸-۱۳ | دیاگرام کلاس از سیستم ATM (پیوستگی توارث). دقت کنید که نام کلاس انتزاعی Transaction بصورت ایتالیک نوشته شده است.

بدلیل اینکه **Withdrawal** نوعی **Transaction** است، نیازی به ترسیم خط پیوستگی مستقیماً مابین کلاس **ATM** و کلاس **Withdrawal** نداریم. کلاس‌های مشتق شده **BalanceInquiry** و **Deposit** نیز این پیوستگی را به ارث می‌برند.

همچنین یک پیوستگی مابین کلاس **Transaction** و **BankDatabase** اضافه کرده‌ایم (شکل ۲۸-۱۳). تمام تراکنش‌ها نیازمند داشتن یک مراجعه به **BankDatabase** هستند. از اینرو می‌توانند به اطلاعات حساب دسترسی پیدا کنند. هر کلاس مشتق شده از **Transaction** این مراجعه را به ارث می‌برد و بنابر این پیوستگی موجود مابین کلاس **Withdrawal** و **BankDatabase** را مدل نکرده‌ایم.

یک پیوستگی مابین کلاس **Transaction** و **Screen** ایجاد کرده‌ایم به این دلیل که تمام تراکنش در خروجی و در دید کاربر از طریق **Screen** قرار می‌گیرند. هر کلاس مشتق شده‌ای این پیوستگی را به ارث می‌برد. کلاس **Withdrawal** هنوز هم سهیم در پیوستگی با **CashDispenser** و **Keypad** است، این پیوستگی‌ها بر روی کلاس مشتق شده از **Withdrawal** اعمال می‌شود.

www.yourshop.4kia.ir

دیاگرام کلاس به نمایش درآمده در شکل ۲۰-۹ نشان دهنده صفات و عملیات با نشانگر قابل رویت بودن است. اکنون دیاگرام اصلاح شده کلاس را در شکل ۲۹-۱۳ عرضه کرده‌ایم که شامل کلاس مبنای انتزاعی **Trancaction** می‌باشد. این دیاگرام مختصر شده، رابطه توارث را نشان می‌دهد (این روابط در شکل ۲۸-۱۳ آورده شده‌اند)، اما بجای آن صفات و عملیات‌های که پس از اعمال ارث‌بری به سیستم بدست آمده‌اند را در خود دارد. توجه کنید که نام کلاس انتزاعی **Transaction** و نام عملیات انتزاعی **execute** در کلاس **Transaction** بصورت ایتالیک نشان داده شده است.

پیاده‌سازی طرح سیستم ATM با توارث

در بخش ۱۲-۹ شروع به پیاده‌سازی طرح سیستم ATM با کد C++ کردیم. در این بخش سعی می‌کنیم با اصلاح آن پیاده‌سازی از توارث سود ببریم و از کلاس **Withdrawal** بعنوان یک مثال استفاده می‌کنیم.

۱- اگر کلاس **A** تعمیم دهنده کلاس **B** باشد، پس کلاس **B** از کلاس **A** مشتق شده است. برای مثال، کلاس مبنای انتزاعی **Transaction** تعمیم دهنده کلاس **Withdrawal** است. از اینرو، کلاس **Withdrawal** از کلاس **Transaction** مشتق شده است. شکل ۳۰-۱۳ حاوی بخشی از فایل سرآیند کلاس **Withdrawal** می‌باشد، که در آن تعریف کلاس نشاندهنده رابطه ارث‌بری مابین **Withdrawal** و **Transaction** است (خط 9)

۲- اگر کلاس **A** یک کلاس انتزاعی باشد و کلاس **B** از کلاس **A** مشتق شده باشد، اگر کلاس **B** یک کلاس غیرانتزاعی باشد، پس کلاس **B** باید توابع **virtual** محض کلاس **A** را پیاده‌سازی کند. برای مثال، کلاس **Transaction** حاوی تابع **execute** است که **virtual** محض می‌باشد، از اینرو کلاس **Widthdrawal** باید این تابع عضو را پیاده‌سازی کند، اگر مایل باشیم نمونه‌ای از شی **Withdrawal** ایجاد کنیم. شکل ۳۱-۱۳ حاوی فایل سرآیند C++ برای کلاس **Withdrawal** از شکل ۲۸-۱۳ و شکل ۲۹-۱۳ می‌باشد. کلاس **Withdrawal** عضو داده **accountNumber** را از کلاس مبنای **Transaction** به ارث می‌برد، از اینرو **Withdrawal** این عضو داده را اعلان نکرده است. همچنین کلاس **Withdrawal** مراجعه‌های به **Screen** و **BankDatabase** را از کلاس مبنای خود یعنی **Transcation** به ارث برده است، بنابر این نیازی نیست تا این مراجعه‌ها را در کد خودمان وارد کنیم. شکل ۲۹-۱۳ نشاندهنده صفت **amount** و عملیات **execute** متعلق به کلاس **Withdrawal** است. خط 19 از شکل ۳۱-۱۳ یک عضو داده برای صفت **amount** اعلان کرده است. خط 16 حاوی نمونه اولیه تابع برای عملیات **execute** است. بخاطر داشته باشید که برای غیرانتزاعی بودن یک کلاس، کلاس مشتق شده **Withdrawal** بایستی یک پیاده‌سازی غیرانتزاعی از تابع **execute** که **virtual** محض در کلاس مبنای

www.yourshop.4kia.ir

Transaction می‌باشد تدارک دیده باشد. نمونه اولیه در خط 16 هشدار می‌دهد که تابع **virtual** محض در کلاس مبنا را override کنید.

اگر بخواهید یک پیاده‌سازی در فایل cpp. داشته باشید، باید این نمونه اولیه را تدارک ببینید. مراجعه‌های **keypad** و **cashDispenser** (خطوط 21-20) اعضای داده مشتق شده از پیوستگی **Withdrawal** در شکل ۲۸-۱۳ هستند. برای اینکه بتوانیم مراجعه‌های موجود در خطوط 21-20 را کامپایل کنیم از اعلان‌های رو به جلو در خطوط 9-8 استفاده کرده‌ایم.

```
1  // Fig. 13.30: Withdrawal.h
2  // Withdrawal class definition. Represents a withdrawal transaction.
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  #include "Transaction.h" // Transaction class definition
7
8  //class withdrawal derives from base class Transaction
9  class Withdrawal : public Transaction
10 {
11 }; // end class Withdrawal
12
13 #endif // WITHDRAWAL_H
```

شکل ۳۰-۱۳ | تعریف کلاس Withdrawal که از Transaction مشتق شده است.

```
1  // Fig. 13.31: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction.
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  #include "Transaction.h" // Transaction class definition
7
8  class Keypad; // forward declaration of class Keypad
9  class CashDispenser; // forward declaration of class CashDispenser
10
11 //class withdrawal derives from base class Transaction
12 class Withdrawal : public Transaction
13 {
14 public:
15    //member function overriding execute in base class Transaction
16    virtual void execute(); // perform the transaction
17 private:
18    // attributes
19    double amount; // amount to withdraw
20    Keypad &keypad; // reference to ATM's keypad
21    CashDispenser &cashDispenser; // reference to ATM's cash dispenser
22 }; // end class Withdrawal
23
24 #endif // WITHDRAWAL_H
```

شکل ۳۱-۱۳ | فایل سرآیند کلاس Withdrawal مبتنی بر شکل‌های ۲۸-۱۳ و ۲۹-۱۳.

خودآزمایی مبحث مهندسی نرم‌افزار

۱-۱۳ زبان UML از یک فلش با..... برای نشان دادن رابطه تعمیم استفاده می‌کند.

a) فلش توپر.

b) فلش مثلثی توخالی.

c) فلش توخالی لوزی شکل.

d) فلش یکپارچه.

www.yourshop.4kia.ir

۱۳-۲ آیا عبارت زیر صحیح است یا اشتباه در زبان UML در زیر اسامی کلاس انتزاعی و عملیات یک خط قرار داده می‌شود.

۱۳-۳ یک فایل سرآیند C++ بنویسید تا پیاده‌سازی از طرح کلاس Transaction مشخص شده در شکل ۱۳-۲۸ و ۱۳-۲۹ باشد. مطمئن شوید تا مراجعه‌های private بر پایه پیوستگی کامل Transaction در نظر گرفته شده باشند. همچنین از توابع سراسری get برای اعضای داده private استفاده کنید تا کلاس‌های مشتق شده بتوانند وظایف خود را انجام دهند.

پاسخ خودآزمایی مبحث آموزشی مهندسی نرم‌افزار

۱۳-۱ b.

۱۳-۲ اشتباه. در UML اسامی و عملیات کلاس انتزاعی بصورت ایتالیک نشان داده می‌شود.

۱۳-۳ حاصل طراحی کلاس Transaction در فایل سرآیند شکل ۱۳-۲۲ آورده شده است.

```
// Fig. 13.32: Transaction.h
// Transaction abstract base class definition.
#ifndef TRANSACTION_H
#define TRANSACTION_H

class Screen; // forward declaration of class Screen
class BankDatabase; // forward declaration of class BankDatabase

class Transaction
{
public:
   int getAccountNumber() // return account number
   Screen &getScreen() // return reference to screen
   BankDatabase &getBankDatabase() // return reference to database

   // pure virtual function to perform the transaction
   virtual void execute() = 0; // overridden in derived classes
private:
   int accountNumber; // indicates account involved
   Screen &screen; // reference to the screen of the ATM
   BankDatabase &bankDatabase; // reference to the account info database
}; // end class Transaction

#endif // TRANSACTION_H
```

شکل ۱۳-۳۲ | فایل سرآیند کلاس Transaction بر پایه شکل‌های ۱۳-۲۸ و ۱۳-۲۹.

www.yourshop.4kia.ir