فصل دهم

کلاس‌ها: نگاهی عمیق‌تر: بخش II

اهداف

در این فصل با مطالب زیر آشنا خواهید شد:

- **مشخص کردن شی‌های ثابت (const) و توابع عضو ثابت.**
- **ایجاد شی‌های مرکب از سایر شی‌ها.**
- **استفاده از توابع و کلاس‌های friend.**
- **استفاده از اشاره‌گر this.**
- **ایجاد و نابود کردن شی‌های دینامیکی با عملگر new و delete.**
- **استفاده از اعضای داده static و توابع عضو.**
- **نکاتی در ارتباط با کلاس‌های تکرار شونده که در میان عناصر کلاس‌های حامل حرکت می‌کنند.**
- **استفاده از کلاس‌های پروکسی برای پنهان نگه‌داشتن جزئیات پیاده‌سازی از دید کلاس‌های سرویس‌گیرنده.**

رئوس مطالب

۱-۱۰ مقدمه

۱۰-۲ شی‌های ثابت (const) و توابع عضو ثابت
۱۰-۳ ترکیب: شی‌ها بعنوان اعضای کلاس
۱۰-٤ توابع و کلاس‌های friend
۱۰-۵ استفاده از اشاره‌گر this
۱۰-٦ مدیریت دینامیکی حافظه با عملگرهای new و delete
۱۰-۷ اعضای static کلاس
۱۰-۸ داده انتزاع و پنهان سازی اطلاعات
۱۰-۸-۱ مثال: نوع داده انتزاعی آرایه
۱۰-۸-۲ مثال: نوع داده انتزاعی رشته
۱۰-۸-۳ مثال: نوع داده انتزاعی صف
۱۰-۹ کلاس‌های حامل و تکرارشونده‌ها
۱۰-۱۰ کلاس‌های پروکسی

۱۰-۱ مقدمه

در این فصل به آموزش کلاس‌ها و داده‌های انتزاعی به همراه چندین مبحث پیشرفته ادامه می‌دهیم. از شی‌ها و توابع عضو **const** برای جلوگیری کردن از تغییر شی‌ها و حفظ حداقل مجوزهای دسترسی استفاده خواهیم کرد. در مورد ترکیب صحبت می‌کنیم که فرمی از استفاده مجدد است که در آن کلاسی می‌تواند دارای شی‌های از سایر کلاس‌ها بعنوان اعضا باشد. سپس به معرفی مبحث دوستی (*friendshisp*) می‌پردازیم، که به طراح کلاس امکان می‌دهد تا توابع غیرعضوی را که می‌توانند به اعضای غیرسراسری (*public*) کلاس دسترسی پیدا کنند را معین نماید. تکنیکی که غالباً در سربارگذاری عملگر بکار گرفته می‌شود (فصل یازدهم). در مورد یک اشاره‌گر خاص بنام **this** صحبت می‌کنیم که یک آرگومان ضمنی برای هر تابع عضو غیراستاتیک کلاس است که به این توابع عضو اجازه دسترسی صحیح به اعضاء داده شی و سایر توابع عضو غیراستاتیکی را فراهم می‌آورد. سپس در ارتباط با مدیریت حافظه دینامیکی صحبت می‌کنیم و نشان می‌دهیم که چگونه با استفاده از عملگرهای **new** و **delete** می‌توان شی‌های دینامیکی را ایجاد و نابود کرد. سپس به بررسی اعضای کلاس استاتیک و نحوه استفاده از اعضای داده استاتیک و توابع عضو در کلاس‌های متعلق بخودمان می‌پردازیم. در پایان، با نحوه ایجاد یک کلاس پروکسی برای پنهان کردن جزئیات پیاده‌سازی یک کلاس (شامل اعضای داده **private** آن) از دید سرویس‌گیرنده‌های کلاس آشنا خواهید شد.

در فصل سوم به معرفی کلاس استاندارد **string** پرداختیم. با این وجود، در این فصل از رشته‌های مبتنی بر اشاره‌گر استفاده خواهیم‌کرد که در فصل هشتم معرفی شده است تا کسانی‌که مایل هستند تا خود را آماده کارهای حرفه‌ای نمایند، از آن کمک بگیرند.

۲-۱۰ شی‌های ثابت (const) و توابع عضو ثابت

یکی از قواعد بنیادین در مهندسی نرم‌افزار، حفظ حداقل مجوزها و پایبندی به آنها است. اجازه دهید تا به بررسی این قواعد در ارتباط با شی‌ها بپردازیم. برخی از شی‌ها نیاز به اصلاح و تغییر دارند و تعدادی هم ندارند. برنامه‌نویس می‌تواند با استفاده از کلمه کلیدی **const** مشخص کند که یک شی تغییرپذیر نبوده و هر عملی که منجر به تغییر آن شی شود با خطای کامپایل مواجه می‌شود. عبارت

```
const Time noon (12,0,0);
```

شی **noon** از کلاس **Time** را بصورت ثابت (**const**) اعلان و با 12 ظهر مقداردهی اولیه کرده است.

مهندسی نرم‌افزار

اعلان یک شی بعنوان ثابت، سبب حفظ حداقل مجوزها یا امتیازها می‌شود.

کارائی

اعلان متغیرها و شی‌ها بصورت ثابت می‌تواند در افزایش کارایی نقش داشته باشد.

امروزه برخی از کامپایلرهای پیشرفته قادر به انجام بهینه‌سازی‌های مشخص بر روی ثابت‌ها هستند که نمی‌توان بر روی متغیرها اعمال کرد. کامپایلرهای C++ اجازه فراخوانی تابع عضو برای شی‌های ثابت را نمی‌دهند مگر اینکه خود توابع عضو بصورت ثابت اعلان شده باشند. این امر حتی در مورد توابع عضو *get* هم که شی را دچار تغییر نمی‌سازند صادق است. علاوه بر این، کامپایلر به توابع عضو اعلان شده بصورت **const** اجازه تغییر در شی را نمی‌دهد.

تابعی بصورت ثابت اعلان می‌شود که هم در نمونه اولیه خود (شکل۱-۱۰، خطوط 24-19) و هم در تعریف خود (شکل۲-۱۰، خطوط 47، 53، 59 و 65) با اعمال کلمه‌کلیدی **const** پس از لیست پارامتری تابع و قبل از براکت چپ که شروع بدنه تابع است (در مورد تعریف تابع) مشخص شده باشد.

خطای برنامه‌نویسی

تعریف یک تابع عضو ثابت که اقدام به تغییر در عضو داده یک شی می‌نماید، خطای کامپایل است.

خطای برنامه‌نویسی

تعریف یک تابع عضو ثابت که یک تابع عضو غیرثابت از کلاسی که از همان کلاس ساخته شده است، خطای کامپایل بدنبال خواهد داشت.

خطای برنامه‌نویسی

فعال کردن یک تابع عضو غیرثابت بر روی یک شی ثابت، خطای کامپایل است.

در این بین برای سازنده‌ها و نابودکننده‌ها که غالباً مبادرت به تغییر در شی‌ها می‌کنند، مشکل بوجود می‌آید. اعلان **const** اجازه‌ای برای سازنده‌ها و نابودکننده‌ها نمی‌دهد. یک سازنده بایستی اجازه تغییر در یک شی را داشته باشد از اینروست که شی می‌تواند بدرستی مقداردهی اولیه شود. یک نابودکننده باید قادر به انجام عملیات قبل از خاتمه قبل از اینکه حافظه اخذ شده توسط شی از سوی برنامه بازپس گرفته شده باشد.

خطای برنامه‌نویسی

*اقدام به اعلان یک سازنده یا نابودکننده **const**، خطای کامپایل است.*

تعریف و استفاده از توابع عضو ثابت

در برنامه شکل‌های ۱-۱۰ الی ۳-۱۰ کلاس **Time** از برنامه‌های ۹-۹ و ۱۰-۹ با اعمال توابع *get* و تابع ثابت **printUniversal** اصلاح شده است. در فایل سرآیند **Time.h** (شکل ۱-۱۰)، خطوط 21-19 و 24 حاوی کلمه‌کلیدی **const** پس از هر لیست پارامتری تابع هستند. تعریف متناظر با هر تابع در شکل ۲-۱۰ آورده شده است (خطوط 59، 53، 47 و 65) با اعمال کلمه‌کلیدی **const** پس لیست پارامتری هر تابع. در شکل ۳-۱۰ دو نمونه از شی **Time** ایجاد شده است. شی **wakeUp** بصورت غیرثابت (خط 7) و شی **noon** بصورت ثابت (خط 8). برنامه اقدام به فعال کردن توابع عضو غیرثابت **setHour** (خط 13) و **printStandard** (خط 20) بر روی شی ثابت **noon** می‌کند. در هر مورد، کامپایلر یک پیغام خطا تولید می‌کند. همچنین برنامه فراخوانی سه تابع عضو دیگر را عرضه کرده است. یک تابع عضو غیرثابت بر روی یک شی غیرثابت (خط 11)، یک تابع عضو ثابت بر روی یک شی غیرثابت (خط 15) و یک تابع عضو ثابت بر روی یک شی ثابت (خطوط 18-17). پیغام‌های خطای تولید شده از فراخوانی توابع عضو غیرثابت بر روی یک شی ثابت در خروجی برنامه نشان داده شده‌اند. توجه کنید که، اگرچه برخی از کامپایلرهای جاری فقط پیغام هشدار برای خطوط 13 و 20 صادر می‌کنند (که در اینحالت برنامه اجرا می‌شود)، اما ما به این هشدار بعنوان خطا نگاه می‌کنیم. استاندارد ANSI/ISO C++ اجازه فراخوانی یک تابع عضو غیرثابت بر روی یک شی ثابت را نمی‌دهد.

```
1  // Fig. 10.1: Time.h
2  // Definition of class Time.
3  // Member functions defined in Time.cpp.
4  #ifndef TIME_H
5  #define TIME_H
6
7  class Time
8  {
9  public:
10    Time( int = 0, int = 0, int = 0 ); // default constructor
11
12    // set functions
13    void setTime( int, int, int ); // set time
14    void setHour( int ); // set hour
15    void setMinute( int ); // set minute
16    void setSecond( int ); // set second
17
```

```
   // get functions (normally declared const)
   int getHour() const; // return hour
   int getMinute() const; // return minute
   int getSecond() const; // return second

   // print functions (normally declared const)
   void printUniversal() const; // print universal time
   void printStandard(); // print standard time (should be const)
private:
   int hour; // 0 - 23 (24-hour clock format)
   int minute; // 0 - 59
   int second; // 0 - 59
}; // end class Time

#endif
```

شکل ۱-۱۰ | تعریف کلاس Time با توابع عضو const.

```
// Fig. 10.2: Time.cpp
// Member-function definitions for class Time.
#include <iostream>
using std::cout;

#include <iomanip>
using std::setfill;
using std::setw;

#include "Time.h" // include definition of class Time

// constructor function to initialize private data;
// calls member function setTime to set variables;
// default values are 0 (see class definition)
Time::Time( int hour, int minute, int second )
{
   setTime( hour, minute, second );
} // end Time constructor

// set hour, minute and second values
void Time::setTime( int hour, int minute, int second )
{
   setHour( hour );
   setMinute( minute );
   setSecond( second );
} // end function setTime

// set hour value
void Time::setHour( int h )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
} // end function setHour

// set minute value
void Time::setMinute( int m )
{
   minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
} // end function setMinute

// set second value
void Time::setSecond( int s )
{
   second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
} // end function setSecond

// return hour value
int Time::getHour() const // get functions should be const
{
   return hour;
} // end function getHour

```

```
// return minute value
int Time::getMinute() const
{
   return minute;
} // end function getMinute

// return second value
int Time::getSecond() const
{
   return second;
} // end function getSecond

// print Time in universal-time format (HH:MM:SS)
void Time::printUniversal() const
{
   cout << setfill( '0' ) << setw( 2 ) << hour << ":"
      << setw( 2 ) << minute << ":" << setw( 2 ) << second;
} // end function printUniversal

// print Time in standard-time format (HH:MM:SS AM or PM)
void Time::printStandard() // note lack of const declaration
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
      << ":" << setfill( '0' ) << setw( 2 ) << minute
      << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
} // end function printStandard
```

شکل ۲-۱۰ | تعریف تابع عضو کلاس Time، شامل توابع عضو ثابت.

```
// Fig. 10.3: fig10_03.cpp
// Attempting to access a const object with non-const member functions.
#include "Time.h" // include Time class definition

int main()
{
   Time wakeUp( 6, 45, 0 ); // non-constant object
   const Time noon( 12, 0, 0 ); // constant object

                            // OBJECT      MEMBER FUNCTION
   wakeUp.setHour( 18 );    // non-const   non-const

   noon.setHour( 12 );      // const       non-const

   wakeUp.getHour();        // non-const   const

   noon.getMinute();        // const       const
   noon.printUniversal();   // const       const

   noon.printStandard();    // const       non-const
   return 0;
} // end main
```

Borland C++ command-line compiler error messages

```
Warning W8037 fig10_03.ccp 13:Non-const function Time::setHour(int)
     called for const object in function main()
Warning W8037 fig10_03.ccp 20:Non-const function Time::printStandard()
     called for const object in function main()
```

Microsoft Visual C++.NET compiler error message

```
C:\cpphtp5_examples\ch10\Fig10_01_03\fig10_03.cpp(13) : error C2662:
   'Time::setHour' : cannot convert 'this' pointer from 'const Time' to
   'Time &'
       Conversion loses qualifiers
C:\cpphtp5_examples\ch10\Fig10_01_03\fig10_03.cpp(20) : error C2662:
   'Time::printStandard':cannot convert 'this' pointer from 'const Time'
to
   'Time &'
       Conversion loses qualifiers
```

GNU C++ compiler error message

```
fig10_03.ccp:13: error: passing 'const Time' as 'this' argument of
```

```
     'void Time::setHour(int)' discards qualifiers
fig10_03.ccp:20: error: passing 'const Time' as 'this' argument of
     'void Time::printStandard()' discards qualifiers
```

شکل ۳-۱۰ | شی‌های ثابت و توابع عضو ثابت.

دقت کنید که حتی اگر یک سازنده بصورت یک تابع عضو غیرثابت باشد (شکل ۲-۱۰، خطوط 18-15) هنوز هم می‌تواند در مقداردهی اولیه یک شی **const** بکار گرفته شود (شکل ۳-۱۰، خط 8). تعریف سازنده Time (شکل ۲-۱۰ خطوط 18-15) نشان می‌دهد که سازنده **Time** تابع عضو غیرثابت دیگری بنام **setTime** (خطوط 26-21) را برای انجام مقداردهی اولیه یک شی **Time** فراخوانی می‌کند. فراخوانی یک تابع عضو غیرثابت از طریق فراخوانی سازنده بعنوان بخشی از مقداردهی اولیه یک شی ثابت، امکان‌پذیر است. توجه کنید که در خط 20 از شکل ۳-۱۰ یک خطای کامپایل تولید می‌شود حتی اگر تابع عضو **printStandard** از کلاس **Time** اقدام به تغییر در شی نکند.

مقداردهی اولیه یک عضو داده ثابت با یک مقداردهی کننده عضو

در برنامه شکل‌های ۴-۱۰ الی ۶-۱۰ به معرفی روش استفاده از گرامر مقداردهی کننده عضو می‌پردازیم. تمام اعضای داده می‌توانند با استفاده از گرامر مقداردهی کننده عضو، مقداردهی اولیه شوند، اما اعضای داده ثابت و اعضای داده که مورد مراجعه هستند بایستی با استفاده از مقداردهی کننده‌های عضو مقداردهی اولیه شوند. در ادامه این فصل، خواهید دید که شی‌های عضو بایستی به این روش مقداردهی اولیه شوند. در فصل دوازدهم به هنگام آموزش توارث، شاهد خواهید بود که قسمت‌های مبتنی بر کلاس از کلاس‌های مشتق شده هم بایستی به این روش مقداردهی اولیه شوند.

تعریف سازنده (شکل ۵-۱۰، خطوط 16-11) از لیست مقداردهی کننده عضو برای مقداردهی اولیه اعضای داده کلاس **Increment** بنام **count** که از نوع صحیح و ثابت نیست و **increment** از نوع صحیح و ثابت است (اعلان شده در خطوط 20-19 از شکل ۴-۱۰) استفاده کرده است. مقداردهی کننده عضو مابین یک لیست پارامتری سازنده و براکت چپ ظاهر می‌شوند که بدنه سازنده با آن آغاز می‌شود. لیست مقداردهی کننده عضو (شکل ۵-۱۰، خطوط 13-12) از لیست پارامتری توسط یک کولن (:) جدا شده است. هر مقداردهی کننده عضو متشکل از نام داده عضو بدنبال آن پرانتزهای حاوی مقدار اولیه عضو است. در این مثال، **count** با مقدار پارامتر سازنده **c** و **increment** با مقدار پارامتر سازنده **i** مقداردهی شده است. توجه کنید که مقداردهی کننده‌های عضو مضاعف توسط کاما از یکدیگر جدا می‌شوند. همچنین لیست مقداردهی کننده عضو قبل از اینکه بدنه سازنده اجرا شود، اجرا می‌شود.

```
1  // Fig. 10.4: Increment.h
2  // Definition of class Increment.
3  #ifndef INCREMENT_H
4  #define INCREMENT_H
5
6  class Increment
7  {
```

```
8  public:
9     Increment( int c = 0, int i = 1 ); // default constructor
10
11    // function addIncrement definition
12    void addIncrement()
13    {
14       count += increment;
15    } // end function addIncrement
16
17    void print() const; // prints count and increment
18 private:
19    int count;
20    const int increment; // const data member
21 }; // end class Increment
22
23 #endif
```

شکل ٤-١٠ | تعریف کلاس Increment حاوی عضو داده غیرثابت count و عضو داده ثابت increment.

```
1  // Fig. 10.5: Increment.cpp
2  // Member-function definitions for class Increment demonstrate using a
3  // member initializer to initialize a constant of a built-in data type.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "Increment.h" // include definition of class Increment
9
10 // constructor
11 Increment::Increment( int c, int i )
12    : count( c ), // initializer for non-const member
13      increment( i ) // required initializer for const member
14 {
15    // empty body
16 } // end constructor Increment
17
18 // print count and increment values
19 void Increment::print() const
20 {
21    cout << "count = " << count << ", increment = " << increment << endl;
22 } // end function print
```

شکل ٥-١٠ | استفاده از مقداردهی کننده عضو برای مقداردهی یک ثابت از نوع توکار.

```
1  // Fig. 10.6: fig10_06.cpp
2  // Program to test class Increment.
3  #include <iostream>
4  using std::cout;
5
6  #include "Increment.h" // include definition of class Increment
7
8  int main()
9  {
10    Increment value( 10, 5 );
11
12    cout << "Before incrementing: ";
13    value.print();
14
15    for ( int j = 1; j <= 3; j++ )
16    {
17       value.addIncrement();
18       cout << "After increment " << j << ": ";
19       value.print();
20    } // end for
21
22    return 0;
23 } // end main
```

```
Before incrementing: count = 10, increment = 5
After increment 1: count = 15, increment = 5
After increment 2: count = 20, increment = 5
```

```
After increment 3: count = 25, increment = 5
```

شکل ٦-١٠ | فراخوانی توابع عضو print و addIncrement از شی increment.

مقداردهی اشتباه یک عضو داده ثابت با عبارت تخصیصی

در برنامه شکل‌های ۷-۱۰ الی ۹-۱۰ به توضیح خطاهای کامپایل رخ داده به هنگام مبادرت به مقداردهی عضو داده ثابت **increment** با یک عبارت تخصیصی (شکل ۸-۱۰، خط 14) در بدنه سازنده **Increment** بجای یک لیست مقداردهی کننده عضو پرداخته شده است. به خط 13 از شکل ۸-۱۰ توجه کنید که خطای کامپایل تولید نمی‌کند، چرا که **count** بصورت ثابت **(const)** اعلان نشده است. همچنین به خطاهای کامپایل تولید شده توسط Microsoft Visual C++.NET در اشاره به عضو داده **increment** از نوع **int** بعنوان یک «*شی ثابت*» توجه کنید. همانند کلاس‌های نمونه‌سازی شده، متغیرهای که از نوع‌های بنیادین هستند هم در حافظه فضا اشغال می‌کنند و از اینرو غالباً از آنها بعنوان «شی» یاد می‌شود.

خطای برنامه‌نویسی

لیست مقداردهی کننده عضو برای یک عضو داده ثابت فراهم نکنید که با خطای کامپایل مواجه می‌شوید.

مهندسی نرم‌افزار

اعضای داده ثابت (شی‌های ثابت و متغیرهای ثابت) و اعضای داده اعلان شده بعنوان مراجعه باید با گرامر مقداردهی کننده عضو، مقداردهی اولیه شوند، اقدام به تخصیص به این نوع از داده‌ها در بدنه سازنده مجاز نمی‌باشد.

توجه کنید که تابع **print** (شکل ۸-۱۰، خطوط 21-18) بصورت ثابت اعلان شده است. ممکن است این عنوان برای این تابع کمی عجیب بنظر برسد، چرا که محتملاً برنامه هرگز دارای یک شی **Increment** ثابت نخواهد بود. با این وجود، ممکن است که برنامه یک مراجعه ثابت به یک شی **Increment** یا یک اشاره‌گر به ثابتی داشته باشد که به یک شی **Increment** اشاره می‌کند. معمولاً اینحالت زمانی رخ می‌دهد که شی‌های از کلاس **Increment** به توابع ارسال یا از توابع برگشت داده شوند. در این موارد، فقط توابع عضو ثابت از کلاس **Increment** می‌توانند از طریق مراجعه یا اشاره‌گر فراخوانی شوند. بنابر این، اعلان تابع **print** بصورت ثابت، معقول بنظر می‌رسد. با انجام چنین کاری از رخ دادن خطاهای در این رابطه جلوگیری می‌شود.

اجتناب از خطا

*تمام توابع عضو کلاس را که در ناحیه عملکردی خود مبادرت به اعمال تغییر در شی نمی‌کنند، از نوع **const** (ثابت) اعلان کنید.*

```
1  // Fig. 10.7: Increment.h
2  // Definition of class Increment.
3  #ifndef INCREMENT_H
4  #define INCREMENT_H
5
```

```
6  class Increment
7  {
8  public:
9     Increment( int c = 0, int i = 1 ); // default constructor
10
11    // function addIncrement definition
12    void addIncrement()
13    {
14       count += increment;
15    } // end function addIncrement
16
17    void print() const; // prints count and increment
18 private:
19    int count;
20    const int increment; // const data member
21 }; // end class Increment
22
23 #endif
```

شکل ۷-۱۰ | تعریف کلاس Increment حاوی یک عضو داده غیرثابت counst و عضو داده ثابت increment.

```
1  // Fig. 10.8: Increment.cpp
2  // Attempting to initialize a constant of
3  // a built-in data type with an assignment.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "Increment.h" // include definition of class Increment
9
10 // constructor; constant member 'increment' is not initialized
11 Increment::Increment( int c, int i )
12 {
13    count = c; // allowed because count is not constant
14    increment = i; // ERROR: Cannot modify a const object
15 } // end constructor Increment
16
17 // print count and increment values
18 void Increment::print() const
19 {
20    cout << "count = " << count << ", increment = " << increment << endl;
21 } // end function print
```

شکل ۸-۱۰ | مقداردهی سهوی یک ثابت از نوع توکار توسط عبارت تخصیصی.

```
1  // Fig. 10.9: fig10_09.cpp
2  // Program to test class Increment.
3  #include <iostream>
4  using std::cout;
5
6  #include "Increment.h" // include definition of class Increment
7
8  int main()
9  {
10    Increment value( 10, 5 );
11
12    cout << "Before incrementing: ";
13    value.print();
14
15    for ( int j = 10; j <= 3; j++ )
16    {
17       value.addIncrement();
18       cout << "After increment " << j << ": ";
19       value.print();
20    } // end for
21
22    return 0;
23 } // end main
```

Borland C++ command-line compiler error messages

```
Error E2024 Increment.cpp 14: Cannot modify a const object in function
```

```
    Increment::Increment(int,int)
```

Microsoft Visual C++.NET compiler error message

```
C:\cpphtp5_examples\ch10\Fig10_07_09\Increment.cpp(12) : error C2758:
   'Increment::increment' :must be initialized in constructor
   base/member initializer list
        C:\cpphtp5_examples\ch10\Fig10_07_09\Increment.h(20) :
        see declaration of 'Increment::increment'
C:\cpphtp5_examples\ch10\Fig10_07_09\Increment.cpp(14): error C2166:
   l-value specifies const object
```

GNU C++ compiler error message

```
Increment.cpp:12: error: uninitialized member 'Increment::increment'with
     'const'type 'const int'
Increment.cpp:14: error: assigment of read-only data-member
     'Increment::increment'
```

شکل ۹-۱۰ | برنامه تست‌کننده کلاس Increment که خطاهای کامپایل تولید می‌کند.

۳-۱۰ ترکیب: شی‌ها بعنوان اعضای کلاس

یک شی **AlarmClock** نیاز دارد تا از زمان فرض شده برای بصدا در آوردن زنگ مطلع باشد، پس چرا نبایستی یک شی **Time** بعنوان عضوی از کلاس **AlarmClock** بحساب آورده نشود؟ چنین قابلیتی *ترکیب* نامیده می‌شود و گاهی اوقات بعنوان «بستگی داشتن یا رابطه داشتن» شناخته می‌شود. یک کلاس می‌تواند شی‌های از سایر کلاس‌ها را بعنوان اعضاء داشته باشد.

مهندسی نرم‌افزار

یکی از فرم‌های استفاده مجدد از نرم‌افزار، ترکیب است که در آن یک کلاس دارای شی‌های از سایر کلاس‌ها بعنوان اعضا است.

زمانیکه یک شی ایجاد می‌شود، سازنده آن بصورت اتوماتیک فراخوانی می‌گردد. قبلاً شاهد نحوه ارسال آرگومان‌ها به سازنده یک شی که در **main** ایجاد می‌کردیم بوده‌اید. در این بخش شاهد خواهید بود که چگونه سازنده یک شی می‌تواند آرگومان‌ه‌ای به سازنده‌های عضو شی ارسال کند که از طریق از مقداردهی کننده‌های عضو صورت می‌گیرد. شی‌های عضو به ترتیبی که در تعریف کلاس اعلان شده‌اند (نه به ترتیبی که در لیست مقداردهی کننده عضو سازنده ظاهر شده‌اند) و قبل از ایجاد شی‌های کلاس احاطه کننده (شی‌های میزبان) ساخته می‌شوند.

در برنامه شکل‌های ۱۰-۱۰ الی ۱۴-۱۰ از کلاس **Date** (شکل‌های ۱۰-۱۰ و ۱۱-۱۰) و کلاس **Employee** (شکل‌های ۱۲-۱۰ و ۱۳-۱۰) برای نشان دادن شی‌هایی بعنوان اعضای سایر شی‌ها استفاده شده است. تعریف کلاس **Employee** (شکل ۱۲-۱۰) حاوی اعضای داده خصوصی بنام‌های **firstName**، **lastName**، **birthDate** و **hireDate** است. اعضای **birthDate** و **hireDate** شی‌های ثابت از کلاس **Date** هستند که حاوی اعضای داده خصوصی بنام‌های **month**، **day** و **year** می‌باشند. سرآیند سازنده **Employee** (شکل ۱۳-۱۰ خطوط 21-18) مشخص می‌کند که سازنده چهار پارامتر دریافت می‌کند(**first, last, dateOfBirth, dateOfHire**). از دو پارامتر اول در بدنه سازنده برای مقداردهی

اولیه آرایه‌های کاراکتری **firstName** و **lastName** استفاده می‌شود. دو پارامتر آخر از طریق لیست‌های مقداردهی کننده به سازنده کلاس **Date** ارسال می‌شوند. کولن (:) بکار رفته در سرآیند منجر به جداسازی مقداردهی کننده‌های عضو از لیست پارامتری می‌شود. مقداردهی کننده‌های عضو، مشخص می‌کنند که پارامترهای سازنده **Employee** به سازنده‌های شی‌های عضو **Date** ارسال می‌گردند. پارامتر **dateOfBirth** به شی سازنده **birthDate** (شکل ۱۳-۱۰، خط 20) و پارامتر **dateOfHire** به شی سازنده **hireDate** (شکل ۱۳-۱۰، خط 21) ارسال می‌شوند. مجموعاً، مقداردهی کننده‌های عضو توسط کاما از یکدیگر جدا شده‌اند.

به هنگام معرفی کلاس **Date** (شکل ۱۰-۱۰) توجه کنید که این کلاس دارای یک سازنده که پارامتری از نوع **Date** دریافت کند نیست. از اینرو چگونه لیست مقداردهی کننده عضو در سازنده کلاس **Employee** قادر به مقداردهی شی‌های **birthDate** و **hireDate** با ارسال شی‌های **Date** به سازنده‌های **Date** آنها صورت می‌گیرد؟ همانطوری که در فصل نهم گفته شد، کامپایلر برای هر کلاس یک سازنده پیش‌فرض کپی کننده که مبادرت به کپی هر عضو از شی از آرگومان شی سازنده به عضو متناظر از شی که مقداردهی می‌شود، می‌کند. در فصل یازدهم خواهید آموخت که چگونه برنامه‌نویسان می‌توانند سازنده‌های کپی کننده بهینه شده تعریف کنند.

```
1  // Fig. 10.10: Date.h
2  // Date class definition; Member functions defined in Date.cpp
3  #ifndef DATE_H
4  #define DATE_H
5
6  class Date
7  {
8  public:
9     Date( int = 1, int = 1, int = 1900 ); // default constructor
10    void print() const; // print date in month/day/year format
11    ~Date(); // provided to confirm destruction order
12 private:
13    int month; // 1-12 (January-December)
14    int day; // 1-31 based on month
15    int year; // any year
16
17    // utility function to check if day is proper for month and year
18    int checkDay( int ) const;
19 }; // end class Date
20
21 #endif
```

شکل ۱۰-۱۰ | تعریف کلاس Date.

در برنامه شکل ۱۴-۱۰ دو شی **Date** ایجاد (خطوط 12-11) و آنها را بعنوان آرگومان‌هایی به سازنده شی **Employee** ایجاد شده در خط 13 ارسال می‌کند. خط 16 داده شی **Employee** را در خروجی قرار می‌دهد. زمانیکه هر شی **Date** در خطوط 12-11 ایجاد می‌شود، سازنده **Date** تعریف شده در خطوط 11-28 از شکل ۱۱-۱۰ در یک خط خروجی نمایش می‌دهد که سازنده فراخوانی شده است (به سطر اول خروجی نگاه کنید).

```
1  // Fig. 10.11: Date.cpp
2  // Member-function definitions for class Date.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Date.h" // include Date class definition
8
9  // constructor confirms proper value for month; calls
10 // utility function checkDay to confirm proper value for day
11 Date::Date( int mn, int dy, int yr )
12 {
13    if ( mn > 0 && mn <= 12 ) // validate the month
14       month = mn;
15    else
16    {
17       month = 1; // invalid month set to 1
18       cout << "Invalid month (" << mn << ") set to 1.\n";
19    } // end else
20
21    year = yr; // could validate yr
22    day = checkDay( dy ); // validate the day
23
24    // output Date object to show when its constructor is called
25    cout << "Date object constructor for date ";
26    print();
27    cout << endl;
28 } // end Date constructor
29
30 // print Date object in form month/day/year
31 void Date::print() const
32 {
33    cout << month << '/' << day << '/' << year;
34 } // end function print
35
36 // output Date object to show when its destructor is called
37 Date::~Date()
38 {
39    cout << "Date object destructor for date ";
40    print();
41    cout << endl;
42 } // end ~Date destructor
43
44 // utility function to confirm proper day value based on
45 // month and year; handles leap years, too
46 int Date::checkDay( int testDay ) const
47 {
48    static const int daysPerMonth[ 13 ] =
49       { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
50
51    // determine whether testDay is valid for specified month
52    if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
53       return testDay;
54
55    // February 29 check for leap year
56    if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
57       ( year % 4 == 0 && year % 100 != 0 ) ) )
58       return testDay;
59
60    cout << "Invalid day (" << testDay << ") set to 1.\n";
61    return 1; // leave object in consistent state if bad value
62 } // end function checkDay
```

شکل ۱۱-۱۰ | تعریف تابع عضو کلاس Date.

```
1  // Fig. 10.12: Employee.h
2  // Employee class definition.
3  // Member functions defined in Employee.cpp.
4  #ifndef EMPLOYEE_H
5  #define EMPLOYEE_H
6
```

```
7  #include "Date.h" // include Date class definition
8
9  class Employee
10 {
11 public:
12    Employee( const char * const, const char * const,
13       const Date &, const Date & );
14    void print() const;
15    ~Employee(); // provided to confirm destruction order
16 private:
17    char firstName[ 25 ];
18    char lastName[ 25 ];
19    const Date birthDate; // composition: member object
20    const Date hireDate; // composition: member object
21 }; // end class Employee
22
23 #endif
```

شکل ۱۲-۱۰ | تعریف کلاس Employee که ترکیب را نمایش می‌دهد.

```
1  // Fig. 10.13: Employee.cpp
2  // Member-function definitions for class Employee.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstring> // strlen and strncpy prototypes
8  using std::strlen;
9  using std::strncpy;
10
11 #include "Employee.h" // Employee class definition
12 #include "Date.h" // Date class definition
13
14 // constructor uses member initializer list to pass initializer
15 // values to constructors of member objects birthDate and hireDate
16 // [Note: This invokes the so-called "default copy constructor" which the
17 // C++ compiler provides implicitly.]
18 Employee::Employee( const char * const first, const char * const last,
19    const Date &dateOfBirth, const Date &dateOfHire )
20    : birthDate( dateOfBirth ), // initialize birthDate
21      hireDate( dateOfHire ) // initialize hireDate
22 {
23    // copy first into firstName and be sure that it fits
24    int length = strlen( first );
25    length = ( length < 25 ? length : 24 );
26    strncpy( firstName, first, length );
27    firstName[ length ] = '\0';
28
29    // copy last into lastName and be sure that it fits
30    length = strlen( last );
31    length = ( length < 25 ? length : 24 );
32    strncpy( lastName, last, length );
33    lastName[ length ] = '\0';
34
35    // output Employee object to show when constructor is called
36    cout << "Employee object constructor: "
37       << firstName << ' ' << lastName << endl;
38 } // end Employee constructor
39
40 // print Employee object
41 void Employee::print() const
42 {
43    cout << lastName << ", " << firstName << "  Hired: ";
44    hireDate.print();
45    cout << "  Birthday: ";
46    birthDate.print();
47    cout << endl;
48 } // end function print
49
50 // output Employee object to show when its destructor is called
```

```
51 Employee::~Employee()
52 {
53    cout << "Employee object destructor: "
54       << lastName << ", " << firstName << endl;
55 } // end ~Employee destructor
```

شکل ۱۳-۱۰ | تعریف تابع عضو کلاس Employee شامل سازنده با یک لیست مقداردهی کننده عضو.

```
1  // Fig. 10.14: fig10_14.cpp
2  // Demonstrating composition--an object with member objects.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Employee.h" // Employee class definition
8
9  int main()
10 {
11    Date birth( 7, 24, 1949 );
12    Date hire( 3, 12, 1988 );
13    Employee manager( "Bob", "Blue", birth, hire );
14
15    cout << endl;
16    manager.print();
17
18    cout << "\nTest Date constructor with invalid values:\n";
19    Date lastDayOff( 14, 35, 1994 ); // invalid month and day
20    cout << endl;
21    return 0;
22 } // end main
```

```
Date object constructor date 7/24/1949
Date object constructor date 3/12/1988
Employee object constructor: Bob Blue

Blue, Bob  Hired: 3/12/1988 Birthday: 7/24/1949


Test Date constructor with invalid values:
Invalid month (14) set to 1.
Invalid day (35) set to 1.
Date object constructor for date 1/1/1994

Date object destructor for date 1/1/1994
Employee object destructor: Blue, Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
```

شکل ۱۴-۱۰ | مقداردهی کننده‌های عضو شی.

کلاس **Date** و کلاس **Employee** هر یک شامل یک نابودکننده هستند (به‌ترتیب خطوط 42-37 از شکل ۱۱-۱۰ و خط 55-51 از شکل ۱۳-۱۰) که به هنگام نابودی یک شی از کلاس مربوطه، یک پیغام چاپ می‌کنند. این امکان به ما اجازه می‌دهد تا توسط خروجی برنامه تایید کنیم که شی‌ها از داخل به خارج ایجاد شده و به ترتیب معکوس از خارج به داخل نابود می‌شوند (یعنی، اعضای عضو **Date** پس از اینکه شی **Employee** که حاوی آنها است، نابود شد از بین می‌روند). در خروجی شکل ۱۴-۱۰ به چهار خط پایانی توجه کنید. دو خط پایانی خروجی اجرای نابود کننده **Date** بر روی شی‌های **hire** (خط 12) و **birth** (خط 11) است. این خروجی‌ها تایید می‌کنند که سه شی ایجاد شده در **main** به‌ترتیب معکوس از

ایجاد شدن، نابود شده‌اند. خطوط چهارم و پنجم از خروجی، نمایشی از اجرای نابودکننده‌ها بر روی شی‌های عضو کلاس **Employee** بنام‌های **hireDate** (شکل ۱۲-۱۰، خط 20) و **birthDate** (شکل ۱۲-۱۰، خط 19) است. این خروجی‌ها تایید می‌کنند که شی **Employee** از خارج به درون نابود می‌شود، یعنی ابتدا نابودکننده **Employee** اجرا می‌شود، سپس شی‌های عضو به ترتیب معکوس از حالتی که ایجاد شده‌اند نابود می‌گردند. مجدداً در خروجی شکل ۱۴-۱۰ خبری از سازنده‌ها برای این شی‌ها نیست، چرا که برای آنها سازنده‌های کپی کننده پیش‌فرض توسط کامپایلر C++ تدارک دیده شده است.

یک شی عضو نیازی به مقداردهی صریح اولیه از طریق یک مقداردهی کننده عضو ندارد. اگر یک مقداردهی کننده عضو در نظر گرفته نشده باشد، بطور ضمنی سازنده پیش‌فرض برای آن شی عضو فراخوانی خواهد شد. مقادیر تدارک دیده شده توسط سازنده پیش‌فرض هر چه باشند، می‌توانند توسط توابع *set* بازنویسی شوند. با این همه، برای مقداردهی‌های پیچیده، چنین روشی مستلزم کار و زمان بیشتری است.

در شکل ۱۱-۱۰ و خط 26، به فراخوانی تابع عضو **print** از **Date** توجه کنید. برخی از توابع عضو در C++ نیازی به آرگومان ندارند. به این دلیل که هر تابع عضو حاوی یک دستگیره (هندل) ضمنی (بفرم یک اشاره‌گر) به شی است که بر روی آن عمل می‌کند. در بخش ۵-۱۰ به معرفی اشاره‌گرهای ضمنی خواهیم پرداخت که توسط کلمه کلیدی **this** معرفی می‌شوند.

کلاس **Employee** از دو آرایه 25 کاراکتری (شکل ۱۲-۱۰، خطوط 18-17) برای عرضه نام و نام خانوادگی کارمند سود می‌برد. امکان دارد این آرایه به هنگام مواجه شدن با اسامی کمتر از 24 کاراکتر، فضای حافظه را تلف کند. همچنین اسامی طولانی‌تر از 24 کاراکتر برای اینکه با سایز آرایه هماهنگ شوند، کوتاه خواهند شد. در بخش ۷-۱۰ به معرفی نسخه دیگری از کلاس **Employee** خواهیم پرداخت که بصورت دینامیکی و دقیقاً به میزان مورد نیاز برای نگهداری نام و نام خانوادگی فضا ایجاد می‌کند. یکی از روش‌های ساده برای عرضه نام و نام خانوادگی یک کارمند استفاده از دو شی رشته (***string***) است که فضای مورد نیاز را تدارک می‌بینند. اگر چنین کاری انجام دهیم، سازنده **Employee** بصورت زیر خواهد بود

```
Employee::Employee(const string &first, const string &last,
    const Date &dateOfBirth, const Date &dateOfHire)
    :firstName(first),//initialize firstName
     lastName(last),// initialize lastName
     birthDate(dateOfBirth),// initialize birthDate
     hireDate(dateOfHire)// initialize hireDate
{
     // output Employee object to show when constructor is called
     cout << "Employee object constructor:"
          <<firstName<<' '<<lastName<<endl;
} // end Employee constructor
```

دقت کنید که اعضای داده **firstName** و **lastName** (شی‌های رشته) از طریق مقداردهی کننده‌های عضو، مقداردهی شده‌اند. کلاس‌های **Employee** معرفی شده در فصل‌های ۱۲ و ۱۳ از شی‌های **string** به این روش استفاده می‌کنند. در این فصل، از رشته‌های مبتنی بر اشاره‌گر استفاده کرده‌ایم تا خواننده با کاربرد اشاره‌گرها بیشتر آشنا شود.

۱۰-٤ توابع و کلاس‌های friend

با اینکه تابع **friend** یک کلاس، خارج از قلمرو کلاس تعریف می‌شود، اما هنوز هم دارای مجوز دسترسی اعضای غیرسراسری (و سراسری) کلاس می‌باشد. توابع منفرد یا کل کلاس‌ها می‌توانند برای کلاس دیگری بصورت **friend** (*دوست*) اعلان شوند.

توابع **friend** می‌توانند در افزایشی کارایی موثر باشند. در این بخش به معرفی یک مثال غیرکاربردی از نحوه عملکرد و توابع **friend** می‌پردازیم. سپس در ادامه این کتاب، از توابع **friend** در عملگرهای سربارگذاری شده برای استفاده با شی‌های کلاس (فصل ۱۱) و ایجاد کلاس‌های تکرار شونده استفاده خواهیم کرد.

برای اعلان یک تابع بعنوان *دوست یک کلاس*، قبل از نمونه اولیه تابع در تعریف کلاس از کلمه کلیدی ***friend*** استفاده می‌شود. برای اعلان تمام توابع عضو کلاس **ClassTwo** بصورت دوستان کلاس **ClassOne**، از اعلان زیر در تعریف کلاس **ClassOne** استفاده می‌شود.

```
friend class ClassTwo
```

دقت کنید که دوستی اهدا می‌شود، اما الزاما پذیرفته نمی‌شود، یعنی کلاس **B** می‌تواند دوست کلاس **A** باشد، اما کلاس **A** بایستی بصورت صریح اعلان کند که کلاس **B** دوست او است. همچنین رابطه دوستی حالت متقارن یا انتقالی ندارد، یعنی اگر کلاس **A** دوست کلاس **B** باشد، و کلاس **B** دوست کلاس **C** باشد، نمی‌توانید استنتاج کنید که کلاس **B** دوست کلاس **A** است (دوستی حالت متقارن ندارد)، و کلاس **C** دوست کلاس **B** است (چرا که دوستی حالت متقارن ندارد) یا اینکه کلاس **A** دوست کلاس **C** است (دوستی حالت انتقالی ندارد).

تغییر در داده private یک کلاس توسط تابع friend

در برنامه شکل ۱۵-۱۰ یک مثال غیرکابردی عرضه شده که در آن تابع دوست setX برای تنظیم داده خصوصی (private) عضو داده x از کلاس **Count** تعریف شده است. دقت کنید که اعلان **friend** (خط 10) در ابتدای تعریف کلاس آورده شده است (بطور قراردادی) حتی قبل از اعلان توابع عضو سراسری (public). توجه کنید که این اعلان **friend** می‌تواند در هر کجای کلاس آورده شود.

```
1  // Fig. 10.15: fig10_15.cpp
2  // Friends can access private members of a class.
3  #include <iostream>
4  using std::cout;
```

```
5  using std::endl;
6
7  // Count class definition
8  class Count
9  {
10    friend void setX( Count &, int ); // friend declaration
11 public:
12    // constructor
13    Count()
14       : x( 0 ) // initialize x to 0
15    {
16       // empty body
17    } // end constructor Count
18
19    // output x
20    void print() const
21    {
22       cout << x << endl;
23    } // end function print
24 private:
25    int x; // data member
26 }; // end class Count
27
28 // function setX can modify private data of Count
29 // because setX is declared as a friend of Count (line 10)
30 void setX( Count &c, int val )
31 {
32    c.x = val; // allowed because setX is a friend of Count
33 } // end function setX
34
35 int main()
36 {
37    Count counter; // create Count object
38
39    cout << "counter.x after instantiation: ";
40    counter.print();
41
42    setX( counter, 8 ); // set x using a friend function
43    cout << "counter.x after call to setX friend function: ";
44    counter.print();
45    return 0;
46 } // end main
```

```
counter.x after instantion: 0
counter.x after call to setX friend function: 8
```

شکل ۱۵-۱۰ | دوستان می‌توانند به اعضای خصوصی یک کلاس دسترسی داشته باشند.

تابع **setX** (خطوط 33-30) یک تابع منفرد به سبک C است و تابع عضوی از کلاس **Count** نمی‌باشد. به همین دلیل، زمانیکه **setX** برای شی **counter** فراخوانی می‌شود، خط 42 مبادرت به ارسال **counter** بعنوان یک آرگومان به **setX** بجای استفاده از یک دستگیره (مانند نام یک شی) برای فراخوانی تابع می‌کند، همانند

```
counter.setX(8);
```

همانطوری که قبلاً هم گفته شد برنامه ۱۵-۱۰ یک برنامه غیرکاربردی است که در آن از **friend** استفاده شده است. مقتضی است که تابع **setX** بصورت یک تابع عضو از کلاس **Count** تعریف شود. همچنین متمایز کردن برنامه ۱۵-۱۰ به سه فایل هم می‌تواند مناسب باشد:

۱ـ فایل سرآیند (مانند *Count.h*) حاوی تعریف کلاس **Count**، که حاوی نمونه اولیه تابع دوست **setX** است.

۲ـ پیاده سازی فایل (مانند *Count.cpp*) حاوی تعاریف توابع عضو کلاس **Count** و تعریف تابع دوست **setX**.

۳- برنامه تست کننده (مانند *fig10_15.cpp*) با **main**.

اشتباه سهوی در تغییر یک عضو خصوصی با یک تابع غیردوست

برنامه شکل ۱۶-۱۰ به بررسی پیغام‌های خطا می‌پردازد که توسط کامپایلر و در زمانیکه تابع غیردوست cannotSetX برای تغییر در داده عضو خصوصی x فراخوانی می‌شود (خطوط 32-29). امکان تصریح توابع سربارگذاری شده به عنوان دوستان کلاس وجود دارد. هر تابع سربارگذاری شده که قصد دارد حالت دوست داشته باشد بایستی بصورت صریح در تعریف کلاس بعنوان دوست کلاسی اعلان شده باشد.

```
1  // Fig. 10.16: fig10_16.cpp
2  // Non-friend/non-member functions cannot access private data of a class.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // Count class definition (note that there is no friendship declaration)
8  class Count
9  {
10 public:
11    // constructor
12    Count()
13       : x( 0 ) // initialize x to 0
14    {
15       // empty body
16    } // end constructor Count
17
18    // output x
19    void print() const
20    {
21       cout << x << endl;
22    } // end function print
23 private:
24    int x; // data member
25 }; // end class Count
26
27 // function cannotSetX tries to modify private data of Count,
28 // but cannot because the function is not a friend of Count
29 void cannotSetX( Count &c, int val )
30 {
31    c.x = val; // ERROR: cannot access private member in Count
32 } // end function cannotSetX
33
34 int main()
35 {
36    Count counter; // create Count object
37
38    cannotSetX( counter, 3 ); // cannotSetX is not a friend
39    return 0;
40 } // end main
```

Borland C++ command-line compiler error messages

```
Error E2247 Fig10_16/fig10_16.ccp 31:  'Conut::x'  is not accessible in
     function cannotSetX(Count &,int)
```

Microsoft Visual C++.NET compiler error message

```
C:\cpphtp5_examples\ch10\Fig10_16\fig10_16.cpp(31):error C2248: 'Count::x'
   : cannot access private member declared in class 'Count'
        C:\ccphtp5_examples\ch10\Fig10_16\fig10_16.ccp(24):see declaration
           of 'Count::x'
        C:\cpphtp5_examples\ch10\Fig10_16\fig10_16.cpp(9) :see declaration
```

GNU C++ compiler error message

```
fig10_16.ccp:24: error: 'int Count::x' is private
fig10_16.ccp:31: error: within this context
```

شکل ١٦-١٠ | توابع غیردوست/غیرعضو قادر به دسترسی به اعضای خصوصی نمی‌باشند.

۱۰-۵ استفاده از اشاره‌گر this

مشاهده کردید که یک شی از توابع عضو می‌تواند در داده شی دستکاری کند. چگونه توابع عضو می‌دانند که کدام یک از اعضای داده شی را دستکاری کنند؟ هر شی از طریق یک اشاره‌گر بنام **this** (یک کلمه کلیدی در C++) به آدرس متعلق بخود دسترسی دارد. اشاره‌گر **this** یک شی، بخشی از خود شی نمی‌باشد، یعنی سایز حافظه اشغال شده توسط اشاره‌گر **this** تاثیری در نتیجه اجرای **sizeOf** بر روی شی ندارد. بجای آن اشاره‌گر **this** بصورت یک آرگومان ضمنی به هر تابع عضو غیراستاتیک شی ارسال می‌شود (توسط کامپایلر). در بخش ۷-۱۰ به معرفی اعضای کلاس استاتیک و توضیح اینکه چرا اشاره‌گرهای **this** بصورت غیرصریح به توابع عضو استاتیک ارسال می‌شوند، پرداخته شده است.

شی‌ها از اشاره‌گر **this** بصورت ضمنی (که در این بخش آنرا انجام می‌دهیم) یا صریح برای مراجعه اعضای داده و توابع عضو خود استفاده می‌کنند. نوع اشاره‌گر **this** بستگی به نوع شی دارد و خواه تابع عضو که در آن از **this** استفاده شده ثابت باشد یا خیر. برای مثال، در یک تابع عضو غیرثابت از کلاس **Employee**، اشاره‌گر **this** دارای نوع **Employee *const** است (یک ثابت اشاره‌گر به یک شی غیرثابت **Employee**). در تابع عضو ثابت از کلاس **Employee**، اشاره‌گر **this** دارای نوع داده **const Employee * const** است (یک ثابت اشاره‌گر به یک شی ثابت **Employee**). اولین مثال ما در این بخش نمایش استفاده ضمنی و صریح از اشاره‌گر **this** است.

استفاده ضمنی و صریح از اشاره‌گر this برای دسترسی به اعضا داده یک شی

برنامه شکل ۱۷-۱۰ به بیان نحوه استفاده و صریح از اشاره‌گر **this** بر روی یک تابع عضو از کلاس **Test** برای چاپ داده خصوصی **x** از شی **Test** پرداخته است. برای بیان این هدف، ابتدا تابع عضو **print** (خطوط 37-25) مقدار **x** را با استفاده از اشاره‌گر **this** بصورت ضمنی چاپ می‌کند (خط 28)، فقط نام عضو داده مشخص شده است. پس از **print** به دو روش برای دسترسی به x از طریق اشاره‌گر **this** استفاده شده است. عملگر فلش (<-) و عملگر نقطه (.).

```
1  // Fig. 10.17: fig10_17.cpp
2  // Using the this pointer to refer to object members.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  class Test
8  {
9  public:
10    Test( int = 0 ); // default constructor
11    void print() const;
12 private:
13    int x;
```

```
14 }; // end class Test
15
16 // constructor
17 Test::Test( int value )
18    : x( value ) // initialize x to value
19 {
20    // empty body
21 } // end constructor Test
22
23 // print x using implicit and explicit this pointers;
24 // the parentheses around *this are required
25 void Test::print() const
26 {
27    // implicitly use the this pointer to access the member x
28    cout << "        x = " << x;
29
30    // explicitly use the this pointer and the arrow operator
31    // to access the member x
32    cout << "\n  this->x = " << this->x;
33
34    // explicitly use the dereferenced this pointer and
35    // the dot operator to access the member x
36    cout << "\n(*this).x = " << ( *this ).x << endl;
37 } // end function print
38
39 int main()
40 {
41    Test testObject( 12 ); // instantiate and initialize testObject
42
43    testObject.print();
44    return 0;
45 } // end main
```

```
          x = 12
   this->x = 12
(*this ).x = 12
```

شکل ۱۷-۱۰ | دسترس ضمنی و صریح اشاره‌گر this به اعضای یک شی.

به پرانتزهای قرار گرفته در اطراف **this*** (خط 36) به هنگام استفاده از عملگر انتخاب عضو (.) توجه کنید. وجود پرانتزها ضروری است چرا که عملگر نقطه به نسبت عملگر * از اولویت بالاتری برخوردار است. بدون حضور پرانتزها، عبارت **this.x*** با خطای کامپایل مواجه خواهد شد، چرا که عملگر نقطه نمی‌تواند با اشاره‌گر بکار گرفته شود.

یکی از نکات جالب در استفاده از اشاره‌گر **this** اجتناب از تخصیص یک شی به خودش است. همانطوری که در فصل یازدهم شاهد خواهید بود، تخصیص بخود می‌تواند خطاهای بسیاری جدی در زمانیکه شی حاوی اشاره‌گرها با فضای اخذ شده دینامیکی باشد، بوجود آورد.

استفاده از اشاره‌گر this برای فراخوانی آبشاری تابع

یکی دیگر از کاربردهای اشاره‌گر **this** فراخوانی *آبشاری* توابع عضو است که در آن توابع مضاعف توسط یک عبارت فراخوانی می‌شوند (همانند خط 14 از برنامه شکل ۲۰-۱۰). برنامه شکل‌های ۱۸-۱۰ الی ۲۰-۱۰ تغییر یافته توابع **setTime**، **setHour**، **setMinute** و **setSecound** از کلاس **Time** هستند که هر یک مراجعه‌ای به یک شی **Time** برگشت می‌دهند تا فراخوانی آبشاری تابع امکان‌پذیر باشد. در شکل

۲۷۶فصل دهم ـــــــ

```
1  // Fig. 10.19: Time.cpp
2  // Member-function definitions for Time class.
3  #include <iostream>
4  using std::cout;
5
6  #include <iomanip>
7  using std::setfill;
8  using std::setw;
9
10 #include "Time.h" // Time class definition
11
12 // constructor function to initialize private data;
13 // calls member function setTime to set variables;
14 // default values are 0 (see class definition)
15 Time::Time( int hr, int min, int sec )
16 {
17    setTime( hr, min, sec );
18 } // end Time constructor
19
20 // set values of hour, minute, and second
21 Time &Time::setTime( int h, int m, int s ) // note Time & return
22 {
23    setHour( h );
24    setMinute( m );
25    setSecond( s );
26    return *this; // enables cascading
27 } // end function setTime
28
29 // set hour value
30 Time &Time::setHour( int h ) // note Time & return
31 {
32    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
33    return *this; // enables cascading
34 } // end function setHour
35
36 // set minute value
37 Time &Time::setMinute( int m ) // note Time & return
38 {
39    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
40    return *this; // enables cascading
41 } // end function setMinute
42
43 // set second value
44 Time &Time::setSecond( int s ) // note Time & return
45 {
46    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
47    return *this; // enables cascading
48 } // end function setSecond
49
50 // get hour value
51 int Time::getHour() const
52 {
53    return hour;
54 } // end function getHour
55
56 // get minute value
57 int Time::getMinute() const
58 {
59    return minute;
60 } // end function getMinute
61
62 // get second value
63 int Time::getSecond() const
64 {
65    return second;
66 } // end function getSecond
67
68 // print Time in universal-time format (HH:MM:SS)
69 void Time::printUniversal() const
70 {
```

```
   cout << setfill( '0' ) << setw( 2 ) << hour << ":"
      << setw( 2 ) << minute << ":" << setw( 2 ) << second;
} // end function printUniversal

// print Time in standard-time format (HH:MM:SS AM or PM)
void Time::printStandard() const
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
      << ":" << setfill( '0' ) << setw( 2 ) << minute
      << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
} // end function printStandard
```

شکل ۱۹-۱۰ | تعریف تابع عضو کلاس Time اصلاح شده تا فراخوانی آبشاری تابع عضو امکان‌پذیر شود.

```
// Fig. 10.20: fig10_20.cpp
// Cascading member function calls with the this pointer.
#include <iostream>
using std::cout;
using std::endl;

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

int main()
{
   Time t; // create Time object

   // cascaded function calls
   t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );

   // output time in universal and standard formats
   cout << "Universal time: ";
   t.printUniversal();

   cout << "\nStandard time: ";
   t.printStandard();

   cout << "\n\nNew standard time: ";

   // cascaded function calls
   t.setTime( 20, 20, 20 ).printStandard();
   cout << endl;
   return 0;
} // end main
```

```
Universal time: 18:30:22
Standard time: 6:30:22 PM

New standard time: 8:20:20 PM
```

شکل ۲۰-۱۰ | فراخوانی آبشاری تابع عضو.

۱۰-۶ مدیریت دینامیکی حافظه با عملگرهای new و delete

زبان C++ به برنامه‌نویسان امکان داده تا بر نحوه اخذ و ترخیص حافظه در برنامه‌ها برای هر نوع داده توکار (built-in) یا تعریف شده توسط کاربر کنترل داشته باشند. به اینحالت *مدیریت دینامیکی حافظه* گفته می‌شود و توسط عملگرهای **new** و **delete** انجام می‌گردد. بخاطر دارید که کلاس **Employee** (شکل ۱۲-۱۰ و ۱۳-۱۰) از دو آرایه 25 کاراکتری برای عرضه نام و نام خانوادگی کارمند استفاده می‌کرد. تعریف کلاس **Employee** (شکل ۱۲-۱۰) بایستی تعداد عناصر در هر کدامیک از این آرایه‌ها را در زمان اعلان آنها بعنوان داده‌های عضو مشخص می‌کرد، چرا که سایز عضوهای داده میزان حافظه مورد نیاز برای ذخیره یک شی**Employee** را دیکته می‌کرد. همانطوری‌که گفتیم، این آرایه‌ها می‌توانند در برخورد با

اسامی کوچکتر از 24 کاراکتر، فضای حافظه را تلف کنند. همچنین اسامی بزرگتر از 24 کاراکتر بایستی به منظور قالب شدن در این آرایه‌ها با سایز مشخص شده، قطع کردند.

بهتر نیست از آرایه‌های استفاده کنیم که دقیقاً به تعداد عناصر مورد نیاز مبادرت به ذخیره نام و نام‌خانوادگی کارمندی می‌کنند؟ مدیریت دینامیکی حافظه امکان می‌دهد تا دقیقاً همین کار را انجام دهیم. همانطوری‌که در مثال بخش ۷-۱۰ خواهید دید، اگر اعضای داده آرایه **firstName** و **lastName** را با اشاره‌گرهای به **char** جایگزین سازیم، می‌توانیم از عملگر **new** برای اخذ دینامیکی (رزرو) حافظه به میزان دقیق و مورد نیاز برای نگهداری هر نام در زمان اجرا استفاده کنیم. مدیریت دینامیکی حافظه به این روش سبب ایجاد آرایه در فضای آزاد ذخیره‌سازی (غالباً *heap* نامیده می‌شود) می‌شود، ناحیه‌ای از حافظه تخصیص یافته به هر برنامه به منظور ذخیره‌سازی شی‌های ایجاد شده در زمان اجرا. زمانیکه حافظه برای یک آرایه در heap اخذ شد، می‌توانیم با اشاره دادن یک اشاره‌گر به اولین عنصر آرایه، به آن دست پیدا کنیم. زمانیکه دیگر نیازی به آرایه نداریم، می‌توانیم با استفاده از عملگر **delete** حافظه اخذ شده را آزاد کرده و به heap بازگردانیم. در صورت نیاز به حافظه می‌توانیم توسط عملگر **new** دوباره به آن دست پیدا کنیم.

مجدداً به سراغ کلاس **Employee** می‌رویم که به بررسی آن در بخش ۷-۱۰ خواهیم پرداخت. ابتدا، به بررسی جزئیات استفاده از عملگرهای **new** و **delete** در اخذ دینامیکی حافظه می‌پردازیم تا شی‌ها، نوع‌های بنیادین و آرایه‌ها را در آن مکان ذخیره سازیم.

به اعلان و عبارت زیر توجه کنید:

```
Time *timePtr;
timePtr = new Time;
```

عملگر **new** فضای با سایز مناسب برای یک شی از نوع **Time** اخذ می‌کند، سازنده پیش‌فرض برای مقداردهی اولیه شی فراخوانی شده و یک اشاره‌گر از نوع مشخص شده برگشت می‌یابد (یعنی یک ***Time**). توجه کنید که **new** می‌تواند برای اخذ دینامیکی هر نوع داده بنیادین (همانند **int** یا **double**) یا نوع کلاس بکار گرفته شود. اگر **new** قادر به یافتن فضای کافی در حافظه برای شی نباشد، با به راه انداختن یک استثناء نشان می‌دهد که خطائی رخ داده است. در فصل شانزدهم به بررسی استثناءها و رسیدگی به مشکلات رخ داده با **new** خواهیم پرداخت. در عمل نشان خواهیم داد که چگونه می‌توان یک استثناء رخ داده را گرفتار کرد. زمانیکه برنامه‌ای نتواند یک استثناء رخ داده را گرفتار سازد، بلافاصله خاتمه می‌یابد. برای حذف (آزاد کردن) حافظه اخذ شده توسط یک شی، از عملگر **delete** بصورت زیر استفاده می‌کنیم:

```
delete timePtr;
```

این عبارت ابتدا نابودکننده را بر روی شیی که **timePtr** به آن اشاره می‌کند فراخوانی کرده، سپس حافظه اخذ شده توسط آن شی را باز می‌گرداند. پس از اجرای این عبارت، حافظه برگشتی می‌تواند توسط سیستم در اختیار سایر شی‌ها قرار داده شود.

خطای برنامه‌نویسی

عدم رهاسازی حافظه اخذ شده دینامیکی در زمانیکه دیگر به آن نیازی نیست، می‌تواند سیستم را با مشکل «فقدان حافظه» مواجه سازد.

زبان C++ امکان تدارک دیدن یک مقداردهی کننده برای متغیرهای از نوع بنیادین جدیداً ایجاد شده می‌دهد، همانند

```
double *ptr = new double(3.14159);
```

در عبارت فوق، **double** ایجاد شده با 3.14159 مقداردهی اولیه شده و نتیجه به اشاره‌گر **ptr** تخصیص می‌یابد. از همین گرامر می‌توان در لیستی از آرگومان‌ها با کاماهای متمایز شده از یکدیگر در سازنده یک شی استفاده کرد. برای مثال،

```
Time *timePtr = new Time(12,45,0);
```

مبادرت به مقداردهی اولیه شی جدید **Time** با 12:45PM کرده و نتیجه به اشاره‌گر **timePtr** تخصیص می‌یابد.

همانطوری‌که قبلاً هم گفته شد، عملگر **new** می‌تواند در اخذ آرایه‌های دینامیکی بکارگرفته شود. برای مثال، یک آرایه 10 عضوی از نوع صحیح می‌تواند با عبارت زیر اخذ شده و به **gradeArray** تخصیص یابد:

```
int *gradesArray = new int[10];
```

اشاره‌گر **gradesArray** اعلان شده و آن به اشاره‌گری که به اولین عنصر از آرایه 10 عنصری از نوع صحیح که بصورت دینامیکی اخذ شده تخصیص داده می‌شود. بخاطر دارید که سایز یک آرایه باید در زمان کامپایل و با استفاده از یک ثابت صحیح مشخص شده باشد. با این وجود، سایز آرایه اخذ شده دینامیکی می‌تواند با استفاده از هر عبارت صحیح که می‌تواند در زمان اجرا ارزیابی گردد، تعیین شود. همچنین به این نکته توجه داشته باشید که به هنگام اخذ یک آرایه با شی‌های دینامیکی، برنامه نویس نمی‌تواند آرگومان‌هائی به هر سازنده شی ارسال کند. بجای آن، هر شی در آرایه با سازنده پیش‌فرض خود مقداردهی اولیه می‌شود. برای حذف آرایه اخذ شده دینامیکی که **gradesArray** به آن اشاره می‌کند، از عبارت زیر استفاده می‌کنیم.

```
delete[] gradesArray;
```

عبارت فوق حافظه اخذ شده توسط آرایه‌ای که **gradesArray** به آن اشاره می‌کند را آزاد می‌سازد. اگر اشاره‌گر فوق به آرایه‌ای از شی‌ها اشاره داشته باشد، ابتدا نابودکننده برای هر شی موجود در آرایه

فراخوانی می‌شود، سپس حافظه رها می‌گردد. اگر عبارت فوق فاقد براکت‌ها (**[]**) باشد و **gradesArray** به آرایه‌ای از شی‌ها اشاره داشته باشد، فقط اولین شی در آرایه انتخاب و نابود می‌شود.

خطای برنامه‌نویسی

*استفاده از **delete** بجای **[]delete** در ارتباط با آرایه‌ی از شی‌ها می‌تواند خطاهای زمان اجرا بدنبال داشته باشد.*

۷-۱۰ اعضای static کلاس

یک استثناء مهم در قانونی وجود دارد که می‌گوید هر شی از کلاس دارای یک کپی از تمام اعضای داده خود در کلاس است. در موارد خاصی، فقط یک کپی از یک متغیر باید توسط تمام شی‌های کلاس به اشتراک گذاشته شود. از عضو داده استاتیک به همین منظور و دلایل دیگر استفاده می‌شود. چنین متغیری نشاندهنده اطلاعات «در سطح کلاس» است (یعنی خصیصه‌ای از کلاس که مابین تمام نمونه‌ها به اشتراک گذاشته می‌شود، و نه خصیصه یک شی خاص از کلاس). اعلان یک عضو استاتیک با کلمه کلیدی **static** آغاز می‌شود. از نسخه‌های کلاس **GradeBook** در فصل هفتم بخاطر دارید که از اعضای داده استاتیک برای ذخیره‌سازی ثابت‌های نشاندهنده تعداد امتیازات (نمرات) که کلیه شی‌های **GradeBook** می‌توانند نگهداری کنند، استفاده کردیم.

اجازه دهید بحث را با مثالی که در ارتباط با داده استاتیکی و در سطح کلاس است، ادامه دهیم. فرض کنید که یک بازی ویدئوی با موضوع نبرد مریخی‌ها و دیگر مخلوقات فضایی داریم. هر مریخی مایل است تا شجاع بوده و راغب به حمله‌ور شدن به دیگر مخلوقات فضایی در مواقعی است که بداند در صحنه حداقل پنج مریخی دیگر حضور دارند. اگر کمتر از پنج مریخی در صحنه حضور داشته باشند، هر مریخی تبدیل به یک ترسو می‌شود.

از اینرو هر مریخی نیاز دارد تا از تعداد مریخی‌ها (**martianCount**) مطلع باشد. می‌توانیم به هر نمونه از کلاس **Martian** یک **martianCount** بعنوان یک عضو داده اعطا کنیم. اگر چنین کاری انجام دهیم، هر مریخی دارای یک کپی متمایز از عضو داده خواهد بود. هر زمان که یک مریخی جدید ایجاد کنیم، مجبور هستیم تا عضو داده **martianCount** در تمام شی‌های **Martian** را به روز کنیم. انجام اینکار مستلزم این است که هر شی **Martian** دارای یا دسترسی به، دستگیره‌ای به تمام دیگر شی‌های **Martian** در حافظه داشته باشد. انجام چنین کاری به معنی اتلاف حافظه با کپی‌های که افزونگی ایجاد می‌کنند بوده و زمان هم در این بین تلف می‌شود. بجای اینکار، **martianCount** را بصورت **static** اعلان می‌کنیم. چنین حالتی **martianCount** را به داده‌ای در سطح کلاس تبدیل می‌کند. هر مریخی می‌تواند در صورتیکه عضوی از **Martian** باشد، به **martianCount** دسترسی پیدا کند، و این در صورتی است که

فقط یک کپی از متغیر استاتیکی **martianCount** توسط C++ نگهداری می‌شود. با اینکار در فضای حافظه صرفه‌جویی می‌شود. با افزایش مقدار متغیر استاتیکی **martianCount** توسط سازنده **Martian** و کاستن از مقدار **martianCount** توسط نابودکننده **Martian** در زمان هم صرفه‌جویی می‌کنیم. به‌دلیل وجود یک کپی، مجبور نیستیم تا کپی‌های مجزا از **martianCount** را برای هر شی **Martian** افزایش یا کاهش دهیم.

اگر چه ممکن است اینحالت شبیه متغیرهای سراسری بنظر برسد، اما اعضای داده استاتیکی یک کلاس دارای *قلمرو کلاس* هستند. همچنین اعضای استاتیک می‌توانند بصورت **public**، **private** و **protected** اعلان شوند. یک عضو داده استاتیکی از نوع بنیادین بطور پیش‌فرض با صفر مقداردهی اولیه می‌شود. اگر بخواهید آنرا با مقدار دیگر مقداردهی کنید، عضو داده استاتیکی فقط یکبار مقداردهی اولیه خواهد شد. یک عضو داده استاتیکی ثابت از نوع **int** یا **enum** می‌تواند در اعلان خود در تعریف کلاس مقداردهی اولیه شود. با این همه، دیگر اعضای داده استاتیکی بایستی در قلمرو فایل تعریف شوند (خارج از بدنه تعریف کلاس) و فقط می‌تواند در تعریف آنها مقداردهی اولیه گردند. دقت کنید که اعضای داده استاتیک از نوع کلاس (شی‌های عضو استاتیک) که دارای سازنده‌های پیش‌فرض هستند نیازی به مقداردهی اولیه ندارند چرا که سازنده‌های پیش فرض برای آنها فراخوانی خواهند شد. اعضای استاتیک **private** و **protected** معمولاً از طریق توابع عضو **public** کلاس یا از طریق **friend** (دوستان) کلاس در دسترس قرار می‌گیرند. (در فصل دوازدهم، خواهید دید که اعضای استاتیکی **private** و **protected** می‌توانند از طریق توابع **protected** (محافظت شده) هم در دسترس قرار گیرند). اعضای استاتیکی یک کلاس حتی در زمانیکه شی‌های که از آن کلاس وجود ندارند، وجود دارند. برای دسترسی به یک عضو کلاس استاتیکی **public** در زمانیکه هیچ شی از آن کلاس وجود ندارد، کافیست پیشوند نام کلاس و عملگر باینری تفکیک قلمرو (::) در کنار نام عضو داده آورده شود. برای مثال، اگر متغیر **martianCount** سراسری (public) باشد، می‌توان از طریق عبارت **Martian::martianCount** در زمانیکه هیچ شی از **Martian** وجود ندارد، به آن دسترسی پیدا کرد.

همچنین می‌توان به اعضای کلاس استاتیکی **public** از طریق هر شی از آن کلاس با استفاده از نام شی، عملگر نقطه و نام عضو دسترسی پیدا کرد (مثلاً **MyMartian.martianCount**). برای دسترسی به یک عضو کلاس استاتیک **private** یا **protected** در زمانیکه شی وجود ندارد، یک تابع عضو استاتیک تدارک دیده و تابع با پیشوند نام خود به همراه نام کلاس و عملگر تفکیک قلمرو فراخوانی می‌شود. یک تابع عضو استاتیک سرویسی برای کلاس می‌باشد و نه یک شی خاص از آن کلاس.

برنامه شکل‌های ۲۱-۱۰ الی ۲۳-۱۰ به بیان یک داده عضو استاتیکی خصوصی بنام **count** (شکل ۲۱-۱۰، خط 21) و یک تابع عضو استاتیک سراسری بنام **getCount** (شکل ۲۱-۱۰، خط 15) می‌پردازد. در شکل ۲۲-۱۰، خط 14 مبادرت به تعریف و مقداردهی اولیه داده عضو **count** با صفر در قلمرو فایل کرده و خطوط 21-18 تابع عضو استاتیک **getCount** را تعریف کرده‌اند. توجه کنید که خواه خط 14 یا خط 18 حاوی کلمه کلیدی **static** باشند یا نباشند، هنوز هم هر دو خط به اعضای کلاس استاتیک اشاره دارند. زمانیکه **static** بر روی یک ایتم در قلمرو فایل اعمال می‌شود، آن ایتم فقط در آن فایل شناخته خواهد شد. نیاز است تا اعضای استاتیک یک کلاس از طریق کد هر سرویس‌گیرنده‌ای که به فایل دسترسی دارند، در اختیار آنها قرار داشته باشند، از اینرو نمی‌توانیم آنها را در فایل **cpp.** بصورت **static** اعلان کنیم، فقط می‌توانیم آنها را در فایل **h.** بصورت **static** اعلان نمائیم. عضو داده **count** شمارنده‌ای از تعداد شی‌های کلاس **Employee** است که نمونه‌سازی شده‌اند. زمانیکه شی‌های از کلاس **Employee** وجود دارند، عضو **count** می‌تواند از طریق هر تابع عضو از یک شی **Employee** مورد مراجعه قرار گیرد. در شکل ۲۲-۱۰، **count** توسط هر دو خط 33 در سازنده و خط 48 در نابودکننده مورد مراجعه قرار می‌گیرد. همچنین توجه کنید از آنجا که **count** یک **int** است، می‌توانست در فایل سرآیند در خط 21 از شکل ۲۱-۱۰ مقداردهی اولیه گردد.

خطای برنامه‌نویسی

*قرار دادن کلمه کلیدی **static** در تعریف اعضای داده استاتیکی در قلمرو فایل، خطای کامپایل است.*

در شکل ۲۲-۱۰ به نحوه استفاده از عملگر **new** (خطوط 27 و 30) در سازنده **Employee** به منظور اخذ دینامیکی حافظه به میزان مورد نیاز برای اعضای **firstName** و **lastName** توجه کنید. اگر عملگر **new** قادر به اخذ فضای مورد تقاضا از حافظه برای یک یا هر دو این آرایه‌ها نباشد، بلافاصله برنامه خاتمه می‌یابد. در فصل شانزدهم مکانیزم بهتری برای مواجه شدن با چنین وضعیت‌های در نظر خواهیم گرفت.

```
1  // Fig. 10.21: Employee.h
2  // Employee class definition.
3  #ifndef EMPLOYEE_H
4  #define EMPLOYEE_H
5
6  class Employee
7  {
8  public:
9     Employee( const char * const, const char * const ); // constructor
10    ~Employee(); // destructor
11    const char *getFirstName() const; // return first name
12    const char *getLastName() const; // return last name
13
14    // static member function
15    static int getCount(); // return number of objects instantiated
16 private:
17    char *firstName;
18    char *lastName;
19
20    // static data
```

```
21    static int count; // number of objects instantiated
22 }; // end class Employee
23
24 #endif
```

شکل ۲۱-۱۰ | تعریف کلاس Employee با عضو داده استاتیک برای ردگیری تعداد شی‌های Employee در حافظه.

در شکل ۲۲-۱۰ به پیاده سازی توابع **getFirstName** (خطوط 58-52) و **getLastName** (خطوط 61-67) که اشاره‌گرهای به داده کاراکتری ثابت (**const**) برگشت می‌دهند، دقت کنید.

```
1  // Fig. 10.22: Employee.cpp
2  // Member-function definitions for class Employee.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstring> // strlen and strcpy prototypes
8  using std::strlen;
9  using std::strcpy;
10
11 #include "Employee.h" // Employee class definition
12
13 // define and initialize static data member at file scope
14 int Employee::count = 0;
15
16 // define static member function that returns number of
17 // Employee objects instantiated (declared static in Employee.h)
18 int Employee::getCount()
19 {
20    return count;
21 } // end static function getCount
22
23 // constructor dynamically allocates space for first and last name and
24 // uses strcpy to copy first and last names into the object
25 Employee::Employee( const char * const first, const char * const last )
26 {
27    firstName = new char[ strlen( first ) + 1 ];
28    strcpy( firstName, first );
29
30    lastName = new char[ strlen( last ) + 1 ];
31    strcpy( lastName, last );
32
33    count++; // increment static count of employees
34
35    cout << "Employee constructor for " << firstName
36       << ' ' << lastName << " called." << endl;
37 } // end Employee constructor
38
39 // destructor deallocates dynamically allocated memory
40 Employee::~Employee()
41 {
42    cout << "~Employee() called for " << firstName
43       << ' ' << lastName << endl;
44
45    delete [] firstName; // release memory
46    delete [] lastName; // release memory
47
48    count--; // decrement static count of employees
49 } // end ~Employee destructor
50
51 // return first name of employee
52 const char *Employee::getFirstName() const
53 {
54    // const before return type prevents client from modifying
55    // private data; client should copy returned string before
56    // destructor deletes storage to prevent undefined pointer
57    return firstName;
```

```
58 } // end function getFirstName
59
60 // return last name of employee
61 const char *Employee::getLastName() const
62 {
63    // const before return type prevents client from modifying
64    // private data; client should copy returned string before
65    // destructor deletes storage to prevent undefined pointer
66    return lastName;
67 } // end function getLastName
```

شکل ۲۲-۱۰ | تعریف تابع عضو کلاس Employee.

در این پیاده‌سازی، اگر سرویس‌گیرنده مایل به نگهداری یک کپی از نام و یا نام خانوادگی داشته باشد، سرویس‌گیرنده مسئول کپی کردن حافظه اخذ شده دینامیکی در شی **Employee** پس از بدست آوردن اشاره‌گر به داده کاراکتری ثابت از شی است. همچنین امکان پیاده‌سازی **getFirstName** و **getLastName** وجود دارد، از اینرو سرویس‌گیرنده مستلزم ارسال آرایه کاراکتری و سایز آن به هر تابع است. پس توابع می‌توانند نام یا نام خانوادگی را به آرایه کاراکتری تدارک دیده شده توسط سرویس‌گیرنده کپی کنند. توجه کنید که می‌توانیم از کلاس **string** برای برگشت دادن کپی از یک شی رشته به فراخوان به جای برگشت دادن یک اشاره‌گر به داده خصوصی استفاده کنیم.

در شکل ۳۳-۱۰ از تابع عضو استاتیک **getCount** برای تعیین تعداد شی‌های **Employee** موجود استفاده شده است. توجه کنید زمانیکه هیچ شیء در برنامه ایجاد نشده باشد، فراخوانی تابع **Employee::getCount()** صورت می‌گیرد (خط 14 و 38). با این وجود زمانیکه شی‌هائی ایجاد شده باشند، تابع **getCount** می‌تواند از طریق شی‌ها، همانند عبارت موجود در خطوط 23-22 که از اشاره‌گر **e1Ptr** برای فراخوانی تابع **getCount** استفاده شده، فراخوانی گردد. در خط 23 به نحوه استفاده از **e2Ptr->getCount()** یا **Employee::getCount()** توجه کنید که هر دو نتیجه مشابهی بدست می‌دهند چرا که getCount همیشه به همان عضو استاتیک **count** دسترسی دارد.

```
1  // Fig. 10.23: fig10_23.cpp
2  // Driver to test class Employee.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Employee.h" // Employee class definition
8
9  int main()
10 {
11    // use class name and binary scope resolution operator to
12    // access static number function getCount
13    cout << "Number of employees before instantiation of any objects is "
14       << Employee::getCount() << endl; // use class name
15
16    // use new to dynamically create two new Employees
17    // operator new also calls the object's constructor
18    Employee *e1Ptr = new Employee( "Susan", "Baker" );
19    Employee *e2Ptr = new Employee( "Robert", "Jones" );
20
21    // call getCount on first Employee object
22    cout << "Number of employees after objects are instantiated is "
23       << e1Ptr->getCount();
```

```
24
25    cout << "\n\nEmployee 1: "
26       << e1Ptr->getFirstName() << " " << e1Ptr->getLastName()
27       << "\nEmployee 2: "
28       << e2Ptr->getFirstName() << " " << e2Ptr->getLastName() << "\n\n";
29
30    delete e1Ptr; // deallocate memory
31    e1Ptr = 0; // disconnect pointer from free-store space
32    delete e2Ptr; // deallocate memory
33    e2Ptr = 0; // disconnect pointer from free-store space
34
35    // no objects exist, so call static member function getCount again
36    // using the class name and the binary scope resolution operator
37    cout << "Number of employees after objects are deleted is "
38       << Employee::getCount() << endl;
39    return 0;
40 } // end main
```

```
Number of employees before instantiation of any object is 0
Employee constructor for Susan Baker called.
Employee constructor for Bahram Jones called.
Number of Employee after objects are instantiated is 2

Employee 1: Susan Barker
Employee 2: Robert Jones

~ Employee() called for Susan Baker
~ Employee() called for Robert Jones
Number of Employee after objects are deleted is 0
```

شکل ۲۳-۱۰ |عضو داده استاتیک تعداد شی‌های موجود در کلاس را روگیری می‌کند.

اگر تابع عضو به اعضای داده غیراستاتیک یا توابع عضو غیراستاتیک یک کلاس دسترسی ندارد، باید بصورت **static** اعلان شود. برخلاف توابع عضو غیراستاتیک، یک تابع عضو استاتیک دارای اشاره‌گر **this** نمی‌باشد، چرا که اعضای داده استاتیک و توابع عضو استاتیک بصورت مستقل از هر شی کلاس می‌باشند. بایستی اشاره‌گر **this** به یک شی خاص کلاس اشاره داشته باشد و زمانیکه یک تابع عضو استاتیک فراخوانی می‌شود، امکان وجود هر شی از آن کلاس در حافظه وجود ندارد.

خطای برنامه‌نویسی

*استفاده از اشاره‌گر **this** در یک تابع استاتیک خطای کامپایل بدنبال خواهد داشت.*

در خطوط 18-19 از شکل ۲۳-۱۰ از عملگر **new** برای اخذ دینامیکی حافظه برای دو شی **Employee** استفاده شده است. بخاطر دارید که اگر برنامه قادر به اخذ حافظه برای یک یا هر دو این شی‌ها نشود، بلافاصله پایان می‌یابد. پس از اخذ حافظه برای هر شی **Employee**، سازنده آن فراخوانی می‌شود. زمانیکه از **delete** در خطوط 30 و 32 برای بازپس‌گیری حافظه اخذ شده توسط دو شی **Employee** استفاده می‌شود، نابودکننده این شی‌ها فراخوانی می‌گردند.

۸-۱۰ انتزاع داده و پنهان سازی اطلاعات

۱-۸-۱۰ مثال: نوع داده انتزاعی آرایه

در فصل هفتم به بررسی آرایه‌ها پرداختیم. همانطوری که در آنجا توضیح دادیم، آرایه چیزی بیش از یک اشاره‌گر و مقداری فضا در حافظه نیست. در صورتیکه برنامه‌نویس متوجه باشد می‌تواند از این قابلیت اولیه

به هنگام کار بر روی آرایه استفاده کند. عملیات‌های متعددی می‌توان بر روی آرایه‌ها انجام داده، اما همه آنها بصورت توکار در C++ تعبیه نشده‌اند. با کلاس‌های C++، برنامه‌نویس می‌تواند یک آرایه ADT توسعه دهد که به آرایه‌های اولیه (خام) ترجیح داده می‌شوند. کلاس آرایه می‌تواند قابلیت‌های جدیدی داشته و آنها را در اختیار کاربر خود قرار دهد، همانند:

- بررسی محدودهٔ شاخص
- محدودهٔ اختیاری شاخص بجای شروع شدن از صفر
- تخصیص آرایه
- مقایسه آرایه
- ورودی/خروجی آرایه
- آرایه‌های که از سایز خود مطلع باشند
- آرایه‌های که بصورت دینامیکی خود را با عناصر بیشتر تطبیق می‌دهند
- آرایه‌های که می‌توانند از خود بصورت مرتب در فرمت جدولی چاپ بگیرند.

در فصل یازدهم، چنین کلاس آرایه‌ای را با قابلیت‌های فوق ایجاد خواهیم کرد. بخاطر دارید که کلاس الگوی **vector** از کتابخانه استاندارد C++ (فصل هفتم) برخی از این قابلیت‌ها را به همان اندازه فراهم می‌کرد.

۱۰-۸-۲ مثال: نوع داده انتزاعی رشته

زبان C++ عمداً یک زبان پراکنده است که برای برنامه‌نویسان قابلیت‌های خام در نظر گرفته تا برنامه‌نویسان برحسب نیاز از آنها بعنوان ابزار در ایجاد سیستم‌های عریض و طویل استفاده کنند. زبان برای کاستن از هزینه کارایی طراحی شده است. زبان C++ مناسب برای برنامه‌نویسی کاربردی و برنامه‌نویسی سیستم است. مطمئناً، امکان قراردادن نوع داده رشته در میان انواع داده توکار C++ وجود داشت. بجای اینکار، زبان برای در برگرفتن مکانیزمی برای ایجاد و پیاده‌سازی نوع داده انتزاعی رشته از طریق کلاس‌ها طراحی شده است. در فصل سوم به معرفی کلاس **string** از کتابخانه C++ پرداختیم و در فصل یازدهم مبادرت به ایجاد رشته ADT متعلق بخودمان خواهیم‌کرد. در فصل هیجدهم، کلاس **string** به تفصیل توضیح داده شده است.

۱۰-۹ کلاس‌های حامل و تکرارشونده‌ها

در میان انواع کلاس‌های پرطرفدار، *کلاس‌های حامل* از جایگاه خاصی برخوردارند. به این کلاس‌ها گاهاً *کلاس‌های کلکسیون* هم گفته می‌شود، یعنی کلاس‌های که برای نگهداری کلکسیونی از شی‌ها طراحی شده‌اند. معمولاً کلاس‌های حامل سرویس‌های همانند درج، حذف، جستجو، مرتب‌سازی و تست یک

ایتم برای تعیین اینکه آیا عضوی از کلکسیون است یا خیر، ارائه می‌دهند. آرایه‌ها، پشته‌ها، صف‌ها، درخت‌ها و لیست‌های پیوندی نمونه‌های از کلاس‌های حامل هستند. در فصل هفتم در ارتباط با آرایه مطالبی آموختیم و در فصل ۲۱ به بررسی ساختمان‌های داده دیگر خواهیم پرداخت.

شریک دانستن شی‌های تکرار شونده با کلاس‌های حامل معقول به نظر می‌رسد. یک تکرار شونده، شی است که در میان یک کلکسیون «قدم» می‌زند، و ایتم بعدی را برگشت می‌دهد (یا عملیاتی بر روی ایتم بعدی انجام می‌دهد). زمانیکه یک تکرار شونده برای کلاسی در نظر گرفته می‌شود، بدست آوردن عنصر بعدی از آن کلاس کار آسانی می‌شود. یک کلاس حامل می‌تواند چندین تکرار شونده داشته باشد. هر تکرار شونده مسئول نگهداری اطلاعات موقعیت خود است.

۱۰-۱۰ کلاس‌های پروکسی

بخاطر دارید که دو جنبه و قاعده یک مهندسی نرم‌افزار ایده‌ال عبارت بودند از جداسازی واسط از پیاده‌سازی و پنهان‌سازی جزئیات پیاده‌سازی. برای برآورده کردن این اهداف سعی می‌کنیم تا کلاس را در یک فایل سرآیند تعریف کنیم و توابع عضو آنرا در یک فایل پیاده‌سازی کننده دیگر پیاده نمائیم. با این همه، همانطوری که در فصل نهم اشاره کردیم، فایلهای سرآیند حاوی بخش‌های از پیاده‌سازی کلاس هستند و تا حدودی به دیگران هم اشاره دارند. برای مثال اعضای خصوصی یک کلاس در تعریف کلاس در فایل سرآیند لیست می‌شوند، از اینرو این اعضا در دید سرویس‌گیرندگان قرار دارند، حتی اگر سرویس‌گیرنده‌ها به اعضای خصوصی دسترسی نداشته باشند. آشکار کردن داده خصوصی یک کلاس به این روش سبب می‌شود تا اطلاعات اختصاصی در معرض دید سرویس‌گیرنده‌های کلاس قرار گیرد. حال به معرفی نظریه کلاس پروکسی می‌پردازیم که اجازه می‌دهد تا حتی داده‌های خصوصی کلاس را از دید سرویس‌گیرنده‌های کلاس پنهان سازیم. مشروط بر اینکه سرویس‌گیرنده‌های کلاس با یک کلاس پروکسی مطلع باشند که فقط واسط سراسری به کلاس قادر به ارائه سرویس به سرویس گیرنده‌ها است، بدون اینکه سرویس‌گیرنده‌ها قادر به دسترسی به جزئیات پیاده سازی کلاس باشند.

پیاده‌سازی یک کلاس پروکسی مستلزم چندین مرحله است که در شکل‌های ۲۴-۱۰ الی ۲۷-۱۰ با مثال بیان شده است. ابتدا، تعریف کلاس را انجام می‌دهیم که حاوی پیاده‌سازی اختصاصی است و مایل هستیم تا آنرا پنهان نگه داریم. کلاس ما در این مثال، **Implementation** نام دارد و در شکل ۲۴-۱۰ نشان داده شده است. کلاس پروکسی **Interface** در شکل‌های ۲۵-۱۰ و ۲۶-۱۰ آورده شده است. برنامه تست و خروجی نمونه در شکل ۲۷-۱۰ دیده می‌شود.

کلاس **Implementation** (شکل ۲۴-۱۰) حاوی یک عضو داده خصوصی بنام value (داده‌ای که می‌خواهیم آنرا از دید سرویس‌گیرنده‌ها پنهان سازیم)، یک سازنده برای مقداردهی اولیه **value** و توابع **setValue** و **getValue** است.

یک کلاس پروکسی بنام **Interface** (شکل ۲۵-۱۰) با یک واسط سراسری **public** مشابه (بجز در اسامی سازنده و نابودکننده) برای کلاس **Implementation** تعریف کرده‌ایم.

```
1  // Fig. 10.24: Implementation.h
2  // Header file for class Implementation
3
4  class Implementation
5  {
6  public:
7     // constructor
8     Implementation( int v )
9        : value( v ) // initialize value with v
10    {
11       // empty body
12    } // end constructor Implementation
13
14    // set value to v
15    void setValue( int v )
16    {
17       value = v; // should validate v
18    } // end function setValue
19
20    // return value
21    int getValue() const
22    {
23       return value;
24    } // end function getValue
25 private:
26    int value; // data that we would like to hide from the client
27 }; // end class Implementation
```

شکل ۲۴-۱۰ | تعریف کلاس Implementation.

```
1  // Fig. 10.25: Interface.h
2  // Header file for class Interface
3  // Client sees this source code, but the source code does not reveal
4  // the data layout of class Implementation.
5
6  class Implementation; // forward class declaration required by line 17
7
8  class Interface
9  {
10 public:
11    Interface( int ); // constructor
12    void setValue( int ); // same public interface as
13    int getValue() const; // class Implementation has
14    ~Interface(); // destructor
15 private:
16    // requires previous forward declaration (line 6)
17    Implementation *ptr;
18 }; // end class Interface
```

شکل ۲۵-۱۰ | تعریف کلاس Interface.

تنها عضو خصوص کلاس پروکسی یک اشاره‌گر به شی از کلاس **Implementation** است. با استفاده از اشاره‌گر به این روش می‌توانیم جزئیات پیاده‌سازی کلاس **Implementation** را از دید سرویس‌گیرنده پنهان نگه داریم. توجه کنید که تنها اشاره‌ای که در کلاس **Interface** به کلاس اختصاصی

۲۹۰ فصل دهم ـــــــ

شکل ۲۷-۱۰ کلاس **Interface** را تست می‌کنید. دقت کنید که فقط فایل سرآیند برای **Interface** در کد سرویس گیرنده شامل شده است (خط 7)، در اینجا هیچ ذکری از وجود یک کلاس مجزا بنام **Implementation** نیست. از اینرو، سرویس‌گیرنده هرگز داده خصوصی کلاس **Implementation** را نخواهد دید.

مهندسی نرم‌افزار

کلاس پروکسی کد سرویس‌گیرنده را از تغییرات پیاده‌سازی دور نگه می‌دارد.

```
1  // Fig. 10.27: fig10_27.cpp
2  // Hiding a class's private data with a proxy class.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Interface.h" // Interface class definition
8
9  int main()
10 {
11    Interface i( 5 ); // create Interface object
12
13    cout << "Interface contains: " << i.getValue()
14       << " before setValue" << endl;
15
16    i.setValue( 10 );
17
18    cout << "Interface contains: " << i.getValue()
19       << " after setValue" << endl;
20    return 0;
21 } // end main
```

```
Interface contains: 5 before setValue
Interface contains: 10 after setValue
```

شکل ۲۷-۱۰ | پیاده‌سازی کلاس پروکسی.

خود آزمایی

۱-۱۰ جاهای خالی را با کلمات مناسب پر کنید.

a) بایستی از _____ برای مقداردهی اولیه عضوهای ثابت یک کلاس استفاده کرد.

b) یک تابع غیر عضو باید بصورت یک _____ برای کلاسی اعلان شود که دارای دسترسی به اعضای داده خصوصی کلاس است.

c) عملگر _____ بصورت اتوماتیک حافظه دینامیکی برای شی از نوع مشخص شده اخذ کرده و یک _____ به آن نوع برگشت می‌دهد.

d) یک شی ثابت بایستی _____ شود، پس از ایجاد شدن مقدار آن قابل تغییر نخواهد بود.

e) یک عضو داده _____ نشانه اطلاعات در سطح کلاس است.

f) توابع عضو غیر استاتیک دارای دسترسی اشاره‌گر به خود هستند که اشاره‌گر _____ نامیده می‌شود.

۲۹۲فصل دهم ـــــــ

۱۰-۳ به مقایسه اخذ حافظه دینامیکی و باز پس‌گیری آن توسط عملگرهای new، new[]، delete و delete[] بپردازید.

۱۰-۴ مفهوم دوستی را در C++ توضیح دهید. به بررسی جنبه‌های منفی رابطه دوستی هم بپردازید.

۱۰-۵ آیا می‌توان تعریف کلاس Time را به نحوی اصلاح کرد که در برگیرنده هر دو سازنده زیر باشد؟ اگر پاسخ منفی است، توضیح دهید چرا نمی‌توان اینکار را انجام داد.

```
Time ( int h = 0, int m = 0, int s= 0 ) ;
Time();
```

۱۰-۶ در صورت‌ی‌که نوع برگشتی حتی از نوع void برای یک سازنده یا نابود کننده مشخص شود، چه اتفاقی خواهد افتاد؟

۱۰-۷ کلاس Date بکار رفته در شکل ۱۰-۱۰ را برای داشتن قابلیت‌های زیر تغییر دهید:

a) خروجی تاریخ در فرمت‌های مختلف همانند

DDD YYYY
MM/DD/YY
June 14,1992

b) استفاده از سازنده‌های سربارگذاری شده برای ایجاد شی‌های Date مقداردهی شده با فرمت‌های تاریخی مطرح شده در بخش (a).

c) ایجاد سازنده Date که تاریخ سیستم را با استفاده از توابع کتابخانه استاندارد <ctime> خوانده و اعضای Date را تنظیم کند.

۱۰-۸ کلاس SavingsAccaunt را ایجاد کنید. از یک عضو داده استاتیکی بنام annualInterestRate برای نرخ سود سالیانه برای هر سپرده استفاده کنید. هر عضو از کلاس حاوی یک عضو داده private بنام savingsBalance است که دلالت بر میزان پس انداز جاری در سپرده دارد. تابع عضو calculateMonthlyInterest را در نظر بگیرید که سود ماهانه را با ضرب موجودی (balance) در annualInterestRate تقسیم بر 12 بدست می‌آورد، این سود باید به savingsBalance افزوده شود. یک تابع عضو استاتیک بنام modifyInterestRate در نظر بگیرید که مقدار annualInterestRate را با یک مقدار جدید تنظیم کند. برنامه راه اندازی برای تست کلاس SavingsAccount بنویسید. دو شی نمونه سازی شده از کلاس SavingsAccount بنام‌های saver2 و saver1 ایجاد کنید. موجودی saver1 را با $2000.00 و saver2 را با $3000.00 تنظیم کنید. مقدار annualInterestRate را با 3 درصد تنظیم نمائید. سپس نرخ سود ماهانه را محاسبه و موجودی جدید را برای هر پس‌انداز چاپ کنید. سپس نرخ سود سالانه annualInterestRate را با 4 درصد تنظیم کرده، سود ماه بعد را محاسبه و میزان موجودی‌های جدید را چاپ نمائید.