فصل ششم

توابع و مکانیزم بازگشتی

اهداف

- ایجاد مدولار برنامه‌ها با بخش‌هایی بنام توابع.
- استفاده از توابع ریاضی موجود در کتابخانه استاندارد C++.
- ایجاد توابع با پارامترهای مضاعف.
- آشنایی با مکانیزم‌های ارسال اطلاعات مابین توابع و برگشت نتایج.
- آشنایی با مکانیزم فراخوانی و برگشت توابع.
- تکنیک‌های شبیه‌سازی بکار رفته در ایجاد اعداد تصادفی.
- آشنایی با مبحث قلمرو.
- آشنایی با نحوه عملکرد و نوشتن توابعی بازگشتی.

رئوس مطالب

٦-١ مقدمه

بیشتر برنامه‌های کامپیوتری که مسائل دنیای واقعی را برطرف می‌کنند به نسبت برنامه‌هایی که در چند فصل آغازین ارائه شدند، بسیار بزرگتر و پیچیده‌تر هستند. تجربه نشان داده بهترین روش برای ساخت و نگهداری یک برنامه بزرگ این است که برنامه به قسمت‌های کوچکتر تقسیم شود به نحوی که هر قسمت وظیفه خاصی داشته باشد. در اینحالت می‌توان بطرز شایسته‌ای بر برنامه مدیریت داشت. این روش به نام

روش تقسیم و غلبه (divide and conquer) معروف است. در این فصل با روش‌هایی آشنا خواهید شد که در آن طراحی، تکمیل، اجرا و نگهداری برنامه‌های بزرگ به آسانی صورت می‌گیرد.

به بررسی بخشی از توابع ریاضی کتابخانه استاندارد C++،که برخی از آنها به بیش از یک پارامتر نیاز دارند، می‌پردازیم. سپس با نحوه اعلان یک تابع با بیش از یک پارامتر آشنا خواهید شد. همچنین اطلاعات بیشتری در مورد نمونه‌های اولیه تابع بدست آورده و به بررسی نحوه عملکرد کامپایلر در تبدیل نوع آرگومان تابع فراخوانی شده به نوع پارامترهای تابع می‌پردازیم.

سپس، به بررسی مختصر تکنیک‌های شبیه‌سازی با اعداد تصادفی پرداخته و نسخه‌ای از بازی پرتاب تاس بنام craps را ایجاد می‌کنیم. در این برنامه از اغلب تکنیک‌های برنامه‌نویسی که تا بدین‌جا آموخته‌اید استفاده شده است.

در ادامه، به معرفی کلاس‌ها و قوانین قلمرو در C++ می‌پردازیم. قانون قلمرو، تعیین کننده، مدت زمانی است که در طی آن یک شی در حافظه وجود دارد و نیز شناسه‌ای است که در برنامه می‌تواند مورد مراجعه قرار گیرد. همچنین خواهید آموخت که چگونه C++ می‌تواند تابع در حال اجرا را ردگیری کند، نحوه نگهداری پارامترها و سایر متغیرهای محلی در درون حافظه و مدیریت آنها را چگونه انجام می‌دهد، و چگونه یک تابع پس از کامل شدن اجرا می‌داند که به چه مکانی باید بازگردد. در ادامه به بررسی دو مبحثی که به بهبود کارایی برنامه کمک می‌کنند، خواهیم پرداخت، توابع inline که می‌توانند سربارگذاری فراخوانی تابع را حذف سازند و پارامترهای مراجعه که می‌توانند برای ارسال ایتم‌های بزرگ داده‌ی به توابع مورد استفاده قرار گیرند تا کارائی تابع افزایش یابد.

امکان دارد در تعدادی از برنامه‌ها از چند تابع همنام استفاده کنید. این تکنیک، سربارگذاری تابع نامیده می‌شود، و از طرف برنامه‌نویسان برای پیاده‌سازی توابعی که وظایف مشابهی را با آرگومان‌هایی از نوع‌های متفاوت یا با تعداد متفاوتی از آرگومان‌ها انجام می‌دهند بکار گرفته می‌شود. به بررسی الگوهای تابع هم خواهیم پرداخت. الگوی تابع، مکانیزمی برای تعریف خانواده‌ای از توابع سربارگذاری شده است. در بخش پایانی این فصل به بررسی توابعی که خود را فراخوانی می‌کنند، چه بصورت مستقیم یا غیرمستقیم (از طریق تابع دیگر) پرداخته شده است، مبحثی که بعنوان بازگشتی شناخته می‌شود و بطور کاملتر در دوره‌های بالاتر علوم کامپیوتر توضیح داده می‌شود.

۶-۲ کامپونت‌های برنامه در C++

برنامه‌های C++ متشکل از قسمت‌های متعددی از جمله توابع و کلاس‌ها هستند. برنامه‌نویس اقدام به ایجاد و ترکیب توابع و کلاس‌های جدید با کلاس‌های از قبل آماده شده موجود در کتابخانه استاندارد C++ می‌کند. در این فصل، تمرکز ما بر روی توابع است.

کتابخانه استاندارد C++ حاوی کلکسیون با ارزشی از توابع به منظور انجام محاسبات ریاضی، کار با رشته‌ها، کار با کاراکترها، عملیات ورودی/خروجی، تست و بررسی خطا و بسیاری از کاربردهای مناسب دیگر است. این کتابخانه بدلیل تدارک دیدن نیازهای متعدد یک برنامه‌نویس، کار برنامه‌نویس را آسانتر می‌کند. توابع کتابخانه استاندارد C++ بعنوان بخشی از محیط برنامه‌نویسی C++ تدارک دیده شده‌اند.

مهندسی نرم‌افزار

سعی کنید با کلکسیون با ارزش کلاس‌ها و توابع موجود در کتابخانه استاندارد C++ حتماً آشنا شوید.

مهندسی نرم‌افزار

برای ارتقای قابلیت استفاده مجدد از نرم افزار، هر تابع باید مختص انجام یک کار در نظر گرفته شده باشد، وظیفه آن بخوبی تعریف شده باشد، و نام تابع باید بطور موثر بیان‌کننده وظیفه تابع باشد. چنین توابعی کار نوشتن، تست، دیباگ و نگهداری برنامه ها را آسانتر می‌کنند.

اجتناب از خطا

تست و خطایابی یک تابع کوچک که یک وظیفه را انجام می‌دهد به نسبت یک تابع بزرگتر که وظایف متعددی دارد، راحتر است.

مهندسی نرم‌افزار

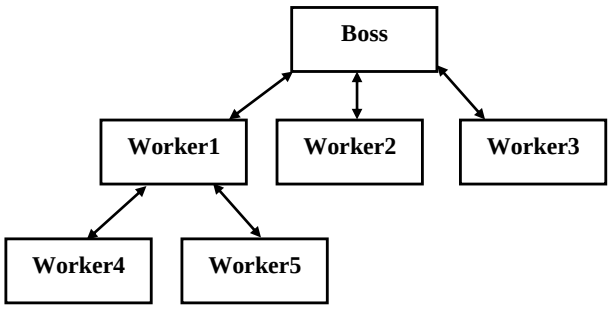
اگر نمی‌توانید نام دقیق و مناسبی برای کاری که تابع انجام می‌دهد انتخاب کنید، احتمالا تابع بیش از یک وظیفه بر عهده دارد. بهتر است چنین توابعی را به توابع کوچکتر تبدیل کرد.

اگر چه کتابخانه استاندار C++ توابع متعددی در نظر گرفته که وظایف زیادی به انجام می‌رسانند، اما نمی‌تواند هر آنچه را که یک برنامه‌نویس به آن نیاز دارد، در اختیار وی قرار دهد، از اینرو به برنامه‌نویسان امکان داده شده تا توابع متعلق به خود را ایجاد کنند تا نیازهای آنها را در حل مسائل خاص برطرف سازند. به این نوع از توابع، *توابع تعریف شده توسط کاربر* یا *توابع تعریف شده توسط برنامه‌نویس* گفته می‌شود. برنامه‌نویسان با نوشتن توابع قصد تعریف وظایف مشخص در یک برنامه را دارند و ممکن است از آنها در طول اجرای برنامه چندین بار استفاده کنند. اگر چه یک تابع ممکن است در چندین نقطه برنامه به دفعات بکار گرفته شود، اما عبارات تابع فقط یک بار نوشته می‌شوند.

به چند دلیل برای مدولار کردن یک برنامه از توابع استفاده می‌شود. در روش تقسیم و غلبه مدیریت توسعه برنامه بسیار بهتر صورت می‌گیرد. دلیل دیگر *استفاده مجدد از نرم‌افزار* است (استفاده دوباره از تابع موجود در ایجاد بلوک‌های یک برنامه جدید). در صورتیکه از تابع مناسب و قابل اعتماد بجای نوشتن کد متعلق بخود استفاده شود، یک برنامه با توابعی استاندارد بدست می‌آید. برای مثال، مجبور نیستیم تا نحوه خواندن یک رشته

متنی از صفحه‌کلید را تعریف کنیم، چراکه C++ دارای تابعی بنام **getline** در فایل سرآیند <**string**> به همین منظور است. سومین دلیل در استفاده از تابع، اجتناب از نوشتن کد تکراری در برنامه است. کدی که بعنوان یک تابع نوشته می‌شود، از این قابلیت برخوردار است که از مکان‌های مختلف برنامه فراخوانی شده و اجرا گردد.

یک تابع با فراخوانی فعال می‌شود و وظیفه‌ای که به آن منظور طراحی شده است به اجرا در می‌آورد. فراخوانی یک تابع مستلزم تهیه نام تابع و اطلاعات مورد نیاز تابع (آرگومان‌ها) از سوی فراخواننده تابع است. هنگامی که تابع وظیفه خود را به اتمام رساند، کنترل را به فراخواننده باز می‌گرداند (فراخواننده تابع). گاهی اوقات، تابع می‌تواند نتیجه‌ای به فراخواننده خود نیز برگشت دهد. سلسله مراتب مدیریت شباهت زیادی به روش عملکرد توابع دارد. بدین ترتیب که کارفرما (Boss) که نقش فراخواننده را دارد، از یک کارگر (worker) که نقش فراخوانده شده را دارد، می‌خواهد کاری انجام داده و نتیجه را پس از پایان کار گزارش دهد. کارفرما اطلاعی از اینکه کدام کارگر عمل درخواست شده را انجام می‌دهد ندارد، چرا که ممکن است کارگر فراخوانده شده، کارگرهای دیگری را فرا بخواند در حالیکه کارفرما از این مسائل اطلاعی ندارد. بزودی، نشان خواهیم داد که چگونه این روش *پنهان کردن جزئیات پیاده سازی* نقش مناسبی در مهندسی نرم‌افزار ایفا می‌کند. در شکل ۱-۶ رابطه تابع **Boss** با توابعی کارگر **Worker1**، **Worker2** و **Worker3** در روش سلسله مراتب دیده می‌شود. دقت کنید که **Worker1** نقش یک تابع کارفرما را برای توابعی **Worker4** و **Worker5** بازی می‌کند.



شکل ۱-٦ | سلسله مراتب رابطه تابع کارفرما/تابع کارگر.

۳-٦ توابع کتابخانه math

همان طوری که می‌دانید، یک کلاس می‌تواند دارای توابع عضوی باشد که سرویس‌های کلاس را انجام می‌دهند. برای مثال، در فصل‌های ۳ الی ۵، توابع عضو نسخه‌های مختلفی از یک شی **Gradebook** را

برای نمایش پیغام خوش‌آمدگویی، تنظیم نام دوره، دستیابی به مجموعه نمرات و محاسبه میانگین این نمرات فراخوانی کرده‌اید.

گاهی اوقات توابع اعضای یک کلاس نیستند. چنین توابعی، توابع سراسری نامیده می‌شوند. نمونه های اولیه تابع در توابع سراسری نیز مانند توابع عضو یک کلاس، در فایلهای سرآیند قرار دارند، از اینروست که توابع سراسری می‌توانند در هر برنامه‌ای که فایل سرآیند مربوطه را ضمیمه می‌کند مورد استفاده مجدد قرار گیرند. برای مثال، به خاطر دارید که از تابع **pow** فایل سرآیند <**cmath**> برای به توان رساندن یک عدد در برنامه شکل ۶-۵ استفاده کردیم. به معرفی توابع مختلف از فایل سرآیند <**cmath**> می‌پردازیم تا مفهوم توابع سراسری را که به یک کلاس خاص تعلق ندارند بیان کنیم. در این فصل و فصلهای آتی، از ترکیبی از توابع سراسری (همانند **main**) و کلاسهای دارای توابع عضو برای پیاده‌سازی مثال‌های خود استفاده خواهیم کرد.

فایل سرآیند <**cmath**> کلکسیونی از توابع تدارک دیده است که به برنامه‌نویس امکان می‌دهند تا بسیاری از محاسبات رایج در ریاضیات را انجام دهد. برای مثال، ممکن است برنامه‌نویس علاقمند به محاسبه و نمایش ریشه دوم 900.0 باشد، از اینرو می‌تواند از عبارت زیر استفاده کند

```
sqrt( 900.0 )
```

زمانیکه این عبارت اجرا شود، تابع **sqrt** فراخوانی می‌شود تا ریشه دوم عدد قرار گرفته در درون پرانتزها (900.0) را محاسبه کند. این تابع آرگومانی از نوع **double** دریافت و نتیجه‌ای از نوع **double** برگشت می‌دهد. دقت کنید که قبل از فراخوانی تابع **sqrt** نیازی به ایجاد هیچ شی نیست. عدد 900.0 آرگومان تابع **sqrt** محسوب می‌شود. عبارت فوق مقدار 30.0 را بدست خواهد داد. همچنین توجه نمائید که تمام توابع موجود در فایل سرآیند <**cmath**> از نوع توابع سراسری هستند و از اینرو، برای فراخوانی کافیست نام تابع و بدنبال آن پرانتزهای حاوی آرگومان مورد نیاز تابع قرار داده شود.

آرگومان‌های تابع می‌توانند مقادیر ثابت، متغیر و حتی عبارات بسیار پیچیده باشند. اگر c=13.0، d=3.0 و f=4.0 باشد، پس عبارت

```
cout << sqrt( c + d * f ) << endl;
```

مبادرت به محاسبه ریشه دوم 25.0 = 4.0 + 3.0 + 13.0 خواهد کرد و پاسخ 5.0 باز می‌گرداند. در جدول شکل ۲-۶ تعدادی از توابع کتابخانه **math** آورده شده است. در این جدول متغیرهای **x** و **y** از نوع **double** هستند.

تابع	توضیح	مثال
fabs(x)	قدر مطلق x	fabs(5.1) is 5.1 fabs(0.0) is 0.0 fabs(-8.76) is 8.76
ceil(x)	گرد کردن x به کوچکترین مقدار صحیح، بطوریکه کوچکتر از x نباشد.	ceil(9.2) is 10.0 ceil(-9.8) is -9.0
cos(x)	محاسبه کسینوس x (x بر حسب رادیان)	cos(0.0) is 1.0
exp(x)	محاسبه e بتوان x	exp(1.0) is 2.71828 exp(2.0) is 7.38906
floor(x)	گرد کردن x به بزرگترین عدد صحیح، بطوریکه بزرگتر از x نباشد.	floor(9.2) is 9.0 floor(-9.8) is -10.0
log(x)	لگاریتم طبیعی x (برپایه e)	log(2.718282) is 1.0 log(7.389056) is 2.0
max(x, y)	مقدار بزرگ x و y	max(2.3, 12.7) is 12.7 max(-2.3, -12.7) is -2.3
min(x, y)	مقدار کوچک x و y	min(2.3, 12.7) is 2.3 min(-2.3, -12.7) is -12.7
pow(x, y)	محاسبه x بتوان y	pow(2.0, 7.0) is 128.0 pow(9.0, .5) is 3.0
round(x)	گرد کردن x به نزدیکترین مقدار صحیح	round(9.75) is 10 round(9.25) is 9
sin(x)	محاسبه سینوس x (x بر حسب رادیان)	sin(0.0) is 0.0
sqrt(x)	محاسبه ریشه دوم x	sqrt(900.0) is 30.0 sqrt(9.0) is 3.0
tan(x)	محاسبه تانژانت x (x برحسب رادیان)	tan(0.0) is 0.0

شکل ۲-٦ |توابع کتابخانه math.

٦-٤ تعریف تابع با پارامترهای مضاعف

در فصل‌های ۳ الی ۵ کلاس‌هایی ارائه شده که حاوی توابع ساده‌ای بودند که حداکثر یک پارامتر داشتند. توابع برای انجام کارهای خود غالبا نیازمند بیش از یک داده هستند. در این بخش به بررسی توابع با پارامترهای مضاعف می‌پردازیم.

برنامه موجود در شکل‌های ۳-۶ الی ۵-۶ کلاس **GradeBook** را با اضافه کردن یک تابع که توسط کاربر تعریف شده، بنام تابع **maximum** اصلاح می‌کند. این تابع، بزرگترین مقدار از میان سه مقدار int را تعیین کرده و برگشت می‌دهد. زمانیکه که اجرای برنامه آغاز می گردد، تابع **main** (خطوط 14-5 در شکل ۵-۶) یک شی از کلاس **GradeBook** را ایجاد می‌کند (خط 8) و تابع عضو **inputGrade** شی را برای گرفتن سه نمره صحیح از کاربر فراخوانی می‌کند (خط 11). در فایل پیاده‌سازی کلاس **GradeBook** (شکل ۴-۶)، خطوط 55-54 تابع عضو **inputGrades** به کاربر اعلان می‌کنند تا سه مقدار **integer** وارد کند. خط 58 تابع عضو **maximum** (در خطوط 75-62) را فراخوانی می‌کند. تابع **maximum** بزرگترین مقدار را مشخص می سازد، سپس فرمان **return** (خط 74) این مقدار را به مکانی که تابع **inputGrades** از آن مکان تابع **maximum** را فراخوانی کرده است، برگشت می‌دهد. سپس تابع عضو **inputGrades** مقدار برگشتی **maximum** را در عضو داده **maximumGrade** ذخیره می‌کند. سپس این مقدار با فراخوانی تابع **displayGradeReport** (خط 12 در شکل ۵-۶) به خروجی ارسال می شود. [نکته: این تابع را **displayGradeReport** نام داده‌ایم، چراکه نسخه های بعدی کلاس **GradeBook** از این تابع برای نمایش یک گزارش کامل از نمرات، شامل نمرات حداکثر و حداقل، استفاده خواهند کرد.] در فصل هفتم، **GradeBook** را به نحوی توسعه می‌دهیم که تعداد دلخواهی از نمرات را پردازش کند.

```
1  // Fig. 6.3: GradeBook.h
2  // Definition of class GradeBook that finds the maximum of three grades.
3  // Member functions are defined in GradeBook.cpp
4  #include <string> // program uses C++ standard string class
5  using std::string;
6
7  // GradeBook class definition
8  class GradeBook
9  {
10 public:
11    GradeBook( string ); // constructor initializes course name
12    void setCourseName( string ); // function to set the course name
13    string getCourseName(); // function to retrieve the course name
14    void displayMessage(); // display a welcome message
15    void inputGrades(); // input three grades from user
16    void displayGradeReport(); // display a report based on the grades
17    int maximum( int, int, int ); // determine max of 3 values
18 private:
19    string courseName; // course name for this GradeBook
20    int maximumGrade; // maximum of three grades
21 }; // end class GradeBook
```

شکل ۳-٦ |فایل سرآیند GradeBook.

```
1   // Fig. 6.4: GradeBook.cpp
2   // Member-function definitions for class GradeBook that
3   // determines the maximum of three grades.
```

```
4   #include <iostream>
5   using std::cout;
6   using std::cin;
7   using std::endl;
8
9   #include "GradeBook.h" // include definition of class GradeBook
10
11  // constructor initializes courseName with string supplied as argument;
12  // initializes maximumGrade to 0
13  GradeBook::GradeBook( string name )
14  {
15     setCourseName( name ); // validate and store courseName
16     maximumGrade = 0; // this value will be replaced by the maximum grade
17  } // end GradeBook constructor
18
19  // function to set the course name; limits name to 25 or fewer characters
20  void GradeBook::setCourseName( string name )
21  {
22     if ( name.length() <= 25 ) // if name has 25 or fewer characters
23        courseName = name; // store the course name in the object
24     else // if name is longer than 25 characters
25     { // set courseName to first 25 characters of parameter name
26        courseName = name.substr( 0, 25 ); // select first 25 characters
27        cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
28           << "Limiting courseName to first 25 characters.\n" << endl;
29     } // end if...else
30  } // end function setCourseName
31
32  // function to retrieve the course name
33  string GradeBook::getCourseName()
34  {
35     return courseName;
36  } // end function getCourseName
37
38  // display a welcome message to the GradeBook user
39  void GradeBook::displayMessage()
40  {
41     // this statement calls getCourseName to get the
42     // name of the course this GradeBook represents
43    cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
44        << endl;
45  } // end function displayMessage
46
47  // input three grades from user; determine maximum
48  void GradeBook::inputGrades()
49  {
50     int grade1; // first grade entered by user
51     int grade2; // second grade entered by user
52     int grade3; // third grade entered by user
53
54     cout << "Enter three integer grades: ";
55     cin >> grade1 >> grade2 >> grade3;
56
57     // store maximum in member studentMaximum
58     maximumGrade = maximum( grade1, grade2, grade3 );
59  } // end function inputGrades
60
61  // returns the maximum of its three integer parameters
62  int GradeBook::maximum( int x, int y, int z )
63  {
64     int maximumValue = x; // assume x is the largest to start
65
66     // determine whether y is greater than maximumValue
67     if ( y > maximumValue )
68        maximumValue = y; // make y the new maximumValue
69
70     // determine whether z is greater than maximumValue
71     if ( z > maximumValue )
72        maximumValue = z; // make z the new maximumValue
73
74     return maximumValue;
75  } // end function maximum
76
77  // display a report based on the grades entered by user
78  void GradeBook::displayGradeReport()
79  {
80     // output maximum of grades entered
81     cout << "Maximum of grades entered: " << maximumGrade << endl;
82    } // end function displayGradeReport
```

شکل ٤-٦ |کلاس GradeBook تعریف کننده تابع maximum.

```
1  // Fig. 6.5: fig06_05.cpp
2  // Create GradeBook object, input grades and display grade report.
3  #include "GradeBook.h" // include definition of class GradeBook
4
5  int main()
6  {
7     // create GradeBook object
8     GradeBook myGradeBook( "CS101 C++ Programming" );
9
10    myGradeBook.displayMessage(); // display welcome message
11    myGradeBook.inputGrades(); // read grades from user
12    myGradeBook.displayGradeReport(); // display report based on grades
13    return 0; // indicate successful termination
14 } // end main
```

```
Welcome to the grade book for
CS101 C++ Programming!


Welcome to the grade book for
CS101 C++ Programming!
Welcome to the grade book for
CS101 C++ Programming!

Enter three integer grades: 67 75 86
Maximum of grades entered: 86
```

شکل ٥-٦ |عملکرد تابع maximum.

مهندسی نرم‌افزار

*کاماهایی که در خط 58از شکل ۴-۶ برای جدا کردن آرگومان‌های تابع **maximum** مورد استفاده قرار گرفته‌اند با عملگرهای کاما که در بخش۳-۵ توضیح داده شده‌اند یکسان نیستند. عملگر کاما تضمین می‌کند که عملوندهای آن از چپ به راست ارزیابی می‌شوند. اما ترتیب ارزیابی آرگومان‌های یک تابع، از سوی **C++** مشخص نمی‌شود. از اینرو، کامپایلرهای مختلف می‌توانند آرگومان‌های تابع را به ترتیب‌های متفاوت ارزیابی کنند.*

قابلیت حمل

گاهی اوقات زمانی که آرگومان‌های یک تابع عبارات بیشتری را شامل می‌شوند، همانند توابعی که توابع دیگری را فراخوانی می‌کنند، ترتیب ارزیابی آرگومان‌ها از سوی کامپایلر می‌تواند مقادیر یک یا چند آرگومان را تحت تاثیر قرار دهد. اگر ترتیب ارزیابی در میان کامپایلرها متفاوت باشد، مقادیر آرگومان ارسال شده به تابع می‌تواند تغییر پیدا کند، که این حالت خطاهای منطقی بوجود می‌آورد.

اجتناب از خطا

اگر در مورد ترتیب ارزیابی آرگومان‌های یک تابع و ترتیب ارزیابی مقادیر ارسالی به تابع شک دارید، آرگومان‌ها را قبل از فراخوانی تابع درعبارات تخصیصی مجزا مورد ارزیابی قرار داده، و نتیجه هر عبارت را به یک متغیر محلی تخصیص دهید، سپس این متغیرها را به عنوان آرگومان به تابع ارسال کنید.

نمونه اولیه تابع عضو **maximum** (شکل ۳-۶، خط 17) تعیین می‌کند که تابع یک مقدار صحیح برگشت می‌دهد، نام تابع **maximum** است و تابع برای انجام کار خود به سه پارامتر صحیح نیاز دارد. سرآیند تابع

maximum (شکل ۴-۶، خط 62) با نمونه اولیه تابع مطابقت دارد و نشان می‌دهد که پارامترها **x**، **y** و **z** نام دارند. زمانیکه **maximum** فراخوانی شود (شکل ۴-۶، خط 58)، پارامتر **x** با مقدار آرگومان **grade1** ، پارامتر **y** با مقدار آرگومان **grade2** و پارامتر **z** با مقدار آرگومان **garde3** مقداردهی اولیه می‌شوند. فراخوانی تابع بر اساس تعریف تابع باید برای هر پارامتر یک آرگومان داشته باشد.

دقت کنید که هم در نمونه اولیه تابع و هم در سرآیند تابع چندین پارامتر به صورت یک لیست که توسط کاما از هم جدا شده‌اند، مشخص شده است. کامپایلر برای بررسی این مطلب که آیا فراخوانی‌های تابع **maximum** تعداد و نوع درستی از آرگومان‌ها را دارد و نوع آرگومان‌ها به ترتیب صحیحی در کنار هم قرار گرفته‌اند به نمونه اولیه تابع مراجعه می‌کند. علاوه بر این، کامپایلر نمونه اولیه را برای اطمینان از درستی مقدار برگشتی تابع به عبارتی که تابع را فراخوانی کرده است مورد استفاده قرار می‌دهد (برای مثال فراخوانی تابعی که **void** را برگشت می‌دهد، نمی‌تواند در سمت راست یک دستور تخصیص بکار گرفته شود). هر آرگومان باید با نوع پارامتر متناظرش سازگار باشد. برای مثال، یک پارامتر از نوع **double** می‌تواند مقادیری از قبیل 7.37، 22 یا 0.03456- را دریافت کند، اما قادر به دریافت رشته‌ای همانند "hello" نیست. اگر آرگومان‌های ارسالی به یک تابع با نوع‌های تعیین شده در نمونه اولیه تابع مطابقت نداشته باشند، کامپایلر آرگومان‌ها را به نوع متناظر تبدیل می‌کند. در بخش ۵-۶ این تبدیل توضیح داده شده است.

خطای برنامه‌نویسی

*اعلان پارامترهای متد از یک نوع بصورت **double x، y** به جای **double x، double y** خطای نحوی است. چرا که برای هر پارامتر در لیست پارامتری باید یک نوع صریح تعریف شود.*

خطای برنامه‌نویسی

اگر نمونه اولیه تابع، سرآیند تابع و فراخوانی‌های تابع همگی از نظر تعداد، نوع و ترتیب آرگومان‌ها و پارامترها، و از نظر نوع برگشتی مطابقت نداشته باشند، خطای کامپایل رخ خواهد داد.

مهندسی نرم‌افزار

تابعی که دارای تعداد زیادی پارامتر است احتمالا وظایف زیادی دارد. تقسیم تابع به توابع کوچکتر که هر یک وظیفه خاصی را انجام می‌دهند، می‌تواند سرآیند تابع را به یک خط محدود کند.

برای تعیین مقدار ماکزیمم (خطوط 75-62 از شکل ۴-۶)، با این فرض کار را آغاز می‌شود که پارامتر **x** حاوی بزرگترین مقدار است، از اینرو خط 64 در تابع **maximum** متغیر محلی **maximumValue** را اعلان کرده و آن را با مقدار پارامتر **x** مقداردهی اولیه می‌کند. البته این امکان وجود دارد که پارامتر **y** یا **z** حاوی بزرگترین مقدار باشند، بنابر این باید هر یک از این مقادیر را با **maximumValue** مقایسه کنیم. عبارت **if** در خطوط 68-67 تعیین می‌کند که آیا **y** بزرگتر از **maximumValue** است یا خیر، اگر چنین

باشد، **y** را به **maximumValue** تخصیص می‌دهد. عبارت **if** در خطوط 72-71 تعیین می‌کند که آیا **z** بزرگتر از **maximumValue** است یا خیر، اگر چنین باشد، **z** را به **maximumValue** تخصیص می‌دهد. در این مرحله بزرگترین مقدار در **maximumValue** قرار دارد، بنابر این خط 74 این مقدار را به فراخوان در خط 58 برگشت می‌دهد. زمانیکه کنترل برنامه به نقطه‌ای از برنامه که **maximum** فراخوانی شده باز می‌گردد، پارامترهای **maximum** یعنی **x**، **y**،**z** دیگر برای برنامه دسترس پذیر نیستند. در بخش بعد به بررسی این مسئله خواهیم پرداخت.

سه روش برای بازگرداندن کنترل به نقطه‌ای که تابع فراخوانی شده است وجود دارد. اگر تابع نتیجه‌ای برگشت ندهد (نوع برگشتی تابع **void** باشد)، کنترل زمانیکه برنامه به انتهای تابع (براکت سمت راست) یا عبارت **return;** برسد، برگشت داده خواهد شد. اگر تابع مقداری برگشت دهد، عبارت

```
return عبارت;
```

مقدار *عبارت* را به فراخوان برگشت می‌دهد. هنگامی که عبارت **return** اجرا می‌شود، بلافاصله کنترل به نقطه‌ای که تابع از آن مکان فعال شده، برگشت داده می‌شود.

٦-٥ نمونه اولیه تابع و تبدیل آرگومان

نمونه اولیه یک تابع (که اعلان یک تابع هم نامیده می‌شود) به کامپایلر نام تابع، نوع داده برگشتی تابع، تعداد پارامترهایی که تابع انتظار دریافت آنها را دارد، نوع و ترتیب این پارامترها را اعلان می‌کند.

مهندسی نرم‌افزار

*نمونه‌های اولیه تابع در C++الزامی هستند. از دستور دهنده‌های پیش پردازنده **#include** برای دستیابی به نمونه‌های اولیه تابع موجود در فایل‌های سرآیند در کتابخانه‌های مناسب (همانند، نمونه اولیه برای تابع ریاضی **sqrt** در فایل سرآیند <**cmath**>) برای توابع کتابخانه استاندارد C++ استفاده کنید. همچنین از **#include** برای دستیابی به فایلهای سرآیند حاوی نمونه‌های اولیه که توسط خودتان یا اعضای گروه نوشته شده، استفاده کنید.*

خطای برنامه‌نویسی

اگر تابعی قبل از آنکه فراخوانی شود تعریف شده باشد، پس تعریف تابع به عنوان نمونه اولیه تابع عمل خواهد کرد، از اینرو نیازی به یک نمونه اولیه مجزا نیست. اگر یک تابع قبل از آنکه تعریف شود فراخوانی گردد و دارای یک نمونه اولیه تابع نباشد، خطای کامپایل رخ خواهد داد.

مهندسی نرم‌افزار

همیشه نمونه‌های اولیه تابع را تدارک ببینید، حتی اگر توابع پیش از آنکه مورد استفاده قرار گیرند تعریف شده باشند (در حالتی که سرآیند تابع به عنوان نمونه اولیه تابع هم عمل کند) تدارک دین نمونه های اولیه از گره خوردن کد به تعریف ترتیب توابع جلوگیری می‌کند.

امضاء تابع

قسمتی از نمونه اولیه یک تابع که شامل نام تابع و نوع آرگومان‌های آن است بعنوان "امضای تابع" یا "امضا" شناخته می شود. امضای تابع نوع برگشتی تابع را تعیین نمی‌کند. توابع موجود در یک قلمرو باید دارای امضاهای منحصر به فرد باشند. قلمرو یک تابع منطقه‌ای از برنامه است که تابع در آن شناخته شده و در دسترس می‌باشد. در بخش ۱۰-۶ به بررسی دقیق‌تر قلمرو پرداخته‌ایم.

خطای برنامه‌نویسی

اگر دو تابع در یک قلمرو دارای امضاهای یکسان باشند اما نوع برگشتی آنها متفاوت باشد، با خطای کامپایل مواجه خواهید شد.

در شکل ۳-۶ اگر نمونه اولیه تابع در خط 17 به اینصورت نوشته شده بود

```
void maximum( int, int, int );
```

کامپایلر خطا گزارش می‌کرد، چرا که نوع برگشتی **void** در نمونه اولیه تابع با نوع برگشتی **int** در سرآیند تابع متفاوت است. به همین ترتیب، چنین نمونه اولیه سبب خواهد شد، عبارت

```
cout << maximum( 6, 9, 0 );
```

خطای کامپایل تولید کند، چرا که عبارت فوق وابسته به **maximum** برای برگشت دادن مقدار برای نمایش است.

الزام یا تبدیل آرگومان

یکی از ویژگی‌های مهم در نمونه اولیه توابع، *تبدیل آرگومان* است (مجبور کردن آرگومان‌ها برای بدست آوردن نوع داده مقتضی مشخص شده در اعلان پارامترها). برای مثال، برنامه‌ای می‌تواند یک تابع با یک آرگومان از نوع صحیح را فراخوانی کند، حتی اگر نمونه اولیه تابع آرگومان را از نوع **double** مشخص کرده باشد. در اینحالت هم برنامه بدرستی کار خواهد کرد.

قوانین ترقی آرگومان

گاهی اوقات، مقادیر آرگومان که دقیقا با نوع‌های پارامتر در نمونه اولیه تابع مطابقت ندارند، می‌توانند قبل از آنکه تابع فراخوانی شود توسط کامپایلر به نوع مناسب تبدیل گردند. این تبدیل‌ها تحت قوانینی بنام قوانین ترقی آرگومان در C++، رخ می‌دهند. قوانین ترقی تعیین می‌کنند که چگونه می‌توان بدون از دست دادن داده‌ها به تبدیل نوع اقدام کرد. یک **int** بدون تغییر یافتن مقدار خود، می‌تواند به یک نوع **double** تبدیل گردد. ولی در تبدیل یک **double** به یک **int**، قسمت اعشاری مقدار **double** از دست خواهد رفت. به خاطر دارید که متغیرهای **double** می‌توانند اعدادی با ارقام بسیار بیشتر از اعداد **int** در خود ذخیره کنند، از اینرو امکان از رفتن داده وجود دارد. همچنین امکان دارد که مقادیر در ضمن تبدیل

نوع‌های عددی بزرگتر به نوع‌های عددی کوچکتر (همانند **long** به **short**)، علامت‌دار به بدون علامت یا بدون علامت به علامت‌دار تغییر یابند.

قوانین ترقی بر روی عباراتی که حاوی مقادیری از یک نوع داده یا چندین نوع داده باشند، اعمال می‌شود، چنین عباراتی به عنوان عبارات ترکیبی شناخته می‌شوند. نوع هر مقدار موجود در یک عبارت نوع ترکیبی به "بالاترین" نوع موجود در عبارت ارتقا می‌یابد (در واقع یک نسخه موقت از هر مقدار ایجاد شده و برای عبارت بکار گرفته می‌شود، مقادیر اصلی بدون تغییر باقی می‌مانند). همچنین ترقی زمانی رخ می‌دهد که نوع آرگومان یک تابع با نوع پارامتر تعیین شده در تعریف تابع یا نمونه اولیه تابع مطابقت نداشته باشد. در جدول شکل ۶-۶ نوع‌های داده بنیادین به ترتیب از "بالاترین نوع" به "پایین‌ترین نوع" لیست شده‌اند.

تبدیل مقادیر به نوع های بنیادین پایین‌تر می‌تواند منجر به تولید مقادیر اشتباه گردد. از اینرو، یک مقدار فقط در صورتی می‌تواند به یک نوع بنیادین پایین‌تر تبدیل شود که به فرم صریح به یک متغیر از نوع پایین‌تر تخصیص داده شود (تعدادی کامپایلرها در اینحالت هشدار صادر می‌کنند) یا از عملگر تبدیل **cast** به این منظور استفاده شود (به بخش ۹-۴ مراجعه کنید). نحوه تبدیل مقادیر آرگومان تابع به نوع های پارامتر در نمونه اولیه یک تابع دقیقا همانند تخصیص دادن مقادیر به متغیرهایی از آن نوع است. اگر تابع **square** که از یک پارامتر عددی از نوع صحیح استفاده می‌کند با آرگومان اعشاری فراخوانی شود، آرگومان به **int** تبدیل می‌شود(نوع پایین تر) و احتمالا **square** مقدار نادرستی برگشت خواهد داد. برای مثال، **square(4.5)** مقدار 16 را برگشت می‌دهد و نه مقدار20.25.

نوع داده
long double
double
float
unsigned long int
long int
unsigned int
int
unsigned short int
short int

```
unsigned char
```

```
char
```

```
bool
```

شکل ٦-٦ | سلسله مراتب ترقی نوع‌های بنیادین.

خطای برنامه‌نویسی

به هنگام تبدیل از نوع داده بالاتر در سلسله مراتب ترقی به یک نوع پایین‌تر، یا مابین نوع‌های علامت‌دار و بدون علامت، امکان از دست رفتن داده وجود دارد.

خطای برنامه‌نویسی

اگر آرگومان‌های موجود در فراخوانی یک تابع با تعداد و نوع پارامترهای اعلان شده در نمونه اولیه تابع متناظر مطابقت نداشته باشند، خطای کامپایل رخ خواهد داد. همچنین اگر تعداد آرگومانهای در فراخوانی‌ها مطابقت داشته باشند، اما نتوان آرگومان‌ها را بصورت ضمنی به نوع‌های مورد نظر تبدیل کرد، با خطا مواجه خواهید شد.

٦-٦ فایل‌های سرآیند کتابخانه استاندارد C++

کتابخانه استاندارد C++ به قسمت‌های بسیاری تقسیم شده، که هر قسمت دارای فایل سرآیند متعلق به خود است. فایل‌های سرآیند حاوی نمونه‌های اولیه تابع برای توابع مرتبطی است که هر بخش از کتابخانه را تشکیل می‌دهند. همچنین فایل‌های سرآیند حاوی تعاریفی از انواع کلاس و توابع به همراه ثابت‌های مورد نیاز این توابع می‌باشند. یک فایل سرآیند کامپایلر را در مورد نحوه برقراری ارتباط با کامپونت‌های کتابخانه و نوشته شده توسط کاربر هدایت می‌کند.

در جدول شکل ۷-۶ تعدادی از فایل‌های سرآیند رایج از کتابخانه استاندارد C++ لیست شده است که در مورد اکثر آنها در این کتاب صحبت خواهد شد. اسامی فایل‌های سرآیند که با **h.** پایان می‌یابند فایل‌های سرآیند از نوع سبک قدیمی هستند که توسط فایل‌های سرآیند کتابخانه استاندارد C++ جایگزین گشته‌اند. در این کتاب فقط از نسخه‌های کتابخانه استاندارد C++ از هر فایل سرآیند استفاده کرده‌ایم تا مطمئن شویم که مثال‌های مطرح شده بر روی اکثر کامپایلرهای استاندارد C++ عمل خواهند کرد.

فایل‌های سرآیند کتابخانه استاندارد C++	توضیحات
<iostream>	حاوی نمونه‌های اولیه تابع برای توابع ورودی و خروجی استاندارد C++ است که در فصل دوم معرفی شده است، و در فصل پانزدهم جزئیات بیشتری از آن بررسی خواهد شد. این فایل سرآیند جایگزین فایل سرآیند <iostream.h> شده است.
<iomanip>	حاوی نمونه‌های اولیه تابع برای دستکاری کننده‌های جریان است که جریان‌های داده را

	قالب‌بندی می‌کنند. از این فایل سرآیند ابتدا در بخش ۹-۴ استفاده شده است و در فصل پانزدهم جزئیات بیشتری از آن بررسی خواهد شد. این فایل سرآیند جایگزین فایل سرآیند <**iomanip.h**> شده است
<cmath>	حاوی نمونه‌های اولیه تابع برای توابع کتابخانه‌ای ریاضی است. این فایل سرآیند جایگزین فایل سرآیند<**math.h**> شده است.
<cstdlib>	حاوی نمونه‌های اولیه تابع به منظور تبدیل اعداد به متن، متن به عدد، تخصیص حافظه، اعداد تصادفی و برخی از توابع یوتیلیتی یا کمکی دیگر است. قسمت‌های از این فایل سرآیند در بخش ۷-۶، فصل یازدهم، فصل شانزدهم، فصل نوزدهم مورد بررسی قرار گرفته‌اند. این فایل سرآیند جایگزین فایل سرآیند<**stdlib.h**> شده است.
<ctime>	حاوی نمونه‌های اولیه و نوع‌های تابع برای مدیریت زمان و تاریخ است. این فایل سرآیند جایگزین فایل سرآیند<time.h> شده است. این فایل سرآیند در بخش ۷-۶ بکار رفته است.
<vector>,<list>, <deque>,<queue>, <stack>,<map>, <set>,<bitset>	این فایل‌های سرآیند حاوی کلاس‌هایی هستند که حامل‌های کتابخانه استاندارد C++ را پیاده‌سازی می‌کنند. حامل‌ها مبادرت به ذخیره‌سازی داده‌ها در طی اجرای یک برنامه می‌کنند. سرآیند <**vector**> ابتدا در فصل هفتم معرفی خواهد شد.
<cctype>	حاوی نمونه‌های اولیه تابع برای توابعی است که کاراکترها را برای خصوصیت‌های خاصی تست می‌کنند (همانند اینکه آیا کاراکتر یک رقم است یا یک نقطه‌گذاری)، و نمونه‌های اولیه تابع برای توابعی است که می‌توانند در تبدیل حروف‌کوچک به حروف‌بزرگ و بالعکس بکار گرفته شوند. این فایل سرآیند جایگزین فایل سرآیند <**ctype.h**> شده است. این مباحث در فصل هشتم مطرح شده‌اند.
<cstring>	حاوی نمونه‌های اولیه تابع برای توابع پردازش رشته به سبک C است و این فایل سرآیند جایگزین فایل سرآیند <**string.h**> شده است. در فصل یازدهم از این سرآیند استفاده شده.
<typeinfo>	حاوی کلاس‌هایی برای شناسایی نوع در زمان اجرا (تعیین نوع داده در زمان اجرا) است. این فایل سرآیند در بخش ۸-۱۳ بکار گرفته شده است.
<exception>, <stdexcept>	این فایل‌های سرآیند حاوی کلاس‌هایی هستند که در مدیریت استثناء بکار می‌روند.
<memory>	حاوی کلاس‌ها و توابعی است که از طرف کتابخانه استاندارد C++ به منظور تخصیص حافظه برای حامل‌های کتابخانه استاندارد C++ بکار گرفته می‌شوند. از این سرآیند در فصل شانزدهم استفاده شده است.
<fstream>	حاوی نمونه‌های اولیه تابع برای توابعی است که عملیلت ورودی از فایل‌های موجود بر روی دیسک و خروجی به فایل‌های موجود بر روی دیسک (در فصل هفدهم بکار گرفته شده‌اند) را انجام می‌دهند. این فایل سرآیند جایگزین فایل سرآیند<**fstream.h**> شده است.
<string>	حاوی تعریف کلاس **string** از کتابخانه استاندارد C++ است(در فصل هیجدهم بکار گرفته شده است).
<sstream>	حاوی نمونه‌های اولیه تابع برای توابعی است که عملیات ورودی از رشته‌های موجود در

	حافظه را پیاده‌سازی می‌کنند (در فصل هیجدهم بکار گرفته شده است).
<functional>	حاوی کلاس‌ها و توابعی است که از طرف الگوریتم‌های کتابخانه استاندارد C++ بکار برده می‌شوند.
<iterator>	حاوی کلاس‌هایی برای دسترسی به داده‌ها در حامل‌های کتابخانه استاندارد C++ است.
<algorithm>	حاوی توابعی برای مدیریت داده‌ها در حامل‌های کتابخانه استاندارد C++ است.
<cassert>	حاوی ماکروهایی برای تشخیص خطاها در برنامه است. این فایل سرآیند جایگزین فایل سرآیند <**assert.h**> در C++ قبل از استاندارد شده است.
<cfloat>	حاوی محدودیت سایز اعشاری سیستم است. این فایل سرآیند جایگزین سرآیند<**float.h**> شده است.
<climits>	حاوی محدودیت سایز اعداد صحیح سیستم است. این فایل سرآیند جایگزین فایل سرآیند <limits.h> شده است.
<cstdio>	حائی نمونه‌های اولیه تابع برای توابع کتابخانه ورودی/خروجی استاندارد سبک C و اطلاعات بکار رفته توسط آنها است. این فایل سرآیند جایگزین فایل سرآیند <**stdio.h**> شده است.
<locate>	حاوی کلاس‌ها و توابعی است که معمولا از طرف پردازش جریان برای پردازش داده‌ها به شکل طبیعی برای زبان‌های مختلف (همانند، فرمت‌های پولی، مرتب‌سازی رشته‌ها، عرضه کاراکترها و ...) بکار گرفته می‌شود.
<limits>	حاوی کلاس‌هایی برای تعریف محدودیت‌های نوع داده بر روی پلات‌فرم هر کامپیوتری است.
<utility>	حاوی کلاس‌ها و توابعی است که توسط بسیاری از فایل‌های سرآیند کتابخانه استاندارد C++ بکار گرفته می‌شوند.

شکل ۷-٦ | فایل‌های سرآیند کتابخانه استاندارد C++.

۷-٦ مبحث آموزشی: تولید اعداد تصادفی

در این بخش به بحث برنامه‌نویسی برنامه‌های بازی و شبیه سازی می‌پردازیم. در این بخش و بخش بعدی، با استفاده از ساختارهای کنترلی که قبلاً با آنها آشنا شده‌اید یک برنامه بازی ایجاد خواهیم کرد که حاوی توابع متعددی است. این بازی در ارتباط با شانس است. عنصر شانس می‌تواند در برنامه‌های کامپیوتری از طریق تابع کتابخانه استاندارد ***rand*** ایجاد شود.

به عبارت زیر توجه نمائید:

```
i = rand();
```

تابع **rand** یک مقدار صحیح بدون علامت مابین صفر و ثابت **RAND_MAX** ایجاد می‌کند. یک ثابت نمادین ایجاد شده در فایل سرآیند <**cstdlib**>. بایستی مقدار **RAND_MAX** حداقل 32767 باشد، حداکثر مقدار مثبت برای یک عدد صحیح دو بایتی (16 بیت). در GNU C++، مقدار **RAND_MAX**

برابر با 214748647 و در ویژوال استودیو این مقدار برابر 32767 است. اگر **rand** مقادیری بصورت تصادفی ایجاد کند، هر مقدار در این محدودهٔ در هر بار فراخوانی تابع **rand** دارای *شانس* (احتمال) برابر خواهد بود.

گاهاً ایجاد اعداد تصادفی در یک برنامه ضرورت پیدا می‌کند. با این وجود، محدودهٔ مقادیر تولید شده توسط **rand** غالباً متفاوت از مقدار مورد نیاز در یک برنامه هستند. برای مثال، در برنامه‌ای که پرتاب را شبیه‌سازی می‌کند، فقط نیاز به مقدار 0 برای نشان دادن "رو" و 1 برای "پشت" سکه نیاز دارد، یا برنامه‌ای که پرتاب یک طاس شش وجهی را شبیه‌سازی می‌کند، نیاز به مقادیر تصادفی از 1 تا 6 دارد. به همین ترتیب، برنامه‌ای که حرکت یک سفینه فضایی را تداعی می‌کند و نیاز به حرکت در چهار جهت را دارد، مستلزم بدست آوردن عدد تصادفی از 1 تا 4 است.

پرتاب طاس شش وجهی

برای توصیف **rand**، اجازه دهید برنامه‌ای ایجاد کنیم (شکل ۸-۶) که 20 پرتاب یک تاس شش وجهی و چاپ مقدار هر پرتاب را شبیه‌سازی نماید. نمونه اولیه تابع **rand** در سرآیند <**cstdlib**> قرار دارد. برای تولید اعداد صحیح در بازه 0 تا 5، از عملگر باقیمانده (**%**) به همراه **rand** استفاده می‌کنیم:

```
rand() % 6
```

این عمل بعنوان تغییر مقیاس شناخته می‌شود. عدد 6 فاکتور تغییر مقیاس نامیده می‌شود. سپس بازه اعداد تولیدی را با افزودن عدد 1 به نتیجه قبلی، جابجا یا شیفت می‌دهیم. برنامه شکل ۸-۶ نشان می‌دهد که نتایج در بازه 1 تا 6 قرار دارند.

```
1  // Fig. 6.8: fig06_08.cpp
2  // Shifted and scaled random integers.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 #include <cstdlib> // contains function prototype for rand
11 using std::rand;
12
13 int main()
14 {
15    // loop 20 times
16    for ( int counter = 1; counter <= 20; counter++ )
17    {
18       // pick random number from 1 to 6 and output it
19       cout << setw( 10 ) << ( 1 + rand() % 6 );
20
21       // if counter is divisible by 5, start a new line of output
22       if ( counter % 5 == 0 )
23          cout << endl;
24    } // end for
25
26    return 0; // indicates successful termination
27 } // end main
```

```
6          6          5          5          6
5          1          1          5          3
6          6          2          4          2
6          2          3          4          1
```

شکل ۸-٦ | اعداد تصادفی در محدوده 6-1.

شش میلیون بار پرتاب یک طاس شش وجهی

برای اینکه نشان دهیم اعداد تولید شده توسط تابع **rand** تقریبا با احتمال برابر رخ می‌دهند، برنامه شکل ۹-۶ شش میلیون پرتاب یک تاس را شبیه‌سازی می‌کند. هر عدد صحیح در بازه 1 تا 6 باید تقریبا یک میلیون بار ظاهر گردد. این حالت در پنجره خروجی شکل ۹-۶ آورده شده است.

```
1  // Fig. 6.9: fig06_09.cpp
2  // Roll a six-sided die 6,000,000 times.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 #include <cstdlib> // contains function prototype for rand
11 using std::rand;
12
13 int main()
14 {
15    int frequency1 = 0; // count of 1s rolled
16    int frequency2 = 0; // count of 2s rolled
17    int frequency3 = 0; // count of 3s rolled
18    int frequency4 = 0; // count of 4s rolled
19    int frequency5 = 0; // count of 5s rolled
20    int frequency6 = 0; // count of 6s rolled
21
22    int face; // stores most recently rolled value
23
24    // summarize results of 6,000,000 rolls of a die
25    for ( int roll = 1; roll <= 6000000; roll++ )
26    {
27       face = 1 + rand() % 6; // random number from 1 to 6
28
29       // determine roll value 1-6 and increment appropriate counter
30       switch ( face )
31       {
32          case 1:
33             ++frequency1; // increment the 1s counter
34             break;
35          case 2:
36             ++frequency2; // increment the 2s counter
37             break;
38         case 3:
39             ++frequency3; // increment the 3s counter
40             break;
41          case 4:
42             ++frequency4; // increment the 4s counter
43             break;
44          case 5:
45             ++frequency5; // increment the 5s counter
46             break;
47          case 6:
48             ++frequency6; // increment the 6s counter
49             break;
50          default: // invalid value
51             cout << "Program should never get here!";
52       } // end switch
53    } // end for
54
55   cout << "Face" << setw( 13 ) << "Frequency" << endl; // output headers
56    cout << "   1" << setw( 13 ) << frequency1
57       << "\n   2" << setw( 13 ) << frequency2
58       << "\n   3" << setw( 13 ) << frequency3
59       << "\n   4" << setw( 13 ) << frequency4
60       << "\n   5" << setw( 13 ) << frequency5
61       << "\n   6" << setw( 13 ) << frequency6 << endl;
62    return 0; // indicates successful termination
63 } // end main
```

```
Face        Frequency
   1           999702
   2          1000823
   3           999378
   4           998898
   5          1000777
   6          1000422
```

شکل ۹-٦ | پرتاپ 6,000,000 بار طاس شش وجهی.

همانطوری که خروجی برنامه نشان می‌دهد، می‌توانیم پرتاب یک تاس شش وجهی را با تغییر دادن مقیاس و شیفت مقادیر تولید شده از طرف **rand** شبیه‌سازی نمائیم. دقت کنید که برنامه هرگز نباید به حالت **default** (خطوط 51-50) در ساختار **switch** وارد گردد، برای اینکه عبارت کنترلی **switch** یعنی **face** همیشه در بازه 6-1 قرار دارد، با این وجود حالت **default** را بعنوان یک تمرین و عمل خوب در نظر گرفته‌ایم. پس از آنکه با آرایه‌ها در فصل هفتم آشنا گردیدید، با نحوه جایگزین ساختن کل ساختار **switch** بکار رفته در شکل ۹-۶ با یک عبارت آشنا خواهید شد.

تصادفی کردن تولید اعداد تصادفی

با اجرای برنامه شکل ۸-۶ مجددا این مقادیر تولید می‌شوند

```
6       6       5       5       6
5       1       1       5       3
6       6       2       4       2
6       2       3       4       1
```

توجه کنید که برنامه دقیقا همان دنباله از مقادیر نشان داده شده در شکل ۸-۶ را چاپ می‌کند. این مقادیر چگونه می‌توانند تصادفی باشند؟ بطور کلی، این قابلیت تکرار یکی از صفات مهم تابع **rand** است. به هنگام دیباگ یک برنامه شبیه‌سازی، این خصیصه تکرار برای اثبات اینکه اصلاحات صورت گرفته در برنامه به درستی عمل می‌کنند، لازم است.

در واقع تابع **rand** اعداد شبه‌تصادفی تولید می‌کند. فراخوانی مکرر **rand** دنباله‌ای از اعداد تولید می‌کند که به نظر تصادفی می‌رسند. با این وجود، خود این توالی هر دفعه که برنامه اجرا می‌شود تکرار می‌گردد. زمانیکه برنامه کاملا خطایابی شد، می‌توان در هر بار اجرا دنباله متفاوتی از اعداد تصادفی تولید کرد. این فرآیند بعنوان تصادفی‌سازی مطرح است و با استفاده از تابع **srand** کتابخانه استاندارد C++ صورت می‌گردد. تابع **srand** یک آرگومان صحیح بدون علامت دریافت و تابع **rand** را برای تولید دنباله متفاوتی از اعداد تصادفی در هر بار اجرای برنامه *تغذیه* یا ***seed*** می‌نماید.

برنامه شکل ۱۰-۶ عملکرد تابع **srand** را نشان می‌دهد. برنامه از نوع داده **unsigned** استفاده می‌کند، که شکل کوتاه شده **unsigned int** است. یک **int** حداقل در دو بایت از حافظه ذخیره می‌شود (عموما چهار بایت از حافظه در سیستم‌های 32 بیتی) و می‌تواند حاوی مقادیر مثبت و منفی باشد. یک متغیر از نوع **unsigned int** حداقل در دو بایت از حافظه ذخیره می‌شود. یک **unsigned int** دو بایتی می‌تواند فقط دارای مقادیر مثبت در بازه 65535-0 باشد. یک **unsigned int** چهار بایتی می‌تواند فقط دارای مقادیر

مثبت در بازه 4294967295-0 باشد. تابع **srand** یک مقدار **unsigned int** را بعنوان آرگومان دریافت می‌کند. نمونه اولیه تابع برای **srand** در فایل سرآیند <**cstdlib**> قرار دارد.

```
1  // Fig. 6.10: fig06_10.cpp
2  // Randomizing die-rolling program.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  #include <iomanip>
9  using std::setw;
10
11 #include <cstdlib> // contains prototypes for functions srand and rand
12 using std::rand;
13 using std::srand;
14
15 int main()
16 {
17    unsigned seed; // stores the seed entered by the user
18
19    cout << "Enter seed: ";
20    cin >> seed;
21    srand( seed ); // seed random number generator
22
23    // loop 10 times
24    for ( int counter = 1; counter <= 10; counter++ )
25    {
26       // pick random number from 1 to 6 and output it
27       cout << setw( 10 ) << ( 1 + rand() % 6 );
28
29       // if counter is divisible by 5, start a new line of output
30       if ( counter % 5 == 0 )
31          cout << endl;
32    } // end for
33
34    return 0; // indicates successful termination
35 } // end main
```

```
Enter seed: 67
         6         1         4         6         2
         1         6         1         6         4
```

```
Enter seed: 432
         4         6         3         1         6
         3         1         5         4         2
```

```
Enter seed: 67
         6         1         4         6         2
         1         6         1         6         4
```

شکل ۱۰-٦ | تصادفی کردن پرتاب طاس شش وجهی.

اجازه دهید تا برنامه را چندین بار اجرا کرده و به بررسی نتایج بپردازیم. دقت کنید که برنامه در هر بار اجرا دنباله متفاوتی از اعداد تصادفی را تولید می‌نماید، چرا که کاربر در هر بار یک seed متفاوتی را وارد می‌کند. در خروجی‌های نمونه اول و سوم از seed یکسانی استفاده شده است، از اینرو در هر دو خروجی، دنباله یکسانی از اعداد نمایش داده شده است. برای تصادفی کردن بدون نیاز به وارد کردن یک seed در هر بار، می‌توانیم از عبارتی همانند

```
srand( time( 0 ) );
```

استفاده کنیم. این عبارت سبب می‌شود تا کامپیوتر مبادرت به خواندن ساعت (زمان) خود کرده و مقدار seed را بدست آورد. تابع **time** (با آرگومان **0** که در عبارت فوق نوشته شده است) زمان جاری را به صورت تعداد ثانیه‌های سپری شده از نیمه شب اول ژانویه 1970 به وقت GMT برگشت می‌دهد. این مقدار به یک عدد صحیح بدون علامت تبدیل شده و بعنوان seed در تولید کننده عدد تصادفی بکار گرفته می‌شود. نمونه اولیه تابع برای **time** در <**ctime**> قرار دارد.

خطای برنامه‌نویسی

فراخوانی تابع ***srand*** بیش از یک بار در یک برنامه، مجددا تولید دنباله اعداد شبه‌تصادفی را از ابتدا آغاز می‌کند و می‌تواند تصادفی بودن اعداد تولید شده توسط ***rand*** را تحت تاثیر قرار دهد.

تعمیم ضریب پیمایش و تغییر مکان اعداد تصادفی

قبل از این، در ارتباط با نحوه نوشتن یک دستور واحد برای شبیه سازی پرتاب یک تاس شش وجهی با عبارت زیر توضیحاتی بیان کردیم

```
face = 1 + rand() % 6;
```

که همیشه یک عدد صحیح را به صورت تصادفی به متغیر **face** در محدوده ی $1 \leq \text{face} \leq 6$ تخصیص می‌دهد. توجه کنید که پهنای این محدوده (یعنی تعداد اعداد صحیح متوالی موجود در محدوده) 6 بوده و عدد آغازین در دنباله 1 می‌باشد. با مراجعه به دستور فوق، مشاهده می‌کنید که پهنای محدوده به وسیله عدد بکار رفته در مقیاس **rand** با عملگر تعیین شده است، و عدد آغازین محدوده معادل با عددی است که با عبارت **rand % 6** جمع شده است (یعنی 1). می توانیم این نتیجه را بصورت زیر تعمیم دهیم

```
عدد = مقدار شیفت + rand() % فاکتور تغییر مقیاس;
```

که "مقدار شیفت" معادل با اولین عدد موجود در بازه دلخواه اعداد صحیح متوالی بوده و "فاکتور تغییر مقیاس" معادل با پهنای دلخواه محدوده اعداد صحیح متوالی می‌باشد. تجربه نشان داده که می‌توان اعداد صحیح را از مقادیر دیگری که بصورت اعداد صحیح متوالی نیستند هم ایجاد کرد.

٦-٨ مبحث آموزشی: بازی شانس و معرفی enum

یکی از بازی‌های مورد علاقه در بسیاری از نقاط جهان، بازی بنام "*craps*" است که با طاس انجام می‌شود. حال به قوانین این بازی توجه کنید:

بازیکن دو طاس می‌اندازد و هر طاس دارای شش وجه است. این وجه‌ها متشکل از 6 یا 5، 4، 3، 2، 1 نقطه هستند. پس از رها کردن طاس‌ها، مجموع نقاط موجود بر روی هر طاس محاسبه می‌شود. اگر مجموع برابر 7 یا 11 در اولین پرتاب طاس‌ها باشد، بازیکن پرتاب کننده، برنده خواهد شد. اگر مجموع 2، 3 یا 12 در اولین پرتاب طاس‌ها باشد، بازیکن پرتاب کننده بازنده خواهد بود. اگر مجموع نقاط 10 یا 9، 8، 6، 5، 4 در اولین پرتاب طاس باشد، این مجموع تبدیل به امتیاز بازیکن خواهد شد. برای برنده شدن، بازیکن باید به پرتاب طاس‌ها ادامه دهد تا به امتیاز تعیین شده دست یابد. اگر بازیکن مقدار 7 را بعد از امتیازگیری بدست آورد، بازنده می‌شود.

شبیه‌سازی این بازی در برنامه شکل ۱۱-۶ ارائه شده است. دقت کنید که بازیکن باید دو طاس را در اولین پرتاب و تمام پرتاب‌های بعدی بکار گیرد.

```
1  // Fig. 6.11: fig06_11.cpp
2  // Craps simulation.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstdlib> // contains prototypes for functions srand and rand
8  using std::rand;
9  using std::srand;
10
11 #include <ctime> // contains prototype for function time
12 using std::time;
13
14 int rollDice(); // rolls dice, calculates amd displays sum
15
16 int main()
17 {
18    // enumeration with constants that represent the game status
19    enum Status { CONTINUE, WON, LOST };
20
21    int myPoint; // point if no win or loss on first roll
22    Status gameStatus; // can contain CONTINUE, WON or LOST
23
24    // randomize random number generator using current time
25    srand( time( 0 ) );
26
27    int sumOfDice = rollDice(); // first roll of the dice
28
29    // determine game status and point (if needed) based on first roll
30    switch ( sumOfDice )
31    {
32       case 7: // win with 7 on first roll
33       case 11: // win with 11 on first roll
34          gameStatus = WON;
35          break;
36       case 2: // lose with 2 on first roll
37       case 3: // lose with 3 on first roll
38       case 12: // lose with 12 on first roll
39          gameStatus = LOST;
40          break;
41       default: // did not win or lose, so remember point
42          gameStatus = CONTINUE; // game is not over
43          myPoint = sumOfDice; // remember the point
44          cout << "Point is " << myPoint << endl;
45          break; // optional at end of switch
46    } // end switch
47
48    // while game is not complete
49    while ( gameStatus == CONTINUE ) // not WON or LOST
50    {
51       sumOfDice = rollDice(); // roll dice again
52
53       // determine game status
54       if ( sumOfDice == myPoint ) // win by making point
55          gameStatus = WON;
56       else
57          if ( sumOfDice == 7 ) // lose by rolling 7 before point
58             gameStatus = LOST;
59    } // end while
60
61    // display won or lost message
62    if ( gameStatus == WON )
63       cout << "Player wins" << endl;
64    else
65       cout << "Player loses" << endl;
66
67    return 0; // indicates successful termination
68 } // end main
69
70 // roll dice, calculate sum and display results
71 int rollDice()
72 {
73    // pick random die values
74    int die1 = 1 + rand() % 6; // first die roll
75    int die2 = 1 + rand() % 6; // second die roll
```

```
76
77    int sum = die1 + die2; // compute sum of die values.
78
79    // display results of this roll
80    cout << "Player rolled " << die1 << " + " << die2
81       << " = " << sum << endl;
82    return sum; // return sum of dice
83 } // end function rollDice
```

```
Player rolled 2 + 5 = 7
Player wins
```

```
Player rolled 6 + 6 = 12
Player loses
```

```
Player rolled 3 + 3 = 6
Point is 6
Player rolled 5 + 3 = 8
Player rolled 4 + 5 = 9
Player rolled 2 + 1 = 3
Player rolled 1 + 5 = 6
Player wins
```

```
Player rolled 1 + 3 = 4
Point is 4
Player rolled 4 + 6 = 10
Player rolled 2 + 4 = 6
Player rolled 6 + 4 = 10
Player rolled 2 + 3 = 5
Player rolled 2 + 4 = 6
Player rolled 1 + 1 = 2
Player rolled 4 + 4 = 8
Player rolled 4 + 3 = 7
Player loses
```

شکل ۱۱-٦ | شبیه‌سازی Craps.

برنامه‌نویسی ایده‌ال

در نام ثابت‌های شمارشی از حروف بزرگ استفاده کنید. این کار سبب مشخص شدن این ثابت‌ها در برنامه می‌شود و به برنامه‌نویس یاد آوری می‌کند که اینها ثابت‌های شمارشی هستند و نه متغیر.

به متغیرهایی که از سوی کاربر و از نوع **status** تعریف شده‌اند فقط یکی از سه مقدار اعلان شده در نوع شمارشی را می‌توان تخصیص داد. زمانیکه بازی با برد تمام می‌شود، برنامه متغیر **gameStatus** را با **WON** تنظیم می‌کند (خطوط 34 و 55). زمانیکه برنامه با باخت تمام می‌شود، برنامه متغیر **gameStatus** را با **CONTINUE** تنظیم می‌کند (خط 42) تا نشان داده شود که تاس‌ها باید مجددا پرتاب شوند.

یک نوع شمارشی پرطرفدار دیگر در زیر آورده شده است

```
enum Months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NPV,
DEC};
```

که در آن نوع تعریف شده از سوی کاربر، **Month** با ثابت‌های شمارشی که نشاندهنده ماه‌های سال هستند، ایجاد شده است. اولین مقدار در نوع شمارشی فوق بطور صریح با 1 تنظیم شده است، سایر مقادیر به ترتیب هر یک، یک واحد افزایش یافته و در نتیجه مقادیر1 الی 12 تولید می‌شوند. به هر ثابت نوع شمارشی می‌توان یک عدد صحیح در تعریف شمارشی تخصیص داد، و ثابت‌های شمارشی بعد از آن با مقداری به اندازه یک واحد بیشتر از مقدار قبل از خود خواهند داشت. این حالت تا زمانیکه مجددا بصورت صریح مقداری مشخص گردد ادامه می‌یابد.

پس از اولین پرتاب، اگر بازی با برد یا باخت همراه شود، برنامه از بدنه عبارت **while** (خطوط 59-49) پرش می‌کند، چرا که **gameStatus** برابر با **CONTINUE** نیست. برنامه با عبارت **if…else** موجود در خطوط 65-62 ادامه پیدا می‌کند. اگر **gameStatus** برابر با **WON** باشد جمله "Player wins" و اگر **gameStatus** برابر با **LOST** باشد، جمله "Player loses" چاپ می‌شود.

پس از اولین پرتاب اگر بازی به پایان نرسد برنامه مجموع را در **mypoint** ذخیره می‌کند(خط 43). اجرا با عبارت **while** ادامه پیدا می‌کند، زیرا **gameStatus** برابر با **CONTINUE** است. در جریان هر تکرار **while**، برنامه مبادرت به فراخوانی **rollDice** برای تولید **sum** جدید می‌کند. اگر **sum** با **myPoint** مطابقت کند، برنامه، مقدار **gameStatus** را به **WON** تغییر داده (خط 55)، تست **while** برقرار نشده، عبارت **if…else** جمله "Player wins" را چاپ کرده و اجرا خاتمه می‌یابد. اگر **sum** معادل با 7 باشد، برنامه، مقدار **gameStatus** را به **LOST** تغییر داده (خط 58)، تست **while** برقرار شده، عبارت **if…else** جمله "Player loses" را چاپ کرده و اجرا خاتمه می‌یابد.

به کاربرد جالب انواع مکانیزم‌های کنترلی که تا بدین مرحله مورد بحث قرار داده‌ایم توجه کنید. برنامه craps از دو تابع **rollDice** و **main** به همراه عبارات **switch**، **while**، **if…else**، **if…else** تودرتو و **if** تودرتو استفاده کرده است.

برنامه‌نویسی ایده‌ال

استفاده از نوع‌های شمارشی بجای ثابت‌های صحیح، می‌تواند وضوح برنامه‌ها را افزایش داده و نگهداری آنها را بهتر سازد. می‌توانید مقدار یک ثابت شمارشی را یک بار و در اعلان نوع شمارشی مشخص نمایید.

خطای برنامه‌نویسی

تخصیص معادل صحیح یک ثابت شمارشی به متغیری از نوع شمارشی، خطای کامپایل است.

خطای برنامه‌نویسی

پس از تعریف ثابت شمارشی، مبادرت به تخصیص یک مقدار دیگر به ثابت شمارشی، خطای کامپایل است.

۹-٦ کلاس‌های ذخیره‌سازی

برنامه‌هایی که تا بدین مرحله مشاهده کرده‌اید، از شناسه‌ها برای اسامی متغیرها استفاده می‌کردند. صفات متغیرها شامل نام، نوع، اندازه و مقدار است. همچنین در این فصل از شناسه‌ها بعنوان اسامی توابع تعریف شده از سوی کاربر استفاده شده است. در واقع، هر شناسه در یک برنامه دارای صفات دیگری شامل *کلاس ذخیره‌سازی*، قلمرو و *پیوند* (*linkage*) است.

زبان C++ پنج تصریح کننده کلاس ذخیره‌سازی تدارک دیده است: **auto**، **register**، **extern**، **mutable** و **static**. در این بخش کلاس‌های ذخیره‌سازی **auto**، **register**، **extern** و **static** مورد بحث قرار خواهند گرفت.

کلاس‌های ذخیره‌سازی، قلمرو و پیوند

شناسه یک کلاس ذخیره‌سازی، تعیین کننده مدت زمانی است، که در آن شناسه در حافظه وجود دارد. برخی از شناسه‌ها مدت زمان کوتاهی در حافظه وجود دارند، برخی به تناوب ایجاد و نابود می‌شوند و بقیه در کل مدت زمان اجرای برنامه در حافظه وجود دارند. در این بخش به بررسی دو کلاس ذخیره‌سازی می‌پردازیم: **static** و **automatic**.

قلمرو یک شناسه مکانی است که شناسه از آن قسمت در برنامه، می‌تواند مورد مراجعه قرار گیرد. تعدادی از شناسه‌ها می‌توانند در سرتاسر یک برنامه مورد مراجعه قرار گیرند، برخی دیگر می‌توانند فقط در قسمت‌های محدودی از یک برنامه مورد مراجعه قرار گیرند. بخش ۱۰-۶ در ارتباط با قلمرو شناسه‌ها است.

پیوند یک شناسه تعیین می‌کند که آیا شناسه فقط در فایل منبع که در آن اعلان شده است شناخته شود یا در میان فایل‌های مضاعف که کامپایل شده‌اند و سپس به یکدیگر لینک شده‌اند هم شناخته شود. کلاس ذخیره‌سازی یک شناسه در تعیین کلاس ذخیره‌سازی و پیوند آن نقش دارد.

رده‌بندی کلاس ذخیره‌سازی

می‌توان کلاس‌های ذخیره‌سازی را به دو گروه یا رده تقسیم کرد: کلاس ذخیره‌سازی اتوماتیک و کلاس ذخیره‌سازی استاتیک. از کلمات کلیدی **auto** و **register** برای اعلان متغیرهایی از کلاس ذخیره‌سازی اتوماتیک استفاده می‌شود. چنین متغیرهایی زمانی ایجاد می‌شوند که اجرای برنامه وارد بلوکی شود که آنها در آن تعریف شده‌اند، این متغیرها تا زمانی که بلوک فعال باشد، وجود خواهند داشت و زمانیکه برنامه از بلوک خارج شود، نابود خواهند شد.

متغیرهای محلی

فقط متغیرهای محلی یک تابع می‌توانند از کلاس ذخیره‌سازی اتوماتیک باشند. معمولا پارامترها و متغیرهای محلی یک تابع، از نوع کلاس ذخیره‌سازی اتوماتیک هستند. تصریح کننده کلاس ذخیره سازی

اتوماتیک بصورت صریح متغیرها را از کلاس ذخیره‌سازی اتوماتیک اعلان می‌کند. برای مثال، اعلان زیر بر این نکته دلالت دارد که متغیرهای **x** و **y** متغیرهای محلی از کلاس ذخیره‌سازی اتوماتیک هستند، این متغیرها فقط در نزدیکترین جفت براکت، بدنه تابعی که در آن تعریف شده اند وجود دارند:

```
auto double x, y;
```

متغیرهای محلی بصورت پیش فرض دارای کلاس ذخیره‌سازی اتوماتیک می‌باشند، از اینرو کلمه کلیدی **auto** بندرت بکار گرفته می‌شود. در مابقی متن، به متغیرهای کلاس ذخیره‌سازی اتوماتیک، فقط با نام متغیرهای اتوماتیک اشاره خواهیم کرد.

کارائی

ذخیره‌سازی اتوماتیک، وسیله‌ای برای صرفه‌جویی در مصرف حافظه است، چرا که متغیرهای کلاس ذخیره‌سازی اتوماتیک فقط زمانی در حافظه وجود دارند که بلوکی که در آن تعریف شده‌اند در حال اجرا باشد.

مهندسی نرم‌افزار

ذخیره‌سازی اتوماتیک مثالی از اصل حداقل مجوز دسترسی است، که از اصول بنیادین و خوب مهندسی نرم افزار است. بر پایه این اصل آن میزان از حق دسترسی باید به کد اعطا شود که برای انجام وظیفه تعیین شده به آن نیاز دارد و نه بیشتر. به چه دلیلی باید متغیرهایی در حافظه ذخیره و دسترس قرار دهیم که مورد نیاز نیستند؟

متغیرهای *register*

معمولا داده‌ها در نسخه زبان ماشین یک برنامه، برای انجام محاسبات و سایر پردازش‌ها به ثبات‌ها یا رجیسترها بارگذاری می‌شوند.

کارائی

*تصریح کننده کلاس ذخیره‌سازی **register** می‌تواند قبل از اعلان یک متغیر اتوماتیک قرار گیرد تا به کامپایلر اعلان کند که متغیر را در یکی از ثبات‌های سخت افزاری کامپیوتر که سرعت بسیار زیادی دارند ذخیره سازد و نه در حافظه. اگر متغیرهای پرکاربردی همانند شمارنده‌ها و مجموع‌ها در ثبات‌های سخت‌افزاری نگهداری شوند، سربار بارگذاری متناوب متغیرها از حافظه به ثبات‌ها و برگشت نتایج به حافظه مرتفع می‌شود.*

خطای برنامه‌نویسی

*استفاده از چندین کلاس ذخیره‌سازی برای یک شناسه خطای نحوی است. به یک شناسه می‌توان فقط یک کلاس ذخیره‌سازی اعمال کرد. برای مثال، اگر از **register** استفاده کنید ، دیگر **auto** را نمی توان بکار گرفت.*

کامپایلر می‌تواند اعلان‌های **register** را نادیده بگیرد. برای مثال، امکان دارد تعداد کافی از ثبات‌ها برای استفاده کامپایلر موجود نباشد. تعریف زیر پیشنهاد می‌دهد که متغیر صحیح **counter** در یکی از ثبات‌های

کامپیوتر قرار داده شود، صرفنظر از اینکه آیا کامپایلر این عمل را انجام دهد یا خیر، **counter** با 1 مقداردهی اولیه شده است :

```
register int counter = 1;
```

کلمه کلیدی **register** فقط می‌تواند با متغیرهای محلی و پارامترهای تابع بکار گرفته شود.

کارائی

*غالبا، ضرورتی به استفاده از ثبات نیست. کامپایلرهای بهینه شده امروزی قادر به شناسایی متغیرهایی هستند که متناوبا بکار گرفته می‌شوند و می‌توانند بدون اینکه نیازی باشد که برنامه‌نویس متغیری را از نوع **register**اعلان کند، تعیین می‌کنند که متغیر در ثبات قرار داده شود یا خیر.*

کلاس ذخیره‌سازی استاتیک

کلمات کلیدی **extern** و **static** شناسه‌های برای توابع و متغیرهای کلاس ذخیره‌سازی استاتیک اعلان می‌کنند. متغیرهای کلاس ذخیره‌سازی استاتیک از نقطه‌ای که برنامه شروع می‌شود، به وجود می‌آیند و طول عمرآنها تا اتمام برنامه است. حافظه یک متغیر کلاس ذخیره‌سازی استاتیک، به هنگام شروع اجرای برنامه اخذ می‌شود. چنین متغیری یک بار در زمان اعلان آن مقداردهی اولیه می‌شود. در مورد توابع، نام تابع در زمان شروع اجرای برنامه به وجود می‌آید، همانند همه توابع دیگر. با این وجود، حتی اگر متغیرها و نام توابع از زمان شروع اجرای برنامه وجود داشته باشند، این بدان معنی نیست که این شناسه‌ها می‌توانند در سرتاسر برنامه بکار گرفته شوند. کلاس ذخیره‌سازی و قلمرو (مکانی که نام می‌تواند بکار گرفته شود) مباحث جدا از یکدیگر هستند و در بخش ۱۰-۶ به این موضوع پرداخته شده است.

شناسه با کلاس ذخیره‌سازی استاتیک

دو نوع شناسه برای کلاس ذخیره‌سازی استاتیک وجود دارد، شناسه‌های خارجی (همانند اسامی متغیرهای سراسری و توابع سراسری) و متغیرهای محلی اعلان شده با تصریح کننده کلاس ذخیره‌سازی استاتیک. متغیرهای سراسری با قرار دادن اعلان‌های متغیر در خارج از تعریف تابع یا کلاس ایجاد می‌شوند. متغیرهای سراسری مقادیر خود را در کل زمان اجرای برنامه حفظ می‌کنند. متغیرهای سراسری و توابع سراسری می‌توانند از طرف هر تابعی که بعد از اعلان یا تعریف آنها در فایل منبع ظاهر می‌شود، بکار گرفته شود.

مهندسی نرم‌افزار

*اعلان یک متغیر سراسری بجای متغیر محلی، امکان رخ دادن اثرات جانبی ناخواسته دارد، زمانیکه تابعی بدون نیاز به این متغیر، به صورت تصادفی یا عمدی به تغییر مقدار آن اقدام کند. اینحالت مثال دیگری از اصل اعطا حداقل مجوزها است. بطور کلی، به جز برای منابع سراسری واقعی همانند **cin** و **cout**، از بکار بردن متغیرهای سراسری اجتناب نمائید، مگر در مواقعی که استفاده از آن سبب افزایش کارآیی برنامه گردد.*

مهندسی نرم‌افزار

متغیرهایی که فقط در یک تابع خاص بکار گرفته می‌شوند، باید بصورت متغیرهای محلی بجای متغیرهای سراسری اعلان شوند.

متغیرهای محلی اعلان شده با کلمه‌کلیدی **static** فقط در تابعی که در آن اعلان شده‌اند شناخته می‌شوند، اما، برخلاف متغیرهای اتوماتیک، متغیرهای محلی استاتیک مقادیر خود را حتی زمانیکه اجرای برنامه از تابع به فراخوان تابع برگشت داده می‌شود نیز حفظ می‌گردد. بار دیگر که تابع فراخوانی شود، متغیرهای محلی استاتیک حاوی مقادیری هستند که در آخرین اجرای تابع داشتند. عبارت زیر متغیر محلی **count** را بصورت استاتیک اعلان و با 1 مقداردهی اولیه کرده است:

```
static int count = 1;
```

تمام متغیرهای عددی از کلاس ذخیره‌سازی استاتیک با صفر مقدار دهی اولیه می‌شوند، اگر بصورت صریح از طرف برنامه‌نویس مقداردهی اولیه نشده باشند، اما مقداردهی اولیه تمامی متغیرها بصورت صریح کار چندان مناسبی نیست.

تصریح کننده‌های کلاس ذخیره‌سازی **extern** و **static** زمانیکه بصورت صریح با شناسه‌های خارجی همانند اسامی متغیرهای سراسری و توابع سراسری بکار گرفته می‌شوند، معنی خاصی پیدا می‌کنند.

۱۰-٦ قوانین قلمرو

به بخشی از برنامه که یک شناسه در آن می‌تواند بکار گرفته شود، قلمرو آن شناسه گفته می‌شود. برای مثال، هنگامی که یک متغیر محلی در یک بلوک اعلان می‌شود، آن متغیر فقط می‌تواند در آن بلوک و در بلوک‌هایی که بصورت تو در تو دورن آن بلوک قرار دارند، مورد مراجعه قرار گیرد. در این بخش به بررسی چهار قلمرو موجود برای یک شناسه می‌پردازیم: قلمرو تابع، قلمرو فایل، قلمرو بلوکی و قلمرو نمونه اولیه تابع.

شناسه اعلان شده در خارج از تابع یا کلاس دارای قلمرو فایل است. چنین شناسه‌ای از مکانی که در آن اعلان شده تا انتهای فایل در تمامی توابع شناخته می‌شود. متغیرهای سراسری، تعریف تابع و نمونه‌های اولیه تابع قرار گرفته در خارج از تابع، همگی دارای قلمرو فایل می‌باشند.

برچسب‌ها (شناسه‌هایی که پس از آنها یک کولن قرار داده می‌شود همانند :start) تنها شناسه‌های با قلمرو تابع هستند. برچسب‌ها می‌توانند در هر جای تابع که در آن ظاهر می‌شوند بکار گرفته شوند، اما در خارج از بدنه تابع نمی‌توانند مورد مراجعه قرار گیرند. از برچسب‌ها در عبارات goto استفاده می‌شود. برچسب‌ها پیاده‌سازی کننده جزئیاتی هستند که توابع آنها را از یکدیگر پنهان نگه می‌دارند.

شناسه‌هایی که درون یک بلوک اعلان شده‌اند، دارای قلمرو بلوکی هستند. قلمرو بلوکی از مکان اعلان شناسه آغاز شده و در مکانی که براکت بسته (}) بلوکی که شناسه در آن اعلان شده قرار دارد، خاتمه

می‌پذیرد. متغیرهای محلی دارای قلمرو بلوکی هستند، همانند پارامترهای تابع که متغیرهای محلی تابع نیز می‌باشند. هر بلوک می‌تواند حاوی اعلان متغیرها باشد. زمانیکه بلوک‌ها بصورت تودرتو هستند و شناسه‌ای همنام با شناسه موجود در بلوک داخلی وجود داشته باشد، شناسه در بلوک خارجی تا زمانی که بلوک داخلی خاتمه یابد، پنهان می‌شود. در زمان اجرای بلوک داخلی، بلوک داخلی مقدار شناسه محلی خود را می‌بیند و به مقدار شناسه همنام خود در بلوک در برگیرنده خود توجهی ندارد. متغیرهای محلی استاتیک هم دارای قلمرو بلوکی هستند، حتی اگر از زمان شروع برنامه وجود داشته باشند. مدت زمان ذخیره‌سازی تاثیری در قلمرو یک شناسه ندارد.

تنها شناسه‌هایی که دارای قلمرو نمونه‌های اولیه تابع می‌باشند، شناسه‌های هستند که در لیست پارامتری نمونه اولیه تابع بکار رفته‌اند. همانطوری که قبلا هم گفته شد، در نمونه‌های اولیه تابع در لیست پارامتری نیازی به حضور نام نیست و فقط نوع آنها مورد نیاز است. اسامی بکار رفته در لیست پارامتری یک نمونه اولیه تابع از طرف کامپایلر نادیده گرفته می‌شوند. شناسه‌هایی که در نمونه اولیه یک تابع بکار برده می‌شوند، می‌توانند در هر کجای برنامه مورد استفاده مجدد قرار گیرند، بدون اینکه ابهامی بوجود آورند. در یک نمونه اولیه منفرد، یک شناسه خاص فقط می‌تواند یکبار بکار برده شود.

خطای برنامه‌نویسی

استفاده از نام مشابه برای یک شناسه در بلوک داخلی و بلوک خارجی، زمانیکه برنامه‌نویس می‌خواهد به شناسه موجود در بلوک خارجی دسترسی پیدا کند، معمولا خطای منطقی بدنبال دارد.

برنامه‌نویسی ایده‌ال

از نامگذاری که سبب می‌شود اسامی در قلمرو خارجی پنهان شوند اجتناب کنید.

برنامه شکل ۱۲-۶ به توصیف مباحث قلمرو در متغیرهای سراسری، متغیرهای محلی اتوماتیک و متغیرهای محلی استاتیک پرداخته است.

```
1  // Fig. 6.12: fig06_12.cpp
2  // A scoping example.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  void useLocal( void ); // function prototype
8  void useStaticLocal( void ); // function prototype
9  void useGlobal( void ); // function prototype
10
11 int x = 1; // global variable
12
13 int main()
14 {
15    int x = 5; // local variable to main
16
17    cout << "local x in main's outer scope is " << x << endl;
18
19    { // start new scope
20       int x = 7; // hides x in outer scope
21
22       cout << "local x in main's inner scope is " << x << endl;
```

```
23    } // end new scope
24
25    cout << "local x in main's outer scope is " << x << endl;
26
27    useLocal(); // useLocal has local x
28    useStaticLocal(); // useStaticLocal has static local x
29    useGlobal(); // useGlobal uses global x
30    useLocal(); // useLocal reinitializes its local x
31    useStaticLocal(); // static local x retains its prior value
32    useGlobal(); // global x also retains its value
33
34    cout << "\nlocal x in main is " << x << endl;
35    return 0; // indicates successful termination
36 } // end main
37
38 // useLocal reinitializes local variable x during each call
39 void useLocal( void )
40 {
41    int x = 25; // initialized each time useLocal is called.
42
43    cout << "\nlocal x is " << x << " on entering useLocal" << endl;
44    x++;
45    cout << "local x is " << x << " on exiting useLocal" << endl;
46 } // end function useLocal
47
48 // useStaticLocal initializes static local variable x only the
49 // first time the function is called; value of x is saved
50 // between calls to this function
51 void useStaticLocal( void )
52 {
53    static int x = 50; // initialized first time useStaticLocal is called
54
55    cout << "\nlocal static x is " << x << " on entering useStaticLocal"
56       << endl;
57    x++;
58    cout << "local static x is " << x << " on exiting useStaticLocal"
59       << endl;
60 } // end function useStaticLocal
61
62 // useGlobal modifies global variable x during each call
63 void useGlobal( void )
64 {
65    cout << "\nglobal x is " << x << " on entering useGlobal" << endl;
66    x *= 10;
67    cout << "global x is " << x << " on exiting useGlobal" << endl;
68 } // end function useGlobal
```

```
local x in main's outer scope is 5
local x in main's inner scope is 7
local x in main's outer scope is 5

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```

اعلان و با 5 مقداردهی اولیه کردنه است. خط 17 برای نشان دادن این موضوع که **x** سراسری در **main** پنهان است، این متغیر را چاپ کرده است. سپس، خطوط 23-19 یک بلوک جدید در **main** تعریف می‌کنند که در آن یک متغیر محلی بنام **x** اعلان و با 7 مقداردهی اولیه شده است (خط 20). خط 22 این متغیر را چاپ می‌کند تا نشان دهد که **x** در بلوک خارجی **main** پنهان است. زمانیکه بلوک به پایان می‌رسد، متغیر **x** با مقدار 7 بصورت اتوماتیک نابود می‌شود. سپس، خط 25 متغیر محلی **x** موجود در بلوک خارجی **main** را چاپ می‌کند تا نشان داده شود که این متغیر دیگر پنهان شده نیست.

برای توصیف قلمروهای دیگر، برنامه سه تابع تعریف کرده است، آرگومانی دریافت نکرده و چیزی برگشت نمی‌دهند. تابع **useLocal** (خطوط **46-39**) متغیر اتوماتیک **x** را اعلان (خط 41) و با 25 مقداردهی اولیه کرده است. زمانیکه برنامه **useLocal** را فراخوانی می‌کند، تابع مقدار متغیر را چاپ کرده، یک واحد آنرا افزایش داده و قبل از آنکه تابع، کنترل برنامه را فراخوان خود برگشت دهد، مجددا آنرا چاپ می‌کند. هر بارکه برنامه این تابع را فراخوانی می‌کند، تابع مجددا متغیر اتوماتیک **x** را ایجاد و با 25 مقداردهی اولیه می‌کند.

تابع **useStaticLocal** (خطوط 60-51) متغیر استاتیک **x** را و آن را با 50 مقداردهی اولیه کرده است. متغیرهای محلی اعلان شده بصورت استاتیک مقادیر خود را حتی هنگامی که خارج از قلمرو هستند (یعنی در تابعی که در آن اعلان شده و آن تابع در حال اجرا نباشد) حفظ می‌کنند. زمانیکه برنامه مبادرت به فراخوانی **useStaticLocal** می‌کند، تابع مقدار **x** را چاپ کرده، آنرا یک واحد افزایش داده و قبل از آنکه تابع مبادرت به برگرداندن کنترل برنامه به تابع فراخوان خود نماید، مجددا آنر چاپ می‌کند. در فراخوانی بعدی این تابع، متغیر محلی استاتیک **x** حاوی مقدار 51 است. مقداردهی بکار رفته در خط 53 فقط یکبار رخ می‌دهد، اولین بار که **useStaticLocal** فراخوانی می‌شود.

تابع **useGlobal** (خطوط 68-63) هیچ متغیری را اعلان نکرده است. بنابر این، هنگامی که به متغیر **x** مراجعه می‌کند، **x** سراسری (**main** قبلی) بکار گرفته می‌شود. زمانیکه برنامه تابع **useGlobal** را فراخوانی می‌کند، تابع متغیر سراسری **x** را چاپ کرده، آنرا در 10 ضرب و قبل از آنکه تابع مبادرت به برگرداندن کنترل برنامه به فراخوان خود نماید، مجددا آن را چاپ می‌کند. بار دیگر که برنامه، تابع **useGlobal** را فراخوانی می‌کند، متغیر سراسری حاوی مقدار تغییر یافته، 10 است. پس از آنکه هر یک از توابع **useLocal**، **useStaticLocal** و **useGlobal** دو بار اجرا شدند، برنامه مقدار متغیر محلی **x** موجود در **main** را مجددا چاپ می‌کند تا نشان دهد که هیچ کدامیک از فراخوانی‌های تابع مقدار **x** موجود در **main** را تغییر نداده‌اند، چرا که تمام توابع به متغیرهای موجود در قلمروهای دیگر مراجعه دارند.

۱۱-٦ عملکرد پشته فراخوانی و ثبت فعالیت‌ها

برای درک نحوه فراخوانی تابع در C++، ابتدا نیاز است به بررسی ساختمان داده (یعنی کلکسیونی از ایتم‌های داده مرتبط با هم) بنام پشته بپردازیم. می‌توانید پشته را همانند تعدادی بشقاب که روی هم قرار گرفته‌اند، در نظر بگیرید. هنگامی که بشقابی بر روی سایر بشقاب‌ها قرار داده می‌شود، معمولا در بالای بشقاب‌ها جای داده می‌شود (به این عمل گذاشتن یا ***push*** کردن بشقاب در پشته گفته می‌شود). به طور مشابه، هنگامی که یک بشقاب از روی بشقاب‌ها برداشته می‌شود، معمولا از بالای بشقاب‌ها برداشته می‌شود (به این عمل برداشتن یا ***pop*** کردن بشقاب از پشته گفته می‌شود). پشته‌ها بعنوان ساختمان‌های داده LIFO (last-in, last-out) شناخته می‌شوند، به این معنی که آخرین ایتمی که در پشته گذاشته می‌شود، اولین ایتمی است که از پشته برداشته می‌شود.

یکی از مهمترین مکانیزم‌های که باید از طرف دانشجویان کامپیوتر درک شود، پشته فراخوانی تابع است (بعنوان پشته اجرای برنامه هم نامیده می‌شود). این ساختمان داده که در پشت صحنه کار می‌کند، از مکانیزم فراخوانی/برگشت تابع پشتیبانی می‌کند. همچنین از ایجاد، نگهداری و نابود کردن متغیرهای اتوماتیک توابع فراخوانی شده پشتیبانی می‌نماید. رفتار LIFO پشته‌ها را با مثال بشقاب‌ها توضیح دادیم. همانطوری که در برنامه‌های شکل ۱۴-۶ الی ۱۶-۶ خواهید دید، این رفتار LIFO دقیقا همان کاری است که تابع ، در زمان بازگشت به تابع فراخوان خود انجام می‌دهد.

زمانیکه تابعی فراخوانی می‌شود، امکان دارد قبل از آنکه برگشت یابد، توابع دیگری را فراخوانی کند، به همین ترتیب امکان دارد این توابع هم، قبل از برگشت به تابع فراخوان خود، توابع دیگری را فراخوانی کنند. سرانجام هر تابعی باید کنترل را به تابع فراخوان خود باز گرداند. از اینرو، باید به روشی، آدرس‌های بازگشت را که هر تابع برای بازگرداندن کنترل به تابع فراخوان خود به آنها نیاز دارد ردگیری و نگهداری کنیم. پشته فراخوان تابع یک ساختمان داده عالی برای رسیدگی به این اطلاعات است. هر زمانیکه تابعی مبادرت به فراخوانی تابع دیگر می‌کند، یک ورودی در پشته ثبت می‌شود یا به عبارتی به پشته push می‌گردد. این ورودی، یک *فریم پشته* (*stack frame*) یا *ثبت فعالیت‌ها* نامیده می‌شود، و حاوی آدرس بازگشت است که تابع فراخوانده شده برای برگشت به تابع فراخوان خود به آن نیاز دارد. همچنین فریم پشته حاوی برخی از اطلاعات دیگر هم است که به زودی در مورد آنها صحبت خواهیم کرد. اگر تابع فراخوانده شده، بجای فراخوانی تابع دیگری قبل از بازگشت، به محل فراخوانی خود برگشت یابد، فریم پشته فراخوانی تابع pop شده، و کنترل به آدرس بازگشت، در فریم پشته pop شده انتقال داده می‌شود.

قابلیت پشته فراخوان در این است که هر تابع فراخوانی شده همیشه اطلاعات مورد نیاز خود برای برگشت به فراخوان خود را در بالای پشته فراخوان پیدا می‌کند. و اگر تابعی، تابع دیگری را فراخوانی کند، به ساده‌گی یک فریم پشته برای تابع جدیدا فراخوانی شده، در پشته فراخوان push می‌شود. از اینرو، هم

اکنون آدرس بازگشت مورد نیاز برای برگشت تابع جدیدا فراخوانی شده به فراخوان خود در بالای پشته قرار دارد.

فریم‌های پشته مسئولیت مهم دیگری هم بر عهده دارند. اکثر توابع دارای متغیرهای اتوماتیک همانند پارامترها و متغیرهای محلی که تابع اعلان می‌کند، هستند. متغیرهای اتوماتیک باید در زمان اجرای یک تابع وجود داشته باشند. اگر تابع مبادرت به فراخوانی توابع دیگری کند، این متغیرها باید فعال نگه داشته شوند. ام زمانیکه تابع فراخوانده شده به فراخوان خود باز می‌گردد، نیاز است تا متغیرهای اتوماتیک تابع فراخوانده شده از بین بروند. فریم پشته تابع فراخوانده شده، مکانی عالی برای رزرو حافظه برای متغیرهای اتوماتیک تابع فراخوانده شده است. این فریم پشته تا زمانیکه تابع فراخوانده شده فعال است وجود خواهد داشت. زمانیکه تابع فراخوانده شده برگشت پیدا می‌کند و دیگر نیازی به متغیرهای اتوماتیک محلی خود ندارد، فریم پشته آن از پشته pop شده، و دیگر این متغیرهای اتوماتیک محلی در برنامه شناخته نمی‌شوند.

البته، مقدار حافظه در کامپیوتر با محدودیت همراه است، از اینرو فقط مقدار مشخصی از حافظه می‌تواند برای ذخیره ثبت فعالیت‌ها در پشته فراخوان تابع بکار گرفته شود. اگر تعداد فراخوانی‌های توابع بیش از مقدار و توان حافظه در نظر گرفته شده به این منظور باشد، با خطای بنام *سرریز پشته* (*stack overflow*) مواجه خواهید شد.

پشته فراخوانی تابع در عمل

بسیار خوب، همانطوری که مشاهده کردیم، پشته فراخوانی و ثبت فعالیت‌ها، از مکانیزم فراخوانی/برگشت دادن تابع، ایجاد و نابود کردن متغیرهای اتوماتیک پشتیبانی می‌کنند. اجازه دهید به بررسی پشتیبانی پشته فراخوانی از عملیات تابع **square** که توسط **main** فراخوانی می‌شود بپردازیم (خطوط 17-11 از شکل۱۳-۶). ابتدا سیستم عامل تابع **main** را فراخوانی می‌کند، اینکار سبب push شدن یک رکورد فعالیت در پشته می‌شود (در شکل ۱۴-۶ نشان داده شده است). رکورد فعالیت به **main** نحوه برگشت به سیستم عامل را بیان کرده (یعنی انتقال به آدرس برگشت **R1**) و حاوی فضا برای متغیر اتوماتیک **main** است (یعنی **a** که با 10 مقداردهی اولیه شده است).

```
1  // Fig. 6.13: fig06_13.cpp
2  // square function used to demonstrate the function
3  // call stack and activation records.
4  #include <iostream>
5  using std::cin;
6  using std::cout;
7  using std::endl;
8
9  int square( int ); // prototype for function square
10
11 int main()
12 {
13    int a = 10; // value to square (local automatic variable in main).
14
15    cout << a << " squared: " << square( a ) << endl;// display a squared
16    return 0; // indicate successful termination
```

```
17 } // end main
18
19 // returns the square of an integer
20 int square( int x ) // x is a local variable
21 {
22    return x * x; // calculate square and return result
23 } // end function square
```

```
10 squared: 100
```

شکل ۱۳-٦ | استفاده از تابع square برای توصیف عملکرد پشته فراخوان تابع و رکورد فعالیت.

شکل ١٤-٦ | پشته فراخوانی تابع پس از اینکه سیستم‌عامل تابع main را برای اجرای برنامه فراخوانی کرده است.

حال تابع **main**، تابع **square** در خط 15 از شکل ۱۳-۶ را قبل از برگشت به سیستم عامل فراخوانی می‌کند. این عمل سبب می‌شود تا یک فریم پشته برای **square** (خطوط 23-20) در پشته فراخوانی تابع push شود (شکل ۱۵-۶). این فریم پشته، حاوی آدرس برگشتی است که **square** برای برگشت به **main** (یعنی **R2**) و حافظه برای متغیر اتوماتیک **x** به آن نیاز دارد.

پس از اینکه **square** مربع آرگومان خود را بدست آورد، نیاز به برگشت به **main** دارد و دیگر نیازی به حافظه برای متغیر اتوماتیک خود یعنی **x** ندارد. از اینرو pop بر روی پشته اعمال می‌شود، امکان برگشت **square** به **main** (یعنی **R2**) فراهم شده و متغیر اتوماتیک **square** از بین می‌رود. شکل ۱۶-۶ پشته فراخوانی تابع پس از **pop** شدن رکورد فعالیت **square** است.

اکنون تابع **main** مبادرت بنمایش نتیجه فراخوانی **square** می‌کند (خط 15)، سپس عبارت **return** را اجرا می‌نماید (خط 16). این عمل سبب می‌شود تا رکورد فعالیت **main** از پشته pop شود. با اینکار آدرس مورد نیاز برای بازگشت به سیستم عامل (**R1** در شکل ۱۴-۶) به **main** داده شده و سبب می‌شود حافظه متغیر اتوماتیک **main** (یعنی **a**) در دسترس نباشد.

حال مشاهده کردید که چگونه ساختمان داده پشته مبادرت به پیاده‌سازی مکانیزمی کلیدی می‌کند که از اجرای برنامه‌ها پشتیبانی می‌نماید. ساختمان‌های داده کاربردهای مهمی در علم کامپیوتر دارند. در فصل بیست‌ویکم در ارتباط با پشته‌ها، صف‌ها، لیست‌ها، درخت‌ها و سایر ساختمان‌های داده صحبت خواهیم کرد.

شکل ١٥-٦ | پشته فراخوانی تابع پس از اینکه main تابع square را برای انجام محاسبه فراخوانی کرده است.

شکل ١٦-٦ | پشته فراخوانی تابع پس از اینکه تابع square به main برگشت داده شده است.

۱۲-٦ توابع با لیست پارامتری تهی

در زبان C++، یک لیست پارامتری تهی با نوشتن **void** یا خالی گذاشتن پارانتزها مشخص می‌شود. عبارت زیر

```
void print();
```

تصریح می‌کند که تابع **print** آرگومانی دریافت نمی‌کند و مقداری را هم برگشت نمی‌دهد. برنامه شکل ۱۷-۶ هر دو روش اعلان و استفاده از توابع با لیست‌های پارامتری تهی را نشان می دهد.

قابلیت حمل

مفهوم لیست پارامتری تهی تابع در C++ بطور قابل توجهی متفاوت از C است. در زبان C، لیست پارامتری تهی به معنی است که بررسی کلیه آرگومان‌ها غیر فعال است (فراخوانی تابع می‌تواند هر آرگومانی را ارسال کند). در C++، بدین معنی است که تابع بطور صریح هیچ آرگومانی دریافت نمی‌کند. بنابر این، برنامه‌های C که از ویژگی استفاده می‌کنند ممکن است در هنگام کامپایل شدن در C++ خطاهای کامپایل تولید کنند.

```
1  // Fig. 6.17: fig06_17.cpp
2  // Functions that take no arguments.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  void function1(); // function that takes no arguments
8  void function2( void ); // function that takes no arguments
9
10 int main()
11 {
12    function1(); // call function1 with no arguments
13    function2(); // call function2 with no arguments
14    return 0; // indicates successful termination
15 } // end main
16
17 // function1 uses an empty parameter list to specify that
18 // the function receives no arguments
19 void function1()
20 {
21    cout << "function1 takes no arguments" << endl;
22 } // end function1
23
24 // function2 uses a void parameter list to specify that
25 // the function receives no arguments
26 void function2( void )
27 {
28    cout << "function2 also takes no arguments" << endl;
29 } // end function2
```

```
function1 takes no arguments
function2 also takes no arguments
```

شکل ۱۷-٦ | توابعی که آرگومان دریافت نمی‌کنند.

۱۳-٦ توابع inline

پیاده‌سازی یک برنامه به صورت مجموعه‌ای از توابع از نظر مهندسی نرم‌افزار کار مناسبی است، اما فراخوانی‌های تابع، سربارگذاری زمان اجرا بدنبال دارد. C++ برای کمک به کاهش سربارگذاری فراخوانی تابع، توابع **inline** را تدارک دیده است، به ویژه برای توابع کوچک. قرار دادن توصیف کننده **inline** قبل از نوع بازگشتی تابع در تعریف تابع، به کامپایلر توصیه می‌کند در محل استفاده از تابع (در زمان مناسب) یک کپی از کد تابع ایجاد و از فراخوانی تابع ممانعت کند. مشکل اینجاست که کپی‌های مضاعف از کد تابع در برنامه وارد می‌شود و اغلب سبب بزرگتر شدن برنامه می‌گردند، بجای اینکه یک

کپی منفرد از تابع که در هر بار فراخوانی تابع کنترل به آن ارسال شود. کامپایلر می‌تواند توصیفی کننده **inline** را نادیده گرفته و عموما نیز این کار را برای تمامی توابع به جز توابع کوچک انجام می‌دهد.

مهندسی نرم‌افزار

*هر تغییری در یک تابع **inline** مستلزم کامپایل مجدد تمامی سرویس‌گیرنده‌های تابع است. اینکار می‌تواند در توسعه و نگهداری برخی از برنامه‌ها قابل توجه باشد.*

برنامه‌نویسی ایده‌ال

*باید توصیف کننده **inline** فقط با توابع کوچک و پرکاربرد بکار گرفته شود.*

کارائی

*استفاده از توابع **inline** می‌تواند زمان اجرا را کاهش دهد، اما می‌تواند سایز برنامه را هم افزایش دهد.*

برنامه شکل ۱۸-۶ از یک تابع **inline** بنام **cube** (خطوط 14-11) برای محاسبه حجم مکعبی با ضلع **side** استفاده می‌کند. کلمه‌کلیدی **const** در لیست پارامتری تابع **cube** (خط 11) به کامپایلر اعلان می‌کند که تابع مبادرت به تغییر متغیر **side** نمی‌کند. با اینکار تضمین می‌شود که مقدار **side** در هنگام انجام محاسبه از طرف تابع تغییر داده نخواهد شد. جزئیات کلمه‌کلیدی **const** در فصل هفتم، فصل هشتم و فصل دهم توضیح داده شده است. توجه کنید که تعریف کامل تابع **cube** قبل از استفاده از آن در برنامه ظاهر شده است. انجام اینکار ضروری است، چراکه با انجام اینکار کامپایلر می‌داند که چگونه فراخوانی تابع **cube** را به کد **inline** آن گسترش دهد. به همین دلیل، معمولا توابع **inline** با قابلیت استفاده مجدد در فایل‌های سرآیند قرار داده می‌شوند ، از اینرو است که تعاریف آنها می‌توانند در هر فایل منبع که از آنها استفاده می‌کند، شامل گردد.

مهندسی نرم‌افزار

*باید توصیف کننده **const** را برای پیاده کردن اصل حداقل حق مجوز دسترسی بکار گرفت. استفاده از این اصل در توسعه نرم‌افزار می‌تواند زمان خطایابی و تاثیرات جانبی را به حداقل رسانده و تغییر و نگهداری از برنامه را آسان‌تر کند.*

```
1  // Fig. 6.18: fig06_18.cpp
2  // Using an inline function to calculate the volume of a cube.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  // Definition of inline function cube. Definition of function appears
9  // before function is called, so a function prototype is not required.
10 // First line of function definition acts as the prototype.
11 inline double cube( const double side )
12 {
13    return side * side * side; // calculate cube
14 } // end function cube
15
16 int main()
17 {
18    double sideValue; // stores value entered by user
19    cout << "Enter the side length of your cube: ";
```

```
20    cin >> sideValue; // read value from user
21
22    // calculate cube of sideValue and display result
23    cout << "Volume of cube with side "
24       << sideValue << " is " << cube( sideValue ) << endl;
25    return 0; // indicates successful termination
26 } // end main
```

```
Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```

شکل ۱۸-٦ | تابع inline که مبادرت به محاسبه حجم یک مکعب می‌کند.

١٤-٦ مراجعه و پارامترهای مراجعه

دو روش برای ارسال آرگومان به توابع در بسیاری از زبان‌های برنامه‌نویسی وجود دارد که عبارتند از "*ارسال با مقدار*" و "*ارسال با مراجعه*". هنگامی که یک آرگومان به روش مقدار ارسال می‌شود، یک کپی از مقدار آرگومان تهیه و به تابع فراخوانده شده ارسال می‌گردد (در پشته فراخوانی تابع). هر گونه تغییر در کپی، تاثیری در مقدار اصلی متغیر در فراخوان اعمال نمی‌کند. اینکار از تاثیرات جانبی و تصادفی که تا حد زیادی در توسعه سیستم‌های نرم‌فزاری صحیح و اطمینان بالا مشکل بوجود می‌آورند، جلوگیری می‌کند. تمامی آرگومانی‌های ارسالی تابدین مرحله از این فصل به روش ارسال با مقدار بودند.

کارائی

یکی از معایب ارسال به روش مقدار این است که اگر یک ایتم داده بزرگ ارسال گردد، کپی کردن آن داده می‌تواند مقدار قابل توجهی از زمان اجرا و فضای حافظه تلف کند.

پارامتر های مراجعه

این بخش به معرفی "*پارامترهای مراجعه یا ارجاعی*" می‌پردازد، یکی از دو روشی که C++ برای ارسال با مراجعه تدارک دیده است. در روش ارسال با مراجعه، فراخوان به تابع فراخوانی شده امکان دسترسی مستقیم به داده‌های خود و اصلاح آن داده‌ها را می‌دهد.

کارائی

روش ارسال با مراجعه در افزایش کارآیی موثر است، چرا که نیاز به سربارگذاری از کپی کردن میزان زیادی از داده‌ها در روش ارسال با مقدار را از بین می‌برد.

مهندسی نرم‌افزار

ارسال با مراجعه می‌تواند در کاهش امنیت تاثیر گذار باشد، چرا که تابع فراخوان می‌تواند داده‌های فراخوان خود را معیوب کند.

بعدا، نشان خواهیم داد که چگونه می‌توان از کارآیی ارسال به روش مراجعه استفاده کرده و با توجه به اصول مهندسی نرم‌افزار، از داده‌های فراخوان محافظت کرد.

یک پارامتر ارجاعی یک نام مستعار برای آرگومان متناظر خود در فراخوانی تابع است. برای نشان دادن اینکه پارامتر تابع به روش مراجعه ارسال می‌شود، کافیست پس از نوع پارامتر در نمونه اولیه تابع یک &

قرار داده شود، از همین روش به هنگام لیست کردن نوع پارامتر در سرآیند تابع هم استفاده کنید. برای مثال، اعلان زیر در سرآیند یک تابع

```
int &count
```

زمانیکه از راست به چپ خوانده شود به معنی "**count** یک مراجعه به یک **int** است" خواهد بود. در فراخوانی تابع، کافی است متغیر را با نام ذکر کنید تا به روش مراجعه ارسال شود. سپس، ذکر متغیر توسط نام پارامتر آن در بدنه تابع فراخوانده شده، در واقع یک مراجعه به متغیر اصلی در تابع فراخوان است، و متغیر اصلی می‌تواند مستقیما از طرف تابع فراخوانده شده تغییر داده شود. باز هم، باید نمونه اولیه و سرآیند تابع با هم موافق باشند.

ارسال آرگومان با مقدار و با مراجعه

برنامه شکل ۱۹-۶ به مقایسه روش ارسال با مقدار و ارسال با مراجعه با پارامترهای ارجاعی پرداخته است. سبک آرگومان‌ها در فراخوانی توابع **squareByValue** و **squareByReference** یکسان است، هر دو متغیر با نام خود در فراخوانی‌ها مشخص شده‌اند. بدون بررسی نمونه‌های اولیه تابع یا تعاریف تابع، نمی‌توان بر اساس فراخوانی‌ها تعیین کرد که تابع می‌تواند در آرگومان‌های خود تغییر بوجود آورد. بدلیل اینکه نمونه‌های اولیه تابع اجباری هستند، کامپایلر مشکلی در رفع این ابهام ندارد.

```
1  // Fig. 6.19: fig06_19.cpp
2  // Comparing pass-by-value and pass-by-reference with references.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int squareByValue( int ); // function prototype (value pass)
8  void squareByReference( int & ); // function prototype (reference pass)
9
10 int main()
11 {
12    int x = 2; // value to square using squareByValue
13    int z = 4; // value to square using squareByReference
14
15    // demonstrate squareByValue
16    cout << "x = " << x << " before squareByValue\n";
17    cout << "Value returned by squareByValue: "
18       << squareByValue( x ) << endl;
19    cout << "x = " << x << " after squareByValue\n" << endl;
20
21    // demonstrate squareByReference
22    cout << "z = " << z << " before squareByReference" << endl;
23    squareByReference( z );
24    cout << "z = " << z << " after squareByReference" << endl;
25    return 0; // indicates successful termination
26 } // end main
27
28 // squareByValue multiplies number by itself, stores the
29 // result in number and returns the new value of number
30 int squareByValue( int number )
31 {
32    return number *= number; // caller's argument not modified
33 } // end function squareByValue
34
35 // squareByReference multiplies numberRef by itself and stores the result
36 // in the variable to which numberRef refers in function main
37 void squareByReference( int &numberRef )
```

```
38 {
39    numberRef *= numberRef; // caller's argument modified
40 } // end function squareByReference
```

```
x = 2 before squareByValue
Value returend by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

شکل ۱۹-٦ | ارسال آرگومان به روش مقدار و مراجعه.

خطای برنامه‌نویسی

بدلیل اینکه پارامترهای ارجاعی فقط با نام در بدنه تابع فراخوانی شده ذکر می‌شوند، ممکن است برنامه‌نویس سهوا با پارامترهای ارجاعی بعنوان پارامترهای ارسالی با مقدار رفتار کند. اگر کپی‌های اصلی از متغیرها توسط تابع تغییر داده شوند، اینکار می‌تواند اثرات جانبی غیرمنتظره‌ای بوجود آورد.

در فصل هشتم در ارتباط با اشاره‌گرها صحبت خواهیم کرد، اشاره‌گرها یک روش جایگزین برای ارسال با مراجعه هستند که در آن سبک فراخوانی، به وضوح دلالت بر ارسال با مراجعه دارد.

کارائی

برای ارسال شی‌های بزرگ، از پارامتر ارجاعی ثابت برای شبیه‌سازی ظاهر و امنیت ارسال با مقدار استفاده کرده و از سربارگذاری ارسال یک کپی از شیء بزرگ اجتناب کنید.

مهندسی نرم‌افزار

برخی از برنامه‌نویسان زحمت اعلان پارامترهای ارسالی با مقدار را به عنوان ثابت را، حتی در صورتی که تابع فراخوانی شده نیازی به تغییر در آرگومان‌های ارسالی نداشته باشد، به خود نمی‌دهند. کلمه کلیدی ***const*** *در این زمینه فقط می‌تواند از یک کپی از آرگومان اصلی محافظت کند، نه خود آرگومان اصلی، که در زمان ارسال به روش مقدار در مقابل تغییرات توسط تابع فراخوان ایمن است.*

برای مشخص کردن یک مراجعه به یک ثابت، توصیف کننده **const** را قبل از مشخص کننده نوع در اعلان پارامتر قرار دهید.

به خط 37 از شکل ۱۹-۶ و به مکان & در لیست پارامتری تابع **squareByReference** توجه کنید. برخی از برنامه‌نویسان C++ ترجیح می‌دهند آنرا بصورت **int& numberRef** بنویسند.

مهندسی نرم‌افزار

به منظورافزایش وضوح و کارآیی، بسیاری از برنامه‌نویسان C++ ترجیح می‌دهند آرگومان‌های تغییرپذیر را با استفاده از اشاره‌گر، آرگومان‌های کوچک غیرقابل تغییر به را روش مقدار و آرگومان‌های بزرگ غیرقابل تغییر را به روش مراجعه به ثابت‌ها، به توابع ارسال کنند.

مراجعه‌ها بعنوان اسامی مستعار در درون یک تابع

مراجعه‌ها همچنین می‌توانند بعنوان اسامی مستعار برای متغیرهای دیگر در درون یک تابع بکار برده شوند (اگرچه آنها معمولا در توابعی به شکلی که در شکل ۱۹-۶ نشان داده شده بکار گرفته می‌شوند). برای مثال، کد

```
int count = 1; // declare integer variable count
int &cRef = count; // create cRef as an alias for count
cRef++; // increment count (using its alias cRef)
```

متغیر **count** را با استفاده از نام مستعار **cRef** یک واحد افزایش می‌دهد. متغیرهای ارجاعی باید در اعلان‌های خود مقداردهی اولیه شوند (شکل ۲۰-۶ و شکل ۲۱-۶) و نمی‌توانند مجددا بعنوان اسامی مستعار به متغیرهای دیگر تخصیص داده شوند. زمانیکه یک مراجعه بعنوان نام مستعار برای متغیر دیگری اعلان شود، کلیه عملیات‌های انجام شده بر روی نام مستعار در واقع بر روی متغیر اصلی انجام می‌شوند. در حقیقت نام مستعار، نام دیگری برای متغیر اصلی به شمار می‌آید. گرفتن آدرس یک مراجعه و مقایسه مراجعه‌ها سبب رخ دادن خطاهای نحوی نمی‌شود، بلکه، در واقع هر عملیاتی بر روی متغیری صورت می‌گیرد که مراجعه برای آن یک نام مستعار می‌باشد.مگر اینکه مراجعه به یک ثابت باشد، یک آرگومان ارجاعی بایستی یک *lvalue* (مثلا نام یک متغیر) باشد، و نه یک ثابت یا عبارتی که یک *rvalue* برگشت می‌دهد (مثلا نتیجه ی یک محاسبه). برای آشنایی با تعاریف کلمات *lvalue* و *rvalue* به بخش ۹-۵ مراجعه کنید.

```
1  // Fig. 6.20: fig06_20.cpp
2  // References must be initialized.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int x = 3;
10    int &y = x; // y refers to (is an alias for) x
11
12    cout << "x = " << x << endl << "y = " << y << endl;
13    y = 7; // actually modifies x
14    cout << "x = " << x << endl << "y = " << y << endl;
15    return 0; // indicates successful termination
16 } // end main
```

```
x = 3
y = 3
x = 7
y = 7
```

شکل ۲۰-٦ | مقداردهی اولیه و استفاده از مراجعه.

```
1  // Fig. 6.21: fig06_21.cpp
2  // References must be initialized.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int x = 3;
10    int &y; // Error: y must be initialized
```

```
11
12    cout << "x = " << x << endl << "y = " << y << endl;
13    y = 7;
14    cout << "x = " << x << endl << "y = " << y << endl;
15    return 0; // indicates successful termination
16 } // end main
```

Borland C++ command-line compiler error message:

```
Error E2304 C:\cppht5_examples\ch06\Fig06_21\fig06_21.cpp 10:
 Reference variable 'y' must be initialized in function main()
```

Microsoft Visual C++ compiler error message:

```
C:\cppht5_examples\ch06\Fig06_21\fig06_21.cpp(10): error C2530: 'y':
 reference must be initialized
```

GNU C++ compiler error message:

```
fig06_21.cpp(10): error: 'y': declared as a reference but not initialized
```

شکل ۲۱-٦ | مراجعه‌ای که مقداردهی اولیه نشده است،سبب تولید خطای نحوی می‌شود.

برگشت دادن یک مراجعه از یک تابع

توابع می‌توانند مراجعه‌ها را برگشت دهند، اما اینکار می‌تواند خطرناک باشد. به هنگام بازگرداندن یک مراجعه به یک متغیر اعلان شده در تابع فراخوانی شده، متغیر باید در درون آن تابع بصورت استاتیک اعلان شود. در غیر اینصورت، مراجعه به یک متغیر اتوماتیک صورت می‌گیرد که با خاتمه یافتن تابع از بین خواهد رفت، چنین متغیری، متغیر "تعریف نشده" گفته می‌شود، و رفتار برنامه غیرقابل پیش بینی می‌شود. مراجعه‌های صورت گرفته به متغیرهای تعریف نشده بعنوان dangling references شناخته می‌شوند.

۱۵-٦ آرگومان‌های پیش‌فرض

برای یک برنامه، فراخوانی مکرر یک تابع با مقدار آرگومان یکسان برای یک پارامتر خاص ،یک حالت غیر عادی شمرده نمی‌شود. در چنین مواردی، برنامه‌نویس می‌تواند مشخص کند که چنین پارامتری دارای یک "*آرگومان پیش فرض*" است، یعنی یک مقدار پیش فرض برای ارسال به آن پارامتر. زمانیکه برنامه، آرگومانی را برای پارامتری با آرگومان پیش فرض در فراخوانی تابع در نظر نمی‌گیرد، کامپایلر فراخوانی تابع را بازنویسی کرده و مقدار پیش فرض آن آرگومان را به جای آرگومانی که ارسال نشده است درج می‌کند.

باید آرگومان‌های پیش فرض، سمت راست‌ترین آرگومان در لیست پارامتری یک تابع باشند. در زمان فراخوانی یک تابع با دو یا چندین آرگومان پیش فرض، اگر آرگومان حذف شده سمت راست‌ترین آرگومان در لیست آرگومان نباشد، باید تمامی آرگومان‌های سمت راست آن آرگومان حذف شوند. آرگومان‌های پیش فرض باید در نخستین مکانی که نام تابع آورده شده مشخص شوند (عموما، در نمونه

اولیه تابع). اگر نمونه اولیه تابع به این دلیل که تعریف تابع خود بعنوان نمونه اولیه تابع مطرح است، حذف گردد، بایستی آرگومان‌های پیش فرض در سرآیند تابع مشخص شوند. مقادیر پیش فرض می‌توانند هر عبارتی از جمله ثابت‌ها، متغیرهای سراسری یا فراخوانی‌های تابع باشند. همچنین می‌توان از آرگومان‌های پیش‌فرض در کنار توابع *inline* استفاده کرد.

برنامه شکل ۲۲-۶ به توصیف نحوه استفاده از آرگومان‌های پیش فرض در محاسبه حجم یک جعبه می‌پردازد. نمونه اولیه تابع برای **boxVolume** (خط 8) مشخص می‌کند که هر سه پارامتر مقدار پیش فرض 1 را بدست آورده‌اند. دقت کنید که به منظور افزایش خوانایی، نام متغیرها را در نمونه اولیه تابع قرار داده‌ایم.

```
1  // Fig. 6.22: fig06_22.cpp
2  // Using default arguments.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // function prototype that specifies default arguments
8  int boxVolume( int length = 1, int width = 1, int height = 1 );
9
10 int main()
11 {
12    // no arguments--use default values for all dimensions
13    cout << "The default box volume is: " << boxVolume();
14
15    // specify length; default width and height
16    cout << "\n\nThe volume of a box with length 10,\n"
17       << "width 1 and height 1 is: " << boxVolume( 10 );
18
19    // specify length and width; default height
20    cout << "\n\nThe volume of a box with length 10,\n"
21       << "width 5 and height 1 is: " << boxVolume( 10, 5 );
22
23    // specify all arguments
24    cout << "\n\nThe volume of a box with length 10,\n"
25       << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
26       << endl;
27    return 0; // indicates successful termination
28 } // end main
29
30 // function boxVolume calculates the volume of a box
31 int boxVolume( int length, int width, int height )
32 {
33    return length * width * height;
34 } // end function boxVolume
```

```
The defualt box volume is: 1

The volume of a box with length 10,
withd 1 and height 1 is: 10

The volume of a box with length 10,
withd 5 and height 1 is: 50

The volume of a box with length 10,
withd 5 and height 2 is: 100
```

شکل ۲۲-٦ آرگومان‌های قراردادی در یک تابع.

اولین فراخوانی **boxVolume** (خط 13) آرگومانی را مشخص نکرده است، بنابر این از هر سه مقدار پیش فرض 1 استفاده شده است. در فراخوانی دوم (خط 17) یک آرگومان **length** ارسال شده است، بنابر این از مقادیر پیش فرض 1 برای آرگومان‌های **width** و **height** استفاده شده است. در فراخوانی سوم (خط 21) آرگومان‌هایی برای **length** و **width** ارسال شده است، بنابر این از یک مقدار پیش فرض 1 برای آرگومان **height** استفاده شده است. آخرین فراخوانی (خط 25) آرگومان‌هایی برای **length**، **width** و **height** ارسال کرده است، بنابر این از مقدار پیش فرض استفاده نشده است. دقت کنید که هر آرگومان منتقل شده به تابع صریحاً از چپ به راست به پارامترهای تابع تخصیص داده می‌شود. بنابر این، زمانیکه **boxVolume** یک آرگومان دریافت می‌کند، تابع مقدار آن آرگومان را به پارامتر **length** خود (یعنی سمت چپ‌ترین پارامتر در لیست پارامتری) تخصیص می‌دهد. زمانیکه **boxVolume** دو آرگومان دریافت می‌کند، تابع مقادیر آن آرگومان‌ها را به ترتیب به پارامترهای **length** و **width** خود تخصیص می‌دهد. سرانجام، هنگامی که **boxVolume** سه آرگومان دریافت می‌کند، تابع مقادیر آرگومان‌ها را به ترتیب به پارامترهای **length**، **width** و **height** تخصیص می‌دهد.

برنامه‌نویسی ایده‌ال

استفاده از آرگومان‌های پیش فرض می‌تواند نوشتن فراخوانی‌های تابع را آسان کند. با این وجود، برخی از برنامه‌نویسان احساس می‌کنند که مشخص کردن صریح تمامی آرگومان‌ها واضح‌تر است.

مهندسی نرم‌افزار

اگر مقادیر پیش فرض برای تابعی تغییر پیدا کنند، بایستی کد تمام سرویس‌گیرنده‌ها مجددا کامپایل شود.

خطای برنامه‌نویسی

مبادرت به استفاده از یک آرگومان پیش فرض که سمت راست‌ترین آرگومان نیست، خطای نحوی بدنبال دارد.

۱۶-۶ عملگر تفکیک قلمرو غیرباینری

امکان اعلان متغیرهای محلی و سراسری با نام مشابه وجود دارد. C++ عملگر غیرباینری یا یکانی تفکیک قلمرو (::) را برای دسترسی به یک متغیر سراسری در زمانیکه یک متغیر محلی همنام با آن در قلمرو وجود دارد، تدارک دیده است. عملگر یگانی تفکیک قلمرو نمی‌تواند برای دسترسی به یک متغیر محلی همنام موجود در یک بلوک خارجی بکار گرفته شود. اگر نام متغیر سراسری همنام با یک متغیر محلی در قلمرو نباشد، متغیر سراسری می‌توان مستقیما بدون استفاده از عملگر یگانی تفکیک قلمرو در دسترس قرار گیرد.

برنامه شکل ۲۳-۶ به توصیف عملکرد، عملگر یگانی تفکیک قلمرو با متغیرهای محلی و سراسری همنام (خطوط 7 و 11) پرداخته است. برای تاکید بر اینکه میان نسخه‌های محلی و سراسری متغیر **number**، تمایز وجود دارد، برنامه یک متغیر از نوع **int** و یکی از نوع **double** اعلان کرده است.

```
1  // Fig. 6.23: fig06_23.cpp
2  // Using the unary scope resolution operator.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int number = 7; // global variable named number
8
9  int main()
10 {
11    double number = 10.5; // local variable named number
12
13    // display values of local and global variables
14    cout << "Local double value of number = " << number
15       << "\nGlobal int value of number = " << ::number << endl;
16    return 0; // indicates successful termination
17 } // end main
```

```
Local double value of number = 10.5
Global int value of number = 7
```

شکل ۲۳-٦ | عملگر تفکیک قلمرو غیربااینری.

اجتناب از خطا

از کاربرد متغیرهای همنام برای مقاصد مختلف در یک برنامه، اجتناب کنید. اگرچه می‌توان اینکار را انجام داد، اما این امر می‌تواند خطا ساز شود.

۱۷-٦ سربارگذاری تابع

C++ امکان تعریف توابع همنام، را تا مادامیکه این توابع دارای مجموعه متفاوتی از پارامترها (حداقل در نوع پارامتر یا تعداد پارامترها یا ترتیب نوع پارامترها) باشند، تدارک دیده است. این قابلیت، سربارگذاری تابع نامیده می‌شود. زمانیکه یک تابع سربارگذاری شده فراخوانی می‌شود، کامپایلر C++ با بررسی تعداد، انواع و ترتیب آرگومان‌ها در تابع فراخوانی شده، مبادرت به انتخاب تابع مناسب می‌کند. معمولا از سربارگذاری تابع برای ایجاد چندین تابع همنام که وظایف مشابهی انجام می‌دهند، اما در نوع داده‌ها با هم اختلاف دارند، استفاده می‌شود. برای مثال، بسیاری از توابع در کتابخانه ریاضی برای نوع‌های داده عددی مختلف سربارگذاری شده‌اند.

برنامه‌نویسی ایده‌ال

سربارگذاری توابعی که وظایف مشابهی انجام می‌دهند، می‌تواند سبب افزایش خوانایی و درک برنامه‌ها شود.

سربارگذاری تابع square

برنامه شکل ۲۴-۶ از توابع سربارگذاری شده **square** به منظور محاسبه مربع یک **int** (خطوط 12-8) و مربع یک **double** (خطوط 19-15) استفاده کرده است. خط 23 نسخه **int** از تابع **square** را با ارسال

مقدار لیترال 7 فراخوانی می‌کند. C++ بطور پیش فرض با مقادیر عددی کامل لیترال بصورت نوع int رفتار می‌کند. به همین ترتیب، خط 25 نسخه **double** از تابع **square** را با ارسال مقدار لیترال 7.5 فراخوانی می‌کند، که C++ به طور پیش فرض با آن به شکل یک مقدار **double** رفتار می‌کند. در هر مورد، کامپایلر بر پایه نوع آرگومان مبادرت به انتخاب تابع مناسب برای فراخوانی می‌کند. دو خط آخر از خروجی بر این نکته تاکید می‌کنند که برای هر مورد، تابع صحیح فراخوانی شده است.

```
1  // Fig. 6.24: fig06_24.cpp
2  // Overloaded functions.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // function square for int values
8  int square( int x )
9  {
10    cout << "square of integer " << x << " is ";
11    return x * x;
12 } // end function square with int argument
13
14 // function square for double values
15 double square( double y )
16 {
17    cout << "square of double " << y << " is ";
18    return y * y;
19 } // end function square with double argument
20
21 int main()
22 {
23    cout << square( 7 ); // calls int version
24    cout << endl;
25    cout << square( 7.5 ); // calls double version
26    cout << endl;
27    return 0; // indicates successful termination
28 } // end main
```

```
square of integer 7 is 49
square of double 7.5 is 56.25
```

شکل ٢٤-٦ | توابع سربارگذاری شده square.

نحوه تفاوت قائل شدن کامپایلر مابین توابع سربارگذاری شده

توابع سربارگذاری شده توسط امضا از یکدیگر متمایز می‌شوند. امضاء ترکیبی از نام تابع و نوع پارامترهای آن است (به ترتیب). کامپایلر شناسه هر تابع را با تعداد و نوع پارامترهای آن رمزگذاری می‌کند تا یک پیوند نوع ایمن (type-safe linkage) بوجود آید. پیوند نوع ایمن ما را مطمئن می‌سازد که تابع سربارگذاری شده صحیح فراخوانی شده و نوع آرگومان‌ها با نوع پارامترها مطابقت دارد.

برنامه شکل ۲۵-۶ با کامپایلر خط فرمان Borland C++ 5.6.4 کامپایل شده است. در این برنامه بجای نشان دادن خروجی حاصل از اجرای برنامه، اسامی تغییر شکل یافته توابع، که توسط زبان اسمبلی در Borland C++ تولید شده‌اند را نشان داده‌ایم. هر نام تغییر شکل یافته با **@** شروع و بدنبال آن نام تابع آورده می‌شود. سپس نام تابع با استفاده از **$q** از لیست پارامتر تغییر شکل یافته متمایز می‌شود. در لیست

پارامتری تابع **nothing2** (خط **25**، به چهارمین خط خروجی نگاه کنید)، **c** نشان دهنده یک **char**، **i** نشان دهنده **int**، **rf** نشان دهنده یک **float &** (یعنی یک مراجعه به **float**) و **rd** نشان دهنده یک **double &** (یعنی یک مراجعه به **double**) است. در لیست پارامتری تابع **nothing1**، **i** نشان دهنده یک **int**، **f** نشان دهنده یک **float**، **c** نشان دهنده یک **char** و **ri** نشان دهنده یک **int &** می‌باشد. دو تابع **square** توسط لیست‌های پارامتری خود از یکدیگر متمایز می‌شوند، یکی **d** را برای **double** و دیگری **i** را برای **int** مشخص کرده است. نوع بازگشتی توابع در اسامی تغییر شکل یافته مشخص نشده است. توابع سربارگذاری شده می‌توانند نوع‌های بازگشتی متفاوتی داشته باشند، اما اگر چنین باشد، باید دارای لیست‌های پارامتری متفاوتی هم باشند. از طرف دیگر، نمی‌توانید دو تابع با امضاهای یکسان و نوع‌های بازگشتی متفاوت داشته باشید. دقت کنید که نام تابع تغییر شکل یافته به کامپایلر وابسته است. همچنین دقت کنید که تابع **main** تغییر شکل پیدا نمی‌کند، برای اینکه نمی‌توان آنرا سربارگذاری کرد.

```
1  // Fig. 6.25: fig06_25.cpp
2  // Name mangling.
3
4  // function square for int values
5  int square( int x )
6  {
7     return x * x;
8  } // end function square
9
10 // function square for double values
11 double square( double y )
12 {
13    return y * y;
14 } // end function square
15
16 // function that receives arguments of types
17 // int, float, char and int &
18 void nothing1( int a, float b, char c, int &d )
19 {
20    // empty function body
21 } // end function nothing1
22
23 // function that receives arguments of types
24 // char, int, float & and double &
25 int nothing2( char a, int b, float &c, double &d )
26 {
27    return 0;
28 } // end function nothing2
29
30 int main()
31 {
32    return 0; // indicates successful termination
33 } // end main
```

```
@square$qi
@square$qd
@nothing1$qifcri
@nothing2$qcirfrd
_main
```

۱۸-٦ الگوهای تابع

معمولا توابع سربارگذاری شده برای انجام عملیات‌های مشابهی که منطق متفاوت داشته و بر روی نوع‌های داده متفاوت اعمال می‌گردند، بکار گرفته می‌شوند. اگر منطق و عملیات برنامه برای هر نوع داده با یکدیگر یکسان باشند، می‌توان سربارگذاری را با استفاده از الگوهای تابع به بفرم خلاصه و راحت‌تری انجام داد. برنامه‌نویس یک تعریف منفرد از الگوی تابع را می‌نویسد. با توجه به نوع آرگومان‌های تدارک دیده شده در فراخوانی‌های این تابع، C++ بصورت اتوماتیک مبادرت به تولید الگوی تابع تخصصی مجزا شده به منظور رسیدگی مناسب به هر نوع داده از فراخوانی شده می‌کند. بنابر این، تعریف یک الگوی تابع منفرد، در اصل تعریف کردن کل خانواده توابع سربارگذاری شده است.

برنامه شکل ۲۶-۶ حاوی تعریف یک الگوی تابع (خطوط 18-4) برای تابع **maximum** است که بزرگترین مقدار از میان سه مقدار را مشخص می‌کند. تعریف تمام الگوهای تابع با کلمه کلیدی **template** شروع (خط 4) و بدنبال آن لیست پارامتری الگو در میان جفت کارکتر < > قرار داده می‌شود. هر پارامتر در لیست پارامتری الگو (غالبا *پارامتر نوع رسمی* نامیده می‌شود) همراه با کلمه‌کلیدی **typename** یا **class** آورده می‌شود. پارامترهای نوع رسمی، نقش جانگهدار برای نوع‌های بنیادین یا نوع‌های تعریف شده از سوی کاربر دارند. از این جانگهدارها برای مشخص کردن نوع پارامترهای تابع (خط 5)، نوع برگشتی تابع (خط 5) و اعلان متغیرها در درون بدنه تعریف تابع استفاده می‌شود (خط 7). تعریف یک الگوی تابع همانند سایر توابع است، اما از پارامترهای نوع رسمی بعنوان جانگهدار برای نوع‌های داده واقعی استفاده می‌کند.

```
1  // Fig. 6.26: maximum.h
2  // Definition of function template maximum.
3
4  template < class T >  // or template< typename T >
5  T maximum( T value1, T value2, T value3 )
6  {
7     T maximumValue = value1; // assume value1 is maximum
8
9     // determine whether value2 is greater than maximumValue
10    if ( value2 > maximumValue )
11       maximumValue = value2;
12
13    // determine whether value3 is greater than maximumValue
14    if ( value3 > maximumValue )
15       maximumValue = value3;
16
17    return maximumValue;
18 } // end function template maximum
```

شکل ٢٦-٦ | فایل سرآیند الگوی تابع maximum.

الگوی تابع در برنامه شکل ۲۶-۶ یک پارامتر رسمی بنام **T** (خط 4) و بعنوان یک جانگهدار برای نوع داده‌ای که توسط تابع **maximum** تست خواهد شد، اعلان کرده است. نام یک پارامتر تابع باید در لیست پارامتری الگو منحصر بفرد باشد. زمانیکه کامپایلر تشخیص می‌دهد که تابع **maximum** در کد منبع برنامه احضار شده است، نوع داده ارسالی به **maximum** جانشین **T** در سرتاسر تعریف الگو شده و C++ یک

تابع کامل برای تعیین بزرگترین مقدار از میان سه مقدار از نوع مشخص، ایجاد می‌کند. سپس تابع جدیدا ایجاد شده کامپایل می‌شود. از اینرو، الگوها ابزاری برای تولید کد هستند.

خطای برنامه‌نویسی

*قرار ندادن کلمه کلیدی **class** یا **typename** قبل از هر پارامتر نوع رسمی یک الگوی تابع (مثلا، نوشتن <**class S, class T**> بجای <**class S, T**>) خطای نحوی است.*

برنامه شکل ۲۷-۶ از الگوی تابع **maximum** (خطوط 20، 30 و 40) برای تعیین بزرگترین مقدار در میان سه مقدار صحیح، سه مقدار **double** و سه مقدار **char** استفاده کرده است.

```
1  // Fig. 6.27: fig06_27.cpp
2  // Function template maximum test program.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  #include "maximum.h" // include definition of function template maximum
9
10 int main()
11 {
12    // demonstrate maximum with int values
13    int int1, int2, int3;
14
15    cout << "Input three integer values: ";
16    cin >> int1 >> int2 >> int3;
17
18    // invoke int version of maximum
19    cout << "The maximum integer value is: "
20       << maximum( int1, int2, int3 );
21
22    // demonstrate maximum with double values
23    double double1, double2, double3;
24
25    cout << "\n\nInput three double values: ";
26    cin >> double1 >> double2 >> double3;
27
28    // invoke double version of maximum
29    cout << "The maximum double value is: "
30       << maximum( double1, double2, double3 );
31
32    // demonstrate maximum with char values
33    char char1, char2, char3;
34
35    cout << "\n\nInput three characters: ";
36    cin >> char1 >> char2 >> char3;
37
38    // invoke char version of maximum
39    cout << "The maximum character value is: "
40       << maximum( char1, char2, char3 ) << endl;
41    return 0; // indicates successful termination
42 } // end main
```

```
Input three integer values: 1 2 3
The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3

Input three characters: A C B
The maximum character value is: C
```

شکل ۲۷-٦ | عملکرد الگوی تابع maximum.

در برنامه شکل ۲۷-۶، سه تابع بعنوان نتیجه فراخوانی‌های صورت گرفته در خطوط 30 ,20 و 40 ایجاد شده است. در الگوی تابع ایجاد شده برای نوع **int** هر **T** با یک **int** بصورت زیر جایگزین می‌شود:

```
 INT maximum( INT value1, INT value2, INT value3 )
{
   INT maximumValue = value1; // assume value1 is maximum

   // determine whether value2 is greater than maximumValue
   if ( value2 > maximumValue )
      maximumValue = value2;

   // determine whether value3 is greater than maximumValue
   if ( value3 > maximumValue )
      maximumValue = value3;

   return maximumValue;
} // end function template maximum
```

۱۹-٦ بازگشتی

برنامه‌هایی که تا بدین‌جا مطرح شده‌اند عموما از توابعی تشکیل شده بودند که یکدیگر را بصورت سلسله‌مراتبی و منظم فراخوانی می‌کردند. برای حل برخی از مسائل بهتر است که توابع اقدام به فراخوانی خود نمایند. یک تابع بازگشتی می‌تواند بصورت مستقیم یا غیرمستقیم از طریق سایر توابع خود را فراخوانی کند. در این بخش و چند بخش بعدی به بررسی مسائل بازگشتی خواهیم پرداخت.

ابتدا به مفهوم بازگشت، می‌پردازیم و سپس به معرفی چند مثال که در این ارتباط هستند، خواهیم پرداخت. حل مسائل بازگشتی دارای یک سری عناصر مشترک است. توابعی که تا بدین جا مطرح کردیم، برای حل مسائل ساده بودند. در فراخوانی این نوع توابع، تابع به فراخوانی خود پایان داده و کنترل به سادگی به تابع فراخوان باز می‌گردد. یک تابع بازگشتی فراخوانی می‌شود تا مسئله‌ای را حل کند. در واقع تابع فقط از نحوه حل ساده‌ترین حالت یا حالت پایه مطلع است. اگر تابع با حالت پایه فراخوانی شود، تابع نتیجه را برگشت خواهد داد. اگر فراخوانی با مسئله بسیار پیچیده‌ای همراه باشد، تابع مسئله را به دو قسمت مفهومی تقسیم می‌کند: یک قسمت می‌داند که چه کاری می‌خواهد انجام دهد و قسمت بعدی اطلاعی از اینکه چه کاری انجام خواهد داد، ندارد. برای پیاده سازی عمل بازگشتی، قسمت دوم باید با مسئله اصلی شباهت داشته باشد، اما باید بصورت ساده‌تر یا نوع کوچکتر از مسئله اصلی را در برگیرد. بدلیل اینکه این مسئله جدید شبیه مسئله اصلی است، تابع موظف به حل مسئله کوچک و ساده خود است، و این به معنی بازگشتی بودن است و با نام گام بازگشتی هم شناخته می‌شود. گام بازگشتی می‌تواند حاوی کلمه کلیدی

return باشد چرا که نتیجه آن با بخشی از مسئله که تابع از نحوه حل آن مطلع است بکار گرفته خواهد شد. ترکیب این نتایج، سرانجام به فراخوان اصلی ارسال خواهند شد. گام بازگشتی تا زمانیکه تابع اصلی فراخوان، به صورت باز (Open) عمل می‌کند، اجرا می‌شود (اجرا خاتمه نیافته است) ممکن است گام بازگشتی برای کسب نتیجه بارها فراخوانی شود (بصورت بازگشتی) و تابع به زیر مسئله جدیدی در دو قسمت تقسیم شود. در این نوع فراخوان، مسئله در هر بار مکرراً کوچک و کوچکتر می‌شود تا به حالت پایه برسد، از اینرو عمل بازگشتی خاتمه می‌یابد. در این نقطه، تابع حالت پایه را تشخیص داده و نتیجه را برگشت می‌دهد و این فرآیند تا رسیدن به جواب نهایی صورت می‌گیرد. حال به بررسی مثالی می‌پردازیم که دارای مفهوم بازگشتی است، رابطه ریاضی بنام فاکتوریل. فاکتوریل یک عدد غیرمنفی صحیح n که بصورت !n نوشته می‌شود، عبارت است از رابطه

```
n.(n - 1).(n - 2)....1
```

که در آن !1 برابر 1 و !0 برابر 1 تعریف شده است. برای مثال، !5 که بصورت 5×4×3×2×1 نوشته می‌شود حاصلی برابر 120 دارد.

فاکتوریل یک عدد صحیح بزرگتر یا برابر صفر (برای مثال **number**) را می‌توان با روش تکرار (غیربازگشتی) با استفاده از ساختار تکرار **for** پیاده سازی کرد:

```
factorial = 1;
for ( int counter = number; counter >= 1; counter-- )
  factorial *= counter;
```

برای پیاده سازی فاکتوریل به روش بازگشتی، ابتدا باید به رابطه‌ای که در تعریف فاکتوریل وجود دارد توجه کرد:

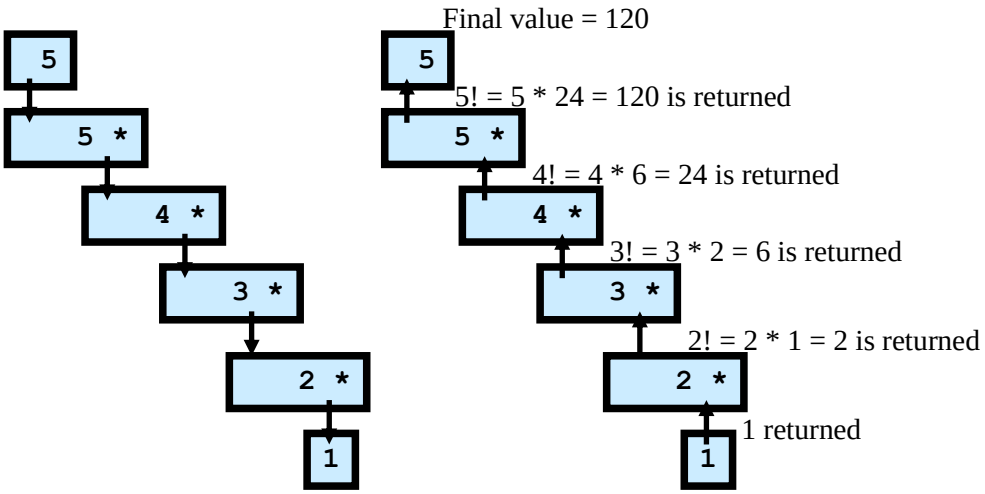
```
n! = n.(n - 1)!
```

برای مثال !5، برابر !4×5 است و می‌توان آنرا بصورت زیر نشان داد:

```
5! = 5.4.3.2.1
5! = 5.(4.3.2.1)
5! = 4.(4!)
```

ارزیابی !5 در حالت بازگشتی در شکل ۲۸-۶ به نمایش درآمده است. در بخش (a) این شکل، نحوه فراخوانی‌های بازگشتی تا رسیدن به !1 که با 1 ارزیابی می‌شود، دیده می‌شود. در بخش (b) برگشت

مقادیر حاصله از هر فراخوانی بازگشتی به فراخوان خود تا محاسبه آخرین مقدار و برگشت آن، به نمایش درآمده است. برنامه شکل ۲۹-۶ از روش بازگشتی برای محاسبه و چاپ فاکتوریل استفاده می‌کند. تابع بازگشتی **factorial** (خطوط 29-23)، ابتدا تستی را برای تعیین اینکه آیا شرط اتمام حلقه برقرار است یا خیر انجام می‌دهد (مقدار **number** کوچکتر یا برابر **1**). اگر مقدار **number** کوچکتر یا برابر **1** باشد، تابع **factorial**، مقدار **1** را برگشت می‌دهد و انجام فراخوانی‌های بازگشتی بعدی ضرورتی نخواهد داشت. اگر مقدار **number** بزرگتر از **1** باشد، عبارت **number * factorial (number - 1)** اجرا شده و تابع **factorial** بصورت بازگشتی فراخوانی می‌شود تا فاکتوریل **number - 1** محاسبه شود. توجه کنید که **factorial (number - 1)** در واقع حالت ساده شده‌ای از مسئله اصلی که محاسبه **factorial (number)** می‌باشد، است.



(a) Procession of recursive calls. (b) Values returned from each recursive call.

شکل ۲۸-٦ | محاسبه !5 به روش بازگشتی.

```
1  // Fig. 6.29: fig06_29.cpp
2  // Testing the recursive factorial function.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 unsigned long factorial( unsigned long ); // function prototype
11
12 int main()
```

```
13 {
14    // calculate the factorials of 0 through 10
15    for ( int counter = 0; counter <= 10; counter++ )
16       cout << setw( 2 ) << counter << "! = " << factorial( counter )
17          << endl;
18
19    return 0; // indicates successful termination
20 } // end main
21
22 // recursive definition of function factorial
23 unsigned long factorial( unsigned long number )
24 {
25    if ( number <= 1 ) // test for base case
26       return 1; // base cases: 0! = 1 and 1! = 1
27    else // recursion step
28       return number * factorial( number - 1 );
29 } // end function factorial
```

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

شکل ۲۹-٦ | محاسبه فاکتوریل به روش بازگشتی.

تابع **factorial** پارامتری از نوع **unsigned long** دریافت و نتیجه‌ای از نوع **unsigned long** برگشت می‌دهد. این نوع کوتاه شده نماد **unsigned long int** است.بر طبق مستندات C++ استاندارد یک متغیر از نوع **unsigned long int** بایستی حداقل در چهار بایت (**32** بیت) ذخیره شود، از اینرو می‌تواند یک مقدار از 0 تا 4294967295 در خود نگهداری کند. همانطوری که در پنجره خروجی برنامه **۲۹-۶** دیده می‌شود، مقادیر فاکتوریل بسرعت افزایش می‌یابند. بدلیل اینکه نوع داده **unsigned long** فضای زیادی در اختیار دارد، و قادر به نگهداری محاسبه اعداد بزرگتر از 7! است، آنرا انتخاب کرده‌ایم. متأسفانه، مقادیر ایجاد شده توسط تابع **factorial** با سرعتی بزرگ می‌شوند که حتی نوع **unsigned long** هم قادر به نگهداری آنها نمی‌باشد. اینحالت یکی از ضعف‌های بسیار شایع در زبان‌های برنامه‌نویسی است، چرا که نمی‌توان به آسانی در آنها نیازهای منحصر بفرد برنامه‌های مختلف همانند محاسبه مقادیر فاکتوریل اعداد بزرگ را تأمین کرد. همانطوری که شاهد خواهید بود، C++ به عنوان یک زبان گسترش یافته، به برنامه‌نویسان امکان برآوردن احتیاجات منحصر بفرد برنامه‌ها را به کمک نوع داده‌های جدید (بنام کلاس‌ها) فراهم می‌آورد.

خطای برنامه‌نویسی

فراموش کردن حالت پایه یا نوشتن گام بازگشتی که هرگز بحالت پایه نرسد، موجب انجام بازگشت‌های بی‌پایان شده و سرانجام حافظه کاملاً پر خواهد شد.

۲۰-٦ مثال بازگشتی: سری فیبوناچی

سری فیبوناچی بصورت زیر تعریف می‌شود:

0,1,1,2,3,4,8,13,21,...

که با صفر و یک آغاز می‌شود و این خصیصه را دارد که هر عدد بعدی در این سری از مجموع دو عدد قبلی حاصل می‌شود.

این سری بصورت طبیعی رخ داده و در واقع بیان کننده یک فرم حلزونی یا مارپیچی است. نسبت متوالی اعداد فیبوناچی در اطراف مقدار ثابت 1.618 قرار دارد و این عدد به صورت فطری و مداوم تکرار شده و بنام نسبت طلائی مشهور است. تعریف بازگشتی سری فیبوناچی بصورت زیر است:

```
fibonacci(0)= 0
fibonacci(1)= 1
fibonacci(n)= fibonacci(n - 1)+fibonacci(n - 2)
```

دقت کنید که در محاسبه فیبوناچی دو حالت پایه وجود دارد: fibonacci(0) که با مقدار صفر و fibonacci(1) که با مقدار 1 تعریف شده است. در برنامه شکل ۳۰-۶ محاسبه بازگشتی، عدد فیبوناچی i^{th} به کمک تابع **fibonacci** صورت گرفته است. دقت کنید که اعداد فیبوناچی، همانند مقادیر فاکتوریل بسرعت افزایش می‌یابند از اینرو از نوع داده **unsigned long** به عنوان نوع پارامتر و مقدار بازگشتی در تابع **fibonacci** استفاده شده است.

```
1   // Fig. 6.30: fig06_30.cpp
2   // Testing the recursive fibonacci function.
3   #include <iostream>
4   using std::cout;
5   using std::cin;
6   using std::endl;
7
8   unsigned long fibonacci( unsigned long ); // function prototype
9
10  int main()
11  {
12     // calculate the fibonacci values of 0 through 10
13     for ( int counter = 0; counter <= 10; counter++ )
14        cout << "fibonacci( " << counter << " ) = "
15           << fibonacci( counter ) << endl;
16
17     // display higher fibonacci values
18     cout << "fibonacci( 20 ) = " << fibonacci( 20 ) << endl;
19     cout << "fibonacci( 30 ) = " << fibonacci( 30 ) << endl;
20     cout << "fibonacci( 35 ) = " << fibonacci( 35 ) << endl;
21     return 0; // indicates successful termination
22  } // end main
23
24  // recursive method fibonacci
25  unsigned long fibonacci( unsigned long number )
26  {
27     if ( ( number == 0 ) || ( number == 1 ) ) // base cases
28        return number;
29     else // recursion step
30        return fibonacci( number - 1 ) + fibonacci( number - 2 );
```

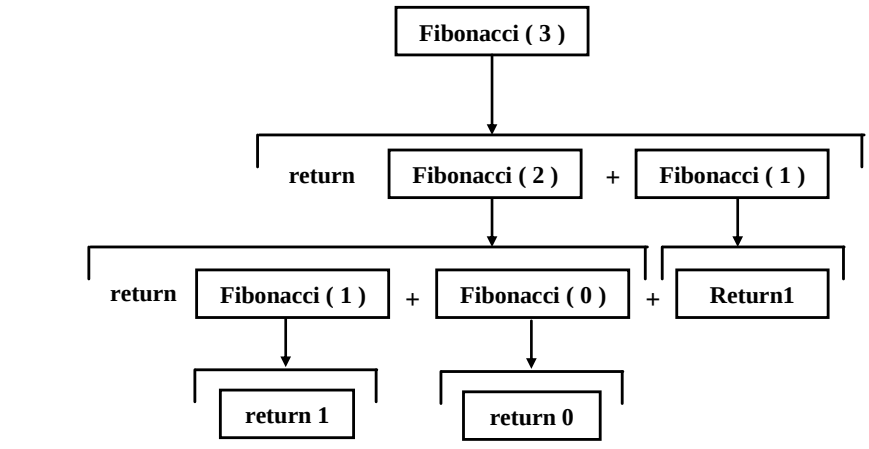
```
31  } // end function fibonacci
```

```
fibonacci(0)= 0
fibonacci(1)= 1
fibonacci(2)= 1
fibonacci(3)= 2
fibonacci(4)= 3
fibonacci(5)= 5
fibonacci(6)= 8
fibonacci(7)= 13
fibonacci(8)= 21
fibonacci(9)= 34
fibonacci(10)= 55
fibonacci(20)= 6765
fibonacci(30)= 832040
fibonacci(35)= 9227465
```

شکل ۳۰-٦ | محاسبه اعداد فیبوناچی به روش بازگشتی.

برنامه با یک عبارت **for** شروع می‌شود که مقادیر فیبوناچی مقادیر 10-0 را محاسبه کرده و بدنبال آن سه فراخوانی برای محاسبه اعداد 30 ,20 و 35 انجام می‌دهد (خطوط 20-18). فراخوانی **fibonacci** (خطوط 19 ,18 ,15 و 20) توسط تابع **main** یک فراخوانی بازگشتی نیست، اما تمام فراخوانی‌های بعدی **fibonacci** (خط30) همگی بازگشتی هستند. هر بار که **fibonacci** اجرا می‌شود، بلافاصله به تست حالت پایه می‌پردازد که به هنگام برابر بودن **number** با 0 یا 1 رخ می‌دهد (خط 27). اگر این شرط برقرار باشد، **number** برگشت داده می‌شود، چرا که (0) fibonacci برابر 0 و (1) fibonacci برابر 1 است. اما در صورتیکه **number** بزرگتر از 1 باشد، گام بازگشتی دو فراخوانی بازگشتی ایجاد می‌کند که هر یک، نوع ساده شده‌ای از مسئله اصلی است.

توجه کنید که برای محاسبه عدد فیبوناچی i^{th}، نیاز به 2^i فراخوانی خواهد بود، پس اگر فیبوناچی 20^{th} محاسبه شود، نیاز به 2^{20} فراخوانی است. شکل ۳۱-۶ نمایشی از نحوه ارزیابی **fibonacci(3)** است.



شکل ۳۱-٦ | فراخوانی‌های بازگشتی تابع fibonacci.

۲۱-٦ بازگشتی یا تکرار

در دو بخش قبلی دو تابع را دیدیم که می‌توانستند بسادگی وظایف خود را با استفاده از روش بازگشتی یا تکرار انجام دهند. در این بخش، این دو روش را باهم مقایسه کرده و در مورد اینکه چرا در برخی از مواقع یکی از این روش‌ها به روش دیگری ترجیح داده می‌شود، بحث خواهیم کرد.

هر دو روش تکرار و بازگشتی، بر مبنی ساختارهای کنترل هستند، تکرار از ساختارهای تکرار شونده همانند **for** یا **while** استفاده می‌کند، در حالیکه روش بازگشتی از ساختارهای انتخاب همانند **if**، **if…else** یا **switch** سود می‌برد. اگر چه هر دو روش مستلزم تکرار هستند، اما روش تکرار بصورت صریح از ساختارهای تکرار استفاده می‌کند در حالیکه روش بازگشتی از طریق فراخوانی‌های مکرر، عمل تکرار را انجام می‌دهد. تکرار و هم بازگشتی هر کدام مستلزم انجام یک تست خاتمه دهنده هستند. در تکرار هنگامی که شرط تکرار حلقه برقرار نباشد، تکرار خاتمه می‌یابد و در بازگشتی این اتمام به هنگام تشخیص حالت اصلی (پایه) صورت می‌گیرد.

در روش تکرار با استفاده از شمارنده-کنترل تکرار به طرف خاتمه حلقه حرکت می‌کنیم و در بازگشتی بصورت تدریجی به خاتمه نزدیک می‌شویم. در تکرار تغییرات شمارنده در نظر گرفته می‌شود تا شمارنده با یک مقدار از قبل تعیین شده، شرط تکرار حلقه را نقض کند و در بازگشتی با حفظ مسئله ساده شده از مسئله اصلی و ادامه آن تا رسیدن به حالت پایه. هم تکرار و هم بازگشتی می‌توانند بصورت نامحدود ادامه داشته باشند یک حلقه بی‌نهایت می‌تواند اگر شرط حلقه هیچ‌گاه برقرار نشود، تا بی‌نهایت تکرار شود، و در بازگشتی اگر گام بازگشتی نتواند در هر بار مسئله را ساده‌تر نماید و بحالت پایه برساند، فراخوانی بی‌نهایت بار اتفاق می‌افتد.

برای بیان تفاوت‌های موجود مابین روش تکرار و بازگشتی، اجازه دهید تا به بررسی راه‌حل تکرار برای مسئله فاکتوریل بپردازیم (شکل ۳۲-۶). دقت کنید که از یک عبارت تکرار (خطوط 29-28 از شکل ۳۲-۶) بجای عبارت انتخاب در راه‌حل بازگشتی مسئله استفاده کرده‌ایم (خطوط 27-24 از شکل ۲۹-۶). توجه کنید که در هر دو راه حل از یک تست خاتمه استفاده شده است. در روش بازگشتی، خط 24 تستی برای حالت پایه انجام می‌دهد. در روش تکرار، خط 28 تستی بر روی شرط حلقه انجام می‌دهد، اگر شرط برقرار نباشد،

حلقه خاتمه می‌پذیرد. در پایان دقت کنید که بجای تولید یک نسخه ساده از مسئله اصلی، راه حل تکرار از یک شمارنده تغییر پذیر استفاده کرده تا اینکه شرط تکرار حلقه برقرار نشود.

```
1  // Fig. 6.32: fig06_32.cpp
2  // Testing the iterative factorial method.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 unsigned long factorial( unsigned long ); // function prototype
11
12 int main()
13 {
14    // calculate the factorials of 0 through 10
15    for ( int counter = 0; counter <= 10; counter++ )
16       cout << setw( 2 ) << counter << "! = " << factorial( counter )
17          << endl;
18
19    return 0;
20 } // end main
21
22 // iterative method factorial
23 unsigned long factorial( unsigned long number )
24 {
25    unsigned long result = 1;
26
27    // iterative declaration of method factorial
28    for ( unsigned long i = number; i >= 1; i-- )
29       result *= i;
30
31    return result;
32 } // end function factorial
```

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

شکل ۳۲-٦ | راه حل تکرار برای تابع فاکتوریل.

بازگشتی دارای معایب زیادی است، مکانیزم فراخوانی‌های متعدد و در نتیجه فراخوانی‌های بسیار زیاد تابع یکی از معایب آن است و این امر می‌تواند برای زمان پردازنده و حافظه گران تمام شود. هر فراخوانی بازگشتی موجب می‌شود تا یکی کپی از تابع (همراه با داده‌ها) تهیه شود و خود این عمل در مصرف حافظه بسیار موثر است. اما در تکرار چنین اتفاقاتی رخ نمی‌دهد. پس چرا بازگشتی را انتخاب می کنیم؟

مهندسی نرم‌افزار

هر مسئله‌ای که با استفاده از روش بازگشتی حل می‌شود، می‌تواند بصورت غیربازگشتی هم حل شود.

بازگشتی یک انتخاب و نزدیک شدن به مسئله بصورت عادی است در حالیکه در تکرار به مسئله بصورت فطری یا ذاتی نزدیک شده و فهم و خطایابی برنامه بسیار آسانتر است. همچنین زمانی از روش بازگشتی استفاده می‌شود که روش تکرار برای حل مسئله مناسب نیست.

کارایی

سعی کنید از بکار بردن روش بازگشتی در حل مسائل خودداری کنید چرا که فراخوانی‌های متعدد باعث افزایش زمان و مصرف حافظه می‌شوند.

مثال ها و تمرین های بازگشت	در کتاب
تابع فاکتوریل	فصل ٦
تابع فیبوناچی	بخش۱۹-٦، شکل ۲۹-٦
مجموع دو عدد صحیح	بخش۱-٦۹، شکل ۳۰-٦
به توان رساندن دو عدد صحیح با یک عدد صحیح	خودآزمایی ۷-٦
برج های هانوی	تمرین ٤۰-٦
تصور بازگشت	تمرین ٤۲-٦
بزرگترین مقسوم علیه مشترک	تمرین ٤٤-٦
"این برنامه چه کاری انجام می دهد؟"	تمرین٤٥-٦
	تمرین۵۰-٦، تمرین۵۱-٦
"این برنامه چه کاری انجام می دهد؟"	فصل ۷
"این برنامه چه کاری انجام می دهد؟"	تمرین ۱۸-۷
مرتب سازی انتخابی	تمرین۲۱-۷
تعیین متقارن بودن یک رشته	تمرین۳۱-۷
جستجوی خطی	تمرین۳۲-۷
جسجوی دودویی	تمرین۳۳-۷
هشت وزیر	تمرین۳٤-۷
چاپ یک آرایه	تمرین۳۵-۷
چاپ یک رشته به ترتیب عکس	تمرین۳٦-۷
کوچکترین مقدار موجوددر یک آرایه	تمرین۳۷-۷
	تمرین۳۸-۷
مرتب سازی سریع	فصل ۸
پیمایش maze	تمرین۲٤-۸
تولید maze به صورت تصادفی	تمرین۲۵-۸
Maze در سایزهای مختلف	تمرین۲٦-۸
	تمرین۲۷-۸
مرتب سازی ادغامی	فصل۲۰
جستجوی خطی	بخش ۳-۳-۲۰، شکل ۵-۲۰ تا ۷-۲۰

جستجوی دودویی **مرتب سازی سریع**	**تمرین۸-۲۰** **تمرین۹-۲۰** **تمرین۱۰-۲۰**
افزودن عضو به درخت دودویی **پیمایش پیش ترتیب یک درخت دودویی** **پیمایش میان ترتیب یک درخت دودویی** **پیمایش پس ترتیب یک درخت دودویی** **چاپ یک لیست پیوندی به ترتیب بر عکس** **جستجوی یک لیست پیوندی** **حذف درخت دودویی** **چاپ درخت**	**فصل ۲۱** **بخش ۷-۲۱-شکل ۲۰-۲۱ تا ۲۲-۲۱** **بخش ۷-۲۱-شکل ۲۰-۲۱ تا ۲۲-۲۱** **بخش ۷-۲۱-شکل ۲۰-۲۱ تا ۲۲-۲۱** **بخش ۷-۲۱ شکل ۲۰-۲۱ تا ۲۲-۲۱** **تمرین۲۰-۲۱** **تمرین۲۱-۲۱** **تمرین۲۲-۲۱** **تمرین۲۵-۲۱**

شکل ۳۳-٦ | مثال‌های بازگشتی و تمرینات مرتبط با آن.

۲۲-۶ مبحث آموزشی مهندسی نرم‌افزار: شناسایی عملیات‌های کلاس در سیستم ATM

در بخش‌های "مبحث آموزشی مهندسی نرم‌افزار" در انتهای فصل‌های سوم، چهارم و پنجم، قدم‌های اولیه در طراحی شی‌گرا سیستم ATM را برداشتیم. در این بخش، به تعیین برخی از عملیات‌های کلاس (یا رفتارها) مورد نیاز در پیاده‌سازی سیستم ATM می‌پردازیم.

شناسایی عملیات‌ها

یک عملیات، سرویسی است که شی‌های یک کلاس به سرویس‌گیرنده‌های کلاس عرضه می‌کنند. به عملیات برخی از شی‌ها در دنیای واقعی توجه کنید. عملیات یک رادیو شامل تنظیم ایستگاه و صدای آن است (معمولا توسط شخصی که کنترل‌های رادیو را تنظیم می‌کند، انجام می‌شوند). عملیات یک اتومبیل شامل شتاب‌گیری (توسط راننده و با فشار دادن پدال گاز)، کاهش شتاب (توسط راننده و با فشار دادن پدال ترمز یا رها کردن پدال گاز)، چرخش و تعویض دنده‌ها است. شی‌ها نرم‌افزاری هم عملیات‌های گوناگونی انجام می‌دهند، برای مثال، یک نرم‌افزار گرافیکی می‌تواند عملیات‌های برای ترسیم دایره، خط، مربع و کارهای دیگر انجام دهد. نرم‌افزار صفحه‌گسترده می‌تواند عملیاتی مانند چاپ صفحه‌گسترده، جمع عناصر در سطر و ستون و چاپ نمودار بصورت میله‌ای یا دایره‌ای انجام دهد.

می‌توانیم با بررسی فعل‌های کلیدی و عبارات فعلی در مستند نیازها، تعدادی از عملیات هر کلاس را استنتاج کنیم. سپس هر یک از آنها را به کلاس‌های خاصی در سیستم خود مرتبط می‌کنیم (شکل ۳۴-۶). عبارات یا جمله‌های فعلی به نمایش در آمده در جدول شکل ۳۴-۶ در تعیین عملیات‌های هر کلاس به ما کمک می‌کنند.

مدلسازی عملیات‌ها

برای شناسایی عملیات‌ها، به بررسی عبارات فعلی لیست شده برای هر کلاس در جدول شکل ۳۴-۶ می‌پردازیم. عبارت "اجرای تراکنش مالی" مرتبط با کلاس **ATM** بطور ضمنی نشان می‌دهد که کلاس **ATM** به تراکنش دستور اجرا شدن صادر می‌کند. بنابر این، کلاس‌های **BalanceInquiry**، **Withdrawal** و **Deposit** هر کدام به یک عملیات برای تدارک دیدن این سرویس برای ATM نیاز دارند. ما این عملیات (که آن را **execute** نام داده‌ایم) را در قسمت سوم از کلاس‌های سه‌گانه تراکنشی در دیاگرام به روز شده شکل ۳۵-۶ قرار داده‌ایم. در مدت زمان یک جلسه ATM، شی **ATM** عملیات **execute** را برای هر شی تراکنشی فراخوانی می‌کند تا به اجرا در آیند.

زبان UML عملیات‌ها را (که بعنوان توابع عضو در C++ پیاده‌سازی می‌شوند) با لیست کردن نام عملیات، و بدنبال آن لیست پارامتری جدا شده با کاما در درون پرانتزها، یک کولن و نوع بازگشتی عرضه می‌کند:

نوع بازگشتی : (پارامتر n ،... ،پارامتر ۲ ،پارامتر ۱) نام عملیات

هر پارامتر در این لیست متشکل از یک نام پارامتر، بدنبال آن یک کولن و نوع پارامتر است:

نوع پارامتر : نام پارامتر

در این بخش مبادرت به لیست کردن پارامترهای عملیات خود نکرده‌ایم، بزودی به شناسایی و مدل کردن پارامترهای برخی از عملیات‌ها اقدام خواهیم کرد. برای برخی از عملیات‌ها، هنوز اطلاعی از نوع برگشتی آنها نداریم، از اینرو در دیاگرام خبری از آنها نیست. در این مرحله از طراحی عدم حضور آنها کاملا عادی است. همانطوری که فرآیند پیاده‌سازی به جلو می‌رود، نوع‌های بازگشتی باقیمانده را به دیاگرام اضافه خواهیم کرد.

کلاس	افعال و عبارات فعلی
`ATM`	اجرای تراکنش مالی
`BalanceInquiry`	{در مستند نیازها وجود ندارد}
`WithDrawal`	{در مستند نیازها وجود ندارد}
`Deposit`	{در مستند نیازها وجود ندارد}
`BankDatabase`	احراز هویت کاربر، بازیابی موجودی حساب، میزان سپرده‌گذاری در حساب، بدهکار کردن حساب به میزان برداشت پول
`Account`	بازیابی موجودی حساب، میزان سپرده‌گذاری در حساب، بدهکار کردن حساب به میزان برداشت پول
`Screen`	نمایش پیغام به کاربر
`Keypad`	دریافت ورودی عددی از کاربر
`CashDispenser`	پرداخت پول، تعیین اینکه به میزان کافی پول نقد برای پاسخ به تقاضا وجود دارد یا خیر

DepositSlot	دریافت پاکت سپرده

شکل ٣٤-٦ | افعال و عبارات فعلی برای هر کلاس در سیستم ATM.

شکل ٣٥-٦ | کلاس‌ها در سیستم ATM به همراه صفات و عملیات‌ها.

عملیات کلاس‌های BankDatabase و Account

در جدول شکل ۳۴-۶ جمله "احراز هویت کاربر" در کنار کلاس **BankDatabase** قرار داده شده است، پایگاه داده شی است حاوی اطلاعات حساب مورد نیاز برای تعیین اینکه آیا شماره حساب و **PIN** وارد شده از طرف کاربر مطابق با شماره حساب و **PIN** نگه‌داری شده در بانک است یا خیر. بنابر این، کلاس **BankDatabase** به عملیاتی نیاز دارد که سرویس احراز هویت را برای **ATM** تدارک ببیند. عملیات **authenticateUser** را در قسمت سوم از کلاس **BankDatabase** جای داده‌ایم (شکل ۳۵-۶). با این وجود، یک شی از کلاس **Account**، و نه کلاس **BankDatabase**، مبادرت به ذخیره شماره حساب و **PIN** می‌کند که بایستی برای احراز هویت کاربر در دسترس باشد، از اینرو کلاس **Accoount** باید سرویسی برای اعتبارسنجی **PIN** وارد شده از سوی کاربر با **PIN** ذخیره شده در یک شی **Account** تدارک دیده باشد. بنابر این، عملیات **validatePIN** را برای کلاس **Account** در نظر گرفته‌ایم. توجه کنید که نوع برگشتی **Boolean** را برای عملیات‌های **authenticateUser** و **validatePIN** انتخاب کرده‌ایم. هر عملیاتی یک مقدار برگشت می‌دهد و این مقدار دلالت بر این دارد که آیا عملیات در انجام وظیفه خود موفق بوده (یعنی برگشت مقدار **true**)یا خیر (یعنی برگشت مقدار **false**).

در جدول شکل ۳۴-۶ چندین عبارت فعلی برای کلاس **BankDatabase** لیست شده است: "بازیابی موجودی حساب"، "میزان سپرده‌گذاری در حساب"، "بدهکار کردن حساب به میزان برداشت پول". همانند "احراز هویت کاربر" این عبارات فعلی اشاره به سرویس‌های دارند که بایستی پایگاه داده برای **ATM** تدارک ببیند، چرا که پایگاه داده تمام اطلاعات حساب را که اعتبارسنجی کاربر و تراکنش‌های **ATM** نقش دارند در خود ذخیره کرده است. با این وجود، در واقع شی‌های از کلاس **Account** عملیات‌های که این عبارات فعلی به آنها اشاره دارند، را انجام می‌دهند. از اینرو، یک عملیات به هر دو کلاس **BankDatabase** و **Account** تخصیص داده‌ایم تا متناظر با هر یک از این عبارات باشند. از بخش ۱۱-۳ به یاد دارید که چون، یک حساب بانکی حاوی اطلاعات حساس است، به **ATM** اجازه دسترسی مستقیم به حساب را ندادیم. پایگاه داده بصورت یک میانجی یا واسط مابین **ATM** و داده‌های حساب عمل می‌کند، بنابر این جلوی دسترسی غیرمجاز گرفته می‌شود. همانطوری که در بخش ۱۲-۷ مشاهده خواهید کرد، کلاس **ATM** عملیات‌های کلاس **BankDatabase** را فراخوانی می‌کند، که هر یک از آنها در ادامه عملیاتی همنام را در کلاس **Account** فراخونی می‌کنند.

جمله "بازیابی موجودی حساب" نشان می‌دهد که کلاس‌های **BankDatabase** و **Account** به عملیات **getBalance** نیاز دارند. با این همه، بخاطر دارید که دو صفت برای کلاس **Account** به منظور نمایش موجودی ایجاد کردیم، **availableBalance** و **totalBalance**. پرس‌وجوی یک موجودی مستلزم دسترسی به هر دو صفت موجودی است، از اینرو است که می‌تواند آنها را به کاربر نشان دهد، اما برداشت پول فقط نیاز به بررسی مقدار **availableBalance** دارد. برای اجازه دادن به شی‌ها در سیستم برای بدست آوردن هر صفت موجودی بصورت مجزا از هم، مبادرت به افزودن عملیات‌های **getAvailableBalance** و **getTotalBalance** به بخش سوم از کلاس‌های **BankDatabase** و **Account** کرده‌ایم (شکل ۳۵-۶). نوع برگشتی را از نوع **Double** برای هر یک از این عملیات‌ها در نظر گرفته‌ایم، چرا که صفات موجودی که آنها بازیابی می‌کنند از نوع **Double** هستند.

جمله " میزان سپرده‌گذاری در حساب" و "بدهکار کردن حساب به میزان برداشت پول" نشان می‌دهند که کلاس‌های **BankDatabase** و **Account** باید عملیاتی برای به روز کردن یک حساب در جریان سپرده‌گذاری و برداشت پول انجام دهند. از اینرو، مبادرت به تخصیص عملیات‌های **credit** و **debit** به کلاس‌های **BankDatabase** و **Account** کرده‌ایم. بخاطر دارید که دادن اعتبار به حساب فقط مقداری را به صفت **totalBalance** اضافه می‌کند. از طرف دیگر، بدهکار کردن یک حساب (در نتیجه برداشت پول) از میزان هر دو صفت موجودی کم می‌کند. این جزئیات پیاده‌سازی را در درون کلاس **Account** پنهان کرده‌ایم. اینحالت مثال خوبی از کپسوله‌سازی و پنهان‌سازی اطلاعات است.

عملیات کلاس Screen

کلاس **Screen** در زمان‌های مختلف در یک جلسه **ATM** "پیغام‌های را به کاربر نشان می‌دهد". تمامی خروجی‌های بصری از طریق صفحه‌نمایش **ATM** رخ می‌دهند. در مستند نیازها انواع مختلفی از پیغام‌ها آورده شده است (همانند، پیغام خوش‌آمدگویی، پیغام خطا، پیغام تشکر) که صفحه‌نمایش برای کاربر بنمایش در می‌آورد. همچنین مستند نیازها نشان می‌دهد که صفحه‌نمایش اعلان‌ها و منوهای را به کاربر نشان می‌دهد. با این همه، اعلان در واقع یک پیغام توضیحی است که به کاربر آنچه را که باید انجام دهد، دیکته می‌کند و منو اصولا نوعی اعلان از چندین پیغام (گزینه‌های منو) است. بنابر این، بجای تخصیص اختصاصی کلاس **Screen** به هر عملیات برای نمایش هر نوع پیغام، اعلان و منو، فقط یک عملیات ایجاد می‌کنیم که می‌تواند هر پیغام مشخص شده توسط پارامتر را به نمایش در آورد. این عملیات را که **displayMessage** نام دارد، در قسمت سوم از کلاس **Screen** دیاگرام کلاس خود جای داده‌ایم (شکل ۳۵-۶). توجه کنید که فعلا نگران پارامترهای این عملیات نیستیم، در انتهای این بخش مبادرت به مدل کردن پارامتر خواهیم کرد.

عملیات کلاس Keypad

از جمله "دریافت ورودی عددی از کاربر" در جدول شکل ۳۴-۶ چنین برداشت می‌کنیم که بایستی کلاس **Keypad** عملیات **getInput** را انجام دهد. چون صفحه کلید **ATM**، برخلاف صفحه کلید کامپیوتر، فقط حاوی اعداد **0-9** است، تعیین کرده‌ایم که این عملیات یک مقدار صحیح برگشت دهد. از مستند نیازها بخاطر دارید که در شرایط مختلف، امکان دارد کاربر ارقام متفاوتی وارد سازد (مثلا، شماره حساب، **PIN**، شماره گزینه منو، میزان سپرده‌گذاری). کلاس **Keypad** بسادگی مقدار عددی را برای سرویس‌گیرنده کلاس بدست می‌آورد، این کلاس مسئول تست مقدار وارد شده تحت ضوابط خاص نیست. هر کلاسی که از این عملیات استفاده می‌کند باید به اعتبارسنجی مقدار وارد شده از سوی کاربر پرداخته و در صورت اشتباه بودن، پیغام مناسب خطا را از طریق کلاس **Screen** بنمایش در آورد.

عملیات کلاس‌های CashDispenser و DepositSlot

در جدول شکل ۳۴-۶ عبارت "پرداخت پول" برای کلاس **CashDispenser** لیست شده است. بنابر این، عملیات **dispenseCash** را ایجاد و آنرا تحت کلاس **CashDispenser** در شکل ۳۵-۶ لیست کرده‌ایم. همچنین کلاس **CashDispenser** حاوی عبارت "تعیین اینکه به میزان کافی پول نقد برای پاسخ به تقاضا وجود دارد یا خیر" است. از اینرو، **isSufficientCashAvailable** را بعنوان عملیاتی که مقداری از نوع **Boolean** در کلاس **CashDispenser** برگشت می‌دهد، در نظر گرفته‌ایم. همچنین در جدول شکل ۳۴-۶ عبارت "دریافت پاکت سپرده‌گذاری" برای کلاس **DepositSlot** لیست شده است. بایستی شکاف سپرده‌گذاری تعیین کند که آیا پاکتی دریافت کرده است یا خیر، از اینرو عملیات **isEnvelopeReceived** را در نظر گرفته‌ایم، که مقداری از نوع **Boolean** در بخش سوم کلاس **DepositSlot** برگشت می‌دهد.

عملیات کلاس ATM

در این لحظه عملیاتی برای کلاس **ATM** لیست نشده است. هنوز از سرویس‌های که کلاس **ATM** برای سایر کلاس‌ها در سیستم تدارک دیده است، اطلاعی نداریم. زمانیکه، سیستم را با کد C++ پیاده‌سازی می‌کنیم، عملیات‌های این کلاس به همراه چندین عملیات دیگر از سایر کلاس‌ها به سیستم خواهند پیوست.

شناسایی و مدل کردن پارامترهای عملیاتی

تا بدین مرحله، توجهی به پارامترها در عملیات‌های خود نداشتیم. حال اجازه دهید از نزدیک نگاهی به پارامترهای عملیاتی بیندازیم. یک پارامتر عملیاتی را با بررسی داده مورد نیاز عملیات برای انجام وظیفه در نظر گرفته شده، شناسایی می‌کنیم.

به عملیات **authenticateUser** در کلاس **BankDatabase** توجه کنید. برای احراز هویت یک کاربر، بایستی این عملیات از شماره حساب و **PIN** تدارک دیده شده از سوی کاربر مطلع باشد. از اینرو، مشخص کرده‌ایم که عملیات **authenticateUser** پارامترهای صحیح **userAccountNumber** و **userPIN** را دریافت کند، که عملیات باید به مقایسه شماره حساب و **PIN** از شی **Account** در پایگاه داده بپردازد. در ابتدای نام این پارامترها پیشوند "user" را قرار داده‌ایم تا از اشتباه شدن مابین اسامی پارامتر عملیات و اسامی صفت که متعلق به کلاس **Account** هستند، اجتناب شود. در شکل ۳۶-۶ این پارامترها را در دیاگرام کلاس لیست کرده‌ایم، که فقط مدل کننده کلاس **BankDatabase** است.

شکل ٣٦-٦ | کلاس BankDatabase با پارامترهای عملیاتی.

بخاطر دارید که UML هر پارامتر را در یک لیست پارامتری مجزا شده با کاما که متعلق به یک عملیات است، با لیست کردن نام پارامتر، بدنبال آن یک کولن و نوع پارامتر مدل‌سازی می‌کند. از اینرو، در شکل ۳۶-۶ مشخص است که عملیات **authenticateUser** دو پارامتر دریافت می‌کند: **userAccountNumber** و **userPIN** که هر دو از نوع **Integer** هستند. زمانیکه سیستم را در C++ پیاده‌سازی کردیم، این پارامترها با مقادیر **int** عرضه خواهند شد.

عملیات‌های **getAvailableBalance, getTotalBalance, credit** و **debit** از کلاس **BankDataBase** نیز نیازمند پارامتر **userAccountNumber** برای شناسایی حسابی هستند که باید پایگاه داده بر روی آن عملیات را اجرا کند، از اینرو این پارامترها را در دیاگرام کلاس شکل ۳۶-۶ وارد کرده‌ایم. علاوه بر این، عملیات‌های **credit** و **debit** هر یک نیازمند پارامتر **amount** از نوع **Double** هستند تا تعیین کننده پول به اعتبار گذاشته شده یا بدهی باشد.

دیاگرام کلاس در شکل ۳۷-۶ مدل کننده پارامترهای عملیاتی کلاس **Account** است. عملیات **validatePIN** فقط نیازمند یک پارامتر بنام **userPIN** است، که حاوی **PIN** وارد شده از سوی کاربر برای مقایسه شدن با **PIN** متناظر در حساب است. همانند همکارهای خود در کلاس **BankDatabase**، عملیات‌های **credit** و **debit** در کلاس **Account** هر یک نیازمند پارامتر **amount** از نوع **Double** هستند.

شکل ٣٧-٦ | کلاس Account با پارامترهای عملیاتی.

عملیات‌های **getAvailableBalance** و **getTotalBalance** در کلاس **Account** نیازی به داده اضافی برای انجام وظایف خود ندارند.

شکل ۳۸-۶ کلاس **Screen** را با پارامتر مشخص شده برای عملیات **displayMessage** مدل کرده است. این عملیات فقط مستلزم یک پارامتر بنام **message** از نوع **String** است که دلالت بر پیغام بنمایش در آمده دارد.

دیاگرام کلاس در شکل ۳۹-۶ مشخص می‌کند که عملیات **dispenseCash** از کلاس **CashDispenser** یک پارامتر بنام **amount** از نوع **Double** دریافت می‌کند که نشاندهنده پول نقد است. همچنین عملیات **isSufficientCashAvailable** پارامتر **amount** را که از نوع **Double** است دریافت می‌کند، که نشاندهنده مقدار پول در درخواست است.

توجه کنید که بحثی در ارتباط با پارامترهای عملیاتی **execute** کلاس‌های **BalanceInquiry,** **Withdrawal** و **Deposit**، عملیات **getInput** از کلاس **Keypad** و عملیات **isEnvelopeReceived** از کلاس **DepositSlot** به میان نیاوردیم. تا بدین مرحله از فرآیند طراحی، نمی‌توانیم تعیین کنیم که آیا این عملیات‌ها نیازمند داده‌های اضافی برای انجام وظایف خود هستند یا خیر، بنابر این لیست پارامتری آنها را خالی نگه داشتیم. همانطوری که به پیش می‌رویم، در مورد افزودن پارامترها به عملیات‌ها تصمیم می‌گیریم.

شکل ۳۸-٦ | کلاس Screen با پارامترهای عملیاتی.

شکل ۳۹-٦ | کلاس CashDispenser با پارامترهای عملیاتی.

تمرینات خودآزمایی مبحث مهندسی نرم‌افزار

٦-١١ کدام یک از موارد زیر یک رفتار نیست؟.

a) خواندن داده از یک فایل

b) چاپ خروجی

c) خروجی متنی

d) بدست آوردن ورودی از کاربر

٦-٢ اگر بخواهید عملیاتی به سیستم ATM اضافه کنید که صفت amount از کلاس Withdrawal را برگشت دهد، چگونه و در کجای دیاگرام شکل ۳۵-۶ اینکار را انجام می‌دهید.

٦-٣ مفهوم عملیات لیست شده در زیر را که ممکن است در دیاگرام کلاسی بکار رفته باشد، چیست؟

```
add( x : Integer, y : Integer ) : Integer
```

پاسخ خودآزمایی مبحث آموزشی مهندسی نرم‌افزار

٦-١ c

٦-٢ می‌توان این عملیات را در قسمت سوم کلاس **Withdrawal** جای داد:

```
getAmount() : Double
```

۶-۳ نام این عملیات add بوده و پارامترهای صحیح x و y را دریافت کرده و یک مقدار صحیح برگشت می‌دهد.

خودآزمایی

۶-۱ جاهای خالی را در عبارات زیر با کلمات مناسب پر کنید.

a) کامپونت‌های برنامه در C++، ..... و ..... نامیده می‌شوند.

b) یک تابع با ..... آن فعال می‌شود.

c) متغیری که فقط در درون تابع اعلان شده شناخته شود، متغیری از نوع ..... نامیده می‌شود.

d) عبارت ..... در تابع فراخوانی شده، موجب ارسال مقداری به تابع فراخواننده می‌شود.

e) تابعی که با کلمه کلیدی ..... تعریف شده باشد، مقدار باز نمی‌گرداند.

f) ..... یک شناسه بخشی از برنامه است که در آن بخش شناسه قابل استفاده می‌باشد.

g) به سه روش می‌توان کنترل را از یک تابع فراخوانی شده به فراخوان بازگرداند، این سه روش عبارتنداز .....، ..... و ..... .

h) یک ..... به کامپایلر اجازه بررسی تعداد، نوع‌ها و ترتیب آرگومان‌های ارسالی به تابع را می‌دهد.

i) تابع ..... اعداد تصادفی ایجاد می‌کند.

j) تابع ..... برای تنظیم عدد تصادفی و تغذیه آن بکار گرفته می‌شود.

k) تصریح کننده‌های کلاس ذخیره‌سازی عبارتند از mutable، .....، .....، ..... و .....

l) متغیرهای اعلان شده در یک بلوک یا لیست پارامترهای تابع دارای کلاس ذخیره‌سازی ..... هستند.

m) تصریح کننده کلاس ذخیره‌سازی ..... به کامپایلر توصیه می‌کند که متغیر را در ثبات کامپیوتر ذخیره کند.

n) متغیر اعلان شده در خارج از هر بلوک یا تابع، متغیر ..... نامیده می‌شود.

o) باید متغیر محلی در تابعی را که مقدار خود را مابین فراخوانی تابع حفظ می‌کند، بصورت ..... در کلاس ذخیره‌سازی اعلان شود.

p) شش قلمرو ممکنه برای یک شناسه عبارتند از .....، .....، .....، .....، ..... و ..... .

q) تابعی که خود را بصورت مستقیم یا غیرمستقیم فراخوانی می‌کند یک تابع ..... است.

r) یک تابع بازگشتی عموماً متشکل از دو کامپونت است: بخشی که منظور از آن اتمام رفتار بازگشتی با تست حالت ..... است و بخشی که مسئله را به فرم بازگشتی مطرح و فراخوانی می‌کند.

s) در C++، امکان داشتن توابع مضاعف با یک نام وجود دارد که در آنها نوع یا تعداد آرگومان متفاوت هستند. این توابع ..... نامیده می‌شوند.

۶-۲ با توجه به برنامه شکل ۴۰-۶، قلمرو هر یک از عناصر زیر را تعیین کنید:

a) متغییر x در main.

b) متغیر y در cube.

c) تابع cube.

d) تابع main.

e) نمونه اولیه تابع برای cube.

f) شناسه y در نمونه اولیه تابع برای cube.

۶-۳ در برنامه زیر، قلمرو هر کدام یک از عناصر زیر را تعیین کنید:

a) متغیر **x**

b) متغیر **y**

c) تابع **cube**

d) تابع **paint**

e) متغیر **yPos**

۶-۴ برای هر کدامیک از موارد زیر یک سرآیند تابع ایجاد کنید:

a) تابع **hypotenuse** که دو آرگومان **side1** و **side2** از نوع **double** دریافت و نتیجه‌ای از نوع **double** برگشت دهد.

b) تابع **smallest** که سه آرگومان **x**، **y** و **z** دریافت و یک مقدار صحیح برگشت دهد.

c) تابع **instructions**، که هیچ آرگومانی دریافت نکرده و هیچ مقداری بر نمی‌گرداند.

d) تابع **intToSingle**، که یک آرگومان از نوع صحیح بنام **number** دریافت و مقداری از نوع **float** برگشت دهد.

پاسخ خودآزمایی

۶-۱ a) کلاس و تابع. b) فراخوانی. c) محلی. d) **return**. e) **void**. f) قلمرو. g) **;return**، **return** *عبارت* و **{**. h) **Random.Next** i) اتوماتیک. j) بازگشتی. k) پایه (اصلی). l) Overload. m) بلوک. n) تکرار. o) انتخاب. p) تابع. q) مشابه.

۶-۲ a) اشتباه. تابع **Abs** از کلاس **Math** مقدار مطلق یک عدد را باز می‌گرداند. b) صحیح. c) صحیح. d) صحیح. e) اشتباه. نوع **Char** می‌تواند به نوع **int** با تبدیل کاهشی، برگردانده شود. f) اشتباه. تابعی که بصورت بازگشتی خود را فراخوانی می‌کند با نام فراخوانی بازگشتی یا گام بازگشتی شناخته

می‌شود. g) صحیح. h) اشتباه. بازگشتی بی‌پایان زمانی رخ می‌دهد که تابع بازگشتی هرگز به حالت پایه نرسد. i) صحیح. j) صحیح.

۶-۳ a) قلمرو کلاس. b) قلمرو بلوک. c) قلمرو کلاس. d) قلمرو کلاس. e) قلمرو بلوک.

۶-۴

(a

```
double hypotenuse (double side1, double side2)
```

(b

```
int smallest (int x,
    int y, int z )
```

(c

```
void instructions( )
```

(d

```
float intToFloat (int number)
```

۶-۵

a) خطا: تابع **h** در تابع **g** تعریف شده است.

اصلاح: تعریف تابع **h** به خارج از تعریف تابع **g** منتقل شود.

b) خطا: فرض تابع بر برگشت یک مقدار **int** است، اما چنین نیست.

اصلاح: حذف عبارت `result = x + y` و جایگزین کردن عبارت زیر:

```
return x + y;
```

یا افزودن عبارت زیر به انتهای بدنه تابع:

```
return result;
```

c) خطا: نتیجه `n + sum(n-1)` توسط این تابع برگشت داده نمی‌شود.

اصلاح: عبارت موجود در شرط else بصورت زیر نوشته شود، `return n + sum(n-1);`

d) خطا: قرار دادن سیمکولن پس از پرانتز سمت راست و تعریف مجدد پارامتر a در تعریف تابع هر دو اشتباه است.

اصلاح: حذف سیمکولن و حذف اعلان `float a;`

e) خطا: تابع مقداری باز می‌گرداند که در نظر گرفته نشده است.

اصلاح: تغییر نوع برگشتی به **int**

تمرینات

۶-۶ در پارکینگی هزینه نگهداری هر اتومبیل تا سه ساعت حداقل ۶ دلار است. هزینه هر ساعت اضافی، ۵/۱ دلار علاوه بر هزینه سه ساعت می‌باشد. حداکثر هزینه نگهداری در ۲۴ ساعت معادل ۲۵ دلار است. فرض کنید که اتومبیل‌ها فقط می‌توانند تا ۲۴ ساعت در پارکینگ نگهداری شوند. برنامه‌ای بنویسید که هزینه نگهداری اتومبیل هر مشتری را محاسبه و به نمایش درآورد. باید ساعت ورود هر اتومبیل ثبت شود. برنامه باید از تابعی بنام **CalculateCharges** برای محاسبه هزینه هر ماشین استفاده کند.

۶-۷ کامپیوتر در امر آموزش نقش مهمی برعهده دارد. برنامه‌ای بنویسید که به دانش‌آموزان مدرسه ابتدائی، جدول ضرب آموزش دهد. از تابع **Next** برای ایجاد دو عدد مثبت یک رقمی استفاده کنید. سپس سئوالی مانند

```
How much is 6 times 7?
```

پرسیده شود. برنامه پاسخ وارد شده را چک کرده و در صورت صحیح بودن، پیغام "Very good" را چاپ کرده و سئوال دیگری مطرح کند. در غیر اینصورت پیغام "No.Please try again" چاپ شده و همان سئوال تا دریافت پاسخ صحیح تکرار شود.

۶-۷ تابعی بنویسید که یک عدد صحیح دریافت کرده و آنرا معکوس کند. برای مثال اگر عدد دریافتی 8456 باشد، خروجی تابع عدد 6548 را به نمایش درآورد.

۶-۸ به عددی، عدد اول گفته می‌شود که فقط به یک و خودش قابل تقسیم باشد برای مثال اعداد 2، 3 و 5 عدد اول هستند، اما اعداد 9، 8، 6 و 4 عدد اول نیستند.

a) تابعی بنویسید تا مشخص کند آیا عددی اول است یا خیر.

b) با استفاده از این تابع در برنامه، اعداد اول قرار گرفته از 1000 تا 1 را مشخص کرده و به نمایش درآورید.