فصل شانزدهم

رسیدگی به استثناء

اهداف

- استثناء چیست و در چه مواقعی از آنها استفاده می‌کنیم.
- استفاده از try، catch و throw برای تشخیص، رسیدگی و نمایان ساختن استثناها.
- پردازش استثناهای غیرمنتظره و گرفتار نشده.
- اعلان کلاس‌های استثناء.
- چگونه باز کردن پشته می‌تواند استثناهای گرفتار نشده در یک قلمرو را در قلمرو دیگری گرفتار سازد.
- رسیدگی به واماندگی new.
- استفاده از auto_ptr برای اجتناب از فقدان حافظه.
- آشنایی با سلسله مراتب استاندارد استثناء.

رئوس مطالب

۱۶-۱ مقدمه

در این فصل، در مورد رسیدگی به استثناها بحث می‌کنیم. یک استثناء دلالت بر وجود مشکلی دارد که در زمان اجرای برنامه رخ می‌دهد. نام "استثناء" از این حقیقت بدست آمده که، اگر چه مشکل می‌تواند همیشه رخ دهد، اما استثناء بندرت رخ می‌دهد. اگر "قاعده‌ای" بر این اصل استوار است که عبارتی بطرز صحیح اجرا شود، اما رویدادی موجب رخ دادن مشکلی گردد پس "استثنای در این قاعده" وجود دارد.

رسیدگی به استثناء به برنامه‌نویسان امکان می‌دهد تا برنامه‌هایی ایجاد کنند که قادر به برطرف کردن (یا رسیدگی) استثناءها هستند. در بسیاری از موارد، رسیدگی به یک استثناء به برنامه امکان می‌دهد در صورت عدم برخورد با مشکلی به اجرای خود ادامه دهد. با این همه، مشکلات جدی واساسی می‌توانند مانع از اجرای عادی برنامه شوند در چنین حالاتی برنامه باید مشکل را به کاربر اطلاع داده و با یک روش کنترل شده به مشکل خاتمه دهد. ویژگی‌های مطرح شده در این فصل برنامه‌نویسان را قادر به نوشتن برنامه‌های واضح، پایدار و مقاوم در برابر خطا می‌کنند.

قالب و جزئیات روش رسیدگی به خطا در C++ مبتنی بر تحقیقات Andrew koenig و Bjarne Stroustrup در مقاله‌ای بنام "Exception Handling for C++ (revised)" است.

این فصل با معرفی مفاهیم رسیدگی به استثناء و توصیف تکنیک‌های پایه در این زمینه آغاز می‌شود.

۲-۱۶ مفهوم رسیدگی به استثناء

منطق یک برنامه، تست مکرر شرایط برای تعیین نحوه عملکرد اجرای برنامه است. به شبه کد زیر توجه کنید:

Perform a task *(انجام یک وظیفه)*

If the preceding task did not execute correctly *(اگر وظیفه قبلی به درستی اجرا نشده باشد)*

Perform error processing *(فرآیند پردازش خطا انجام گیرد)*

Perform next task *(انجام وظیفه بعدی)*

If the preceding task did not execute carrectly *(اگر وظیفه قبلی بدرستی اجرا نشده باشد)*

Perform error processing *(فرآیند پردازش خطا انجام گیرد)*

...

در این شبه کد، کار با انجام یک وظیفه آغاز می‌شود. سپس تست می‌شود که وظیفه بدرستی به انجام رسیده است یا خیر. اگر چنین نباشد، فرآیند خطا به اجرا در می‌آید. در غیر اینصورت، برنامه با اجرای وظیفه بعدی بکار خود ادامه می‌دهد. اگر چه این فرم از رسیدگی به خطا کار می‌کند، اما کاربرد چنین منطقی از رسیدگی به خطا می‌تواند سبب پیچیده شدن ساختار برنامه شده و قرائت، اصلاح، نگهداری و دیباگ آنرا با مشکل مواجه کند. این امر در مورد برنامه‌های کاربردی بزرگ مصادق بیشتری دارد. در واقع، اگر تعدادی از مشکلات نهانی بندرت رخ دهند، چنین منطقی از برنامه و رسیدگی به خطا می‌تواند سبب کاهش کارایی برنامه شود، چرا که برنامه مجبور است تا به بررسی شرایط بیشتری بپردازد تا تعیین کند که آیا وظیفه مورد نظر می‌تواند اجرا شود یا خیر.

ویژگی رسیدگی به استثناء به برنامه‌نویس امکان می‌دهد تا اقدام به حذف کد رسیدگی کننده به خطا از "خط اصلی" در مسیر اجرای برنامه نماید. با این عمل وضوح و کارایی برنامه افزایش می‌یابد. برنامه‌نویسان می‌تواند در مورد اینکه می‌خواهند به کدام استثناء رسیدگی کنند، تصمیم بگیرند، (تمام انواع استثناءها، تمام استثناءها از یک نوع مشخص یا تمام استثناءها از یک گروه مرتبط با یکدیگر). چنین انعطافی احتمال نادیده گرفته شدن خطاها را کاهش داده و از اینرو پایداری برنامه افزایش می‌یابد.

تست و خطایابی

رسیدگی به خطا تحمل‌پذیری برنامه‌ها را در مقابل خطا افزایش می‌دهد.

برنامه‌نویسی ایده‌ال

از بکارگیری رسیدگی به استثناء در مواردی بجز رسیدگی به خطا اجتناب کنید، چرا که چنین استفاده‌ای

می‌تواند وضوح برنامه را کاهش دهد.

زمانیکه از زبان‌های برنامه‌نویسی استفاده می‌شود که از ویژگی رسیدگی به استثناء پشتیبانی نمی‌کنند، غالباً برنامه‌نویسان نوشتن کدهای پردازش خطا را به تعویق می‌اندازند و گاهی اوقات افزودن آنها را به برنامه فراموش می‌کنند. در نتیجه توانایی و پایداری نرم‌افزار تولیدی کاهش می‌یابد. ++C برنامه‌نویسان را قادر می‌سازد تا به روش مناسبی رسیدگی به استثناء را در پروژه‌های خود وارد سازند.

مهندسی نرم‌افزار

در گذشته، برنامه‌نویسان از تکنیک‌های متفاوتی برای پیاده‌سازی کدهای پردازش خطا استفاده می‌کردند. در حالیکه رسیدگی به استثناء ارائه کننده یک تکنیک واحد و متحد برای پردازش خطا عرضه می‌کند.

کارایی

زمانیکه هیچ استثنای رخ ندهد، کد رسیدگی به استثناء، در کارایی برنامه تأثیر منفی نخواهد داشت.

کارایی

از ویژگی استثناء فقط در مورد مشکلاتی استفاده کنید که بندرت رخ می‌دهند.

۱۶-۳ مثال: رسیدگی به خطای تقسیم بر صفر

اجازه دهید تا به بررسی یک مثال ساده از رسیدگی به خطا بپردازیم (شکل‌های ۱۶-۱ و ۱۶-۲). هدف از این مثال اجتناب از یک مشکل رایج در محاسبات یعنی عملیات تقسیم بر صفر است. در ++C، تقسیم بر صفر با استفاده از محاسبات صحیح سبب می‌شود تا برنامه بصورت نابهنگام خاتمه یابد. در محاسبات اعشاری، تقسیم بر صفر امکان‌پذیر است، که حاصل آن بی‌نهایت منفی یا مثبت است که بصورت INF یا – INF به نمایش در می‌آید.

در این مثال، تابعی بنام **quotient** (خارج قسمت) تعریف می‌کنیم که دو ورودی از نوع صحیح از سوی کاربر دریافت و اولین پارامتر **int** خود را به دومین پارامتر **int** تقسیم می‌کند. قبل از مبادرت به تقسیم، تابع مقدار پارامتر اول را به نوع **double** تبدیل می‌کند. سپس مقدار دومین پارامتر به نوع **double** ارتقا داده می‌شود تا محاسبه صورت گیرد. بنابر این تابع **quotient** در واقع تقسیم را با استفاده از دو مقدار **double** انجام می‌دهد و حاصل آن نیز یک مقدار **double** خواهد بود.

اگرچه تقسیم بر صفر در محاسبات اعشاری امکان‌پذیر است، اما برای برآورد کردن هدف این مثال، فرض می‌کنیم هر گونه تقسیم بر صفر یک خطا محسوب می‌شود. بنابر این، تابع **quotient** پارامتر دوم خود را تست می‌کند تا مطمئن گردد که صفر نباشد، قبل از اینکه اجازه تقسیم داده شود. اگر پارامتر دوم، صفر

باشد، تابع با استفاده از یک استثنا به فراخوان نشان می‌دهد که یک مشکل رخ داده است. سپس فراخوان (در این مثال **main**) می‌تواند این استثنا را پردازش کرده و به کاربر اجازه دهد دو مقدار جدید را قبل از فراخوانی مجدد **quotient** وارد سازد. به این روش، برنامه می‌تواند به اجرای خود ادامه دهد حتی پس از اینکه یک مقدار اشتباه وارد شده باشد، بنابر این برنامه از استحکام بیشتری برخوردار خواهد شد.

این مثال، متشکل از دو فایل است: *DivideByZeroException.h* یک کلاس استثنا تعریف می‌کند که نشاندهنده نوع مشکلی است که امکان دارد در این مثال رخ دهد (شکل ۱-۱۶) و *fig16_02.cpp* که تعریف کننده تابع **quotient** و تابع **main** است که آنرا فراخوانی می‌کند. تابع main حاوی کدی است که به توصیف رسیدگی به استثنا می‌پردازد.

تعریف یک کلاس استثنا برای نمایش نوع مشکلی که اتفاق افتاده است

برنامه شکل ۱-۱۶ تعریف کننده کلاس **DivideByZeroException** بعنوان یک کلاس مشتق شده از کلاس **runtime_error** کتابخانه استاندارد است (تعریف شده در فایل سرآیند <**stdexcept**>) کلاس **runtime_error** که از کلاس **exception** مشتق شده است (تعریف شده در فایل سرآیند <**exception**>)، کلاس مبنا استاندارد C++ در عرضه خطاهای زمان اجرا است. کلاس **exeception** کلاس مبنا استاندارد C++ برای تمام استثناها می‌باشد (در بخش ۱۳-۱۶ با این کلاس بیشتر آشنا خواهید شد). یک نمونه از کلاس استثنا که از کلاس **runtime_error** مشتق می‌شود، فقط یک سازنده تعریف می‌کند (همانند، خطوط 13-12) که یک رشته پیغام خطا را به سازنده کلاس مبنا **runtime_error** ارسال می‌نماید. هر کلاس استثنا که مستقیماً یا غیرمستقیم از **exception** مشتق می‌شود حاوی یک تابع مجازی یا **virtual** بنام **what** است که، یک پیغام خطا از شی استثنا برگشت می‌دهد. دقت کنید که مجبور نیستید از یک کلاس استثنا رایج عمل مشتق را انجام دهید، همانند **DividoeByZeroException** از کلاس‌های استثناء استاندارد تدارک دیده شده توسط C++. با این همه، انجام اینکار به برنامه‌نویسان امکان استفاده از تابع مجازی **what** را برای بدست آوردن پیغام مناسبی از خطای رخ داده، فراهم می‌آورد. در برنامه شکل ۲-۱۶ از یک شی کلاس **DivideByZeroExeception** استفاده کرده‌ایم تا نشان دهنده زمان اقدام به عملیات تقسیم بر صفر باشد.

```
1  // Fig. 16.1: DivideByZeroException.
2  // Class DivideByZeroException definition.
3
4  #include <stdexcept> // stdexcept header file contains runtime_error
5  using std::runtime_error; // standard C++ library class runtime_error
6
7  // DivideByZeroException objects should be thrown by functions
8  // upon detecting division-by-zero exceptions
9  class DivideByZeroException : public runtime_error
10  {
11 public:
12    // constructor specifies default error message
13    DivideByZeroException::DivideByZeroException()
```

```
14       : runtime_error( "attempted to divide by zero" ) {}
15 }; // end class DivideByZeroException
```

شکل ۱-۱۶ | تعریف کلاس DivideByZeroException.

```
1  // Fig. 16.2: Fig16_02.cpp
2  // A simple exception-handling example that checks for
3  // divide-by-zero exceptions.
4  #include <iostream>
5  using std::cin;
6  using std::cout;
7  using std::endl;
8  #include "DivideByZeroException.h" // DivideByZeroException class
9
10 // perform division and throw DivideByZeroException object if
11 // divide-by-zero exception occurs
12 double quotient( int numerator, int denominator )
13 {
14    // throw DivideByZeroException if trying to divide by zero
15    if ( denominator == 0 )
16       throw DivideByZeroException(); // terminate function
17
18    // return division result
19    return static_cast< double >( numerator ) / denominator;
20 } // end function quotient
21
22 int main()
23 {
24    int number1; // user-specified numerator
25    int number2; // user-specified denominator
26    double result; // result of division
27
28    cout << "Enter two integers (end-of-file to end): ";
29
30    // enable user to enter two integers to divide
31    while ( cin >> number1 >> number2 )
32    {
33       // try block contains code that might throw exception
34       // and code that should not execute if an exception occurs
35       try
36       {
37          result = quotient( number1, number2 );
38          cout << "The quotient is: " << result << endl;
39       } // end try
40
41       // exception handler handles a divide-by-zero exception
42       catch ( DivideByZeroException &divideByZeroException )
43       {
44          cout << "Exception occurred: "
45             << divideByZeroException.what() << endl;
46       } // end catch
47
48       cout << "\nEnter two integers (end-of-file to end): ";
49    } // end while
50
51    cout << endl;
52    return 0; // terminate normally
53 } // end main
```

```
Enter two integers (end-of-file to end): 100 7
The quotient is: 14.28.57

Enter two integers (end-of-file to end): 100 0
Exception occurred: attempted to divide by zero

Enter two integers (end-of-file to end):^z
```

شکل ۲-۱۶ |مثالی از رسیدگی به استثنا که به هنگام تقسیم بر صفر استثناهای را راه‌اندازی می‌کند.

شرح رسیدگی به استثنا

در برنامه شکل ۲-۱۶ از رسیدگی به استثنا برای پوشاندن کدی که می‌تواند سبب‌ساز استثنا «تقسیم بر صفر» شود و برای رسیدگی به آن استفاده شده است. برنامه به کاربر امکان می‌دهد تا دو مقدار صحیح وارد سازد و آنها را بعنوان آرگومان‌های به تابع **quotient** ارسال می‌کند (خطوط 21-13). این تابع عدد اول (**numerator**) را به عدد دوم (**denominator**) تقسیم می‌کند. با فرض اینکه کاربر مقدار صفر را برای مقسوم علیه وارد نکرده باشد، تابع **quotient** نتیجه تقسیم را برگشت می‌دهد. با این وجود، اگر کاربر مبادرت به وارد کردن صفر بعنوان مقسوم علیه کند، تابع **quotient** یک استثنا به راه می‌اندازد. در خروجی نمونه این برنامه، دو خط ابتدائی نمایشی از موفقیت‌آمیز بودن محاسبه دارند، و دو خط بعدی نمایشی از واماندگی محاسبه در زمان به تقسیم بر صفر هستند. زمانیکه استثنا رخ می‌دهد، برنامه به کاربر اشتباه رخ داده را اطلاع می‌دهد و از وی می‌خواهد دو مقدار صحیح جدید وارد سازد. پس از بررسی کد، به بررسی ورودی‌های کاربر و جریان کنترل برنامه می‌پردازیم که حاصل آن این خروجی‌ها است.

احاطه کردن کد در بلوک try

برنامه با اعلان وارد کردن دو عدد صحیح به برنامه شروع می‌شود. مقادیر صحیح در شرط حلقه **while** وارد می‌شوند (خط 32). پس از اینکه کاربر مقادیری را بعنوان صورت کسر و مقسوم علیه (مخرج کسر) وارد کرد، کنترل برنامه، رهسپار بدنه حلقه می‌شود (خطوط 50-33).

خط 38 این مقادیر را به تابع **quotient** ارسال می‌کند (خطوط 21-13)، که عملیات تقسیم را انجام داده و نتیجه‌ای برگشت می‌دهد یا یک استثنا به راه می‌افتد (یعنی یک خطا رخ داده است) که نشاندهنده تقسیم بر صفر است. رسیدگی به خطا ابزاری برای تابع است که خطای را تشخیص می‌دهد که قادر به رسیدگی به آن نیست.

زبان C++ با تدارک دیدن بلوک‌های **try** قادر به رسیدگی به استثناء است. یک بلوک **try** متشکل از کلمه کلیدی **try** و بدنبال آن براکت‌ها (**{ }**) است که یک بلوک از کد را تعریف می‌کنند که احتمال دارد استثناهای در آنجا رخ دهد. بلوک **try** عباراتی را که احتمال دارد استثناهای را سبب شوند و عباراتی که بایستی در هنگام رخ دادن استثنا از آنها عبور شود، احاطه می‌کند.

به بلوک **try** در خطوط 40-36 توجه کنید که فراخوانی تابع **quotient** و عبارتی که نتیجه خطا را به نمایش در می‌آورد، احاطه کرده است. در این مثال، بدلیل اینکه احضار تابع **quotient** در خط 38 می‌تواند یک استثنا به راه بیاندازد، فراخوانی این تابع را در بلوک **try** قرار داده‌ایم. احاطه کردن عبارت خروجی (خط 39) در بلوک **try** ما را مطمئن می‌سازد که خروجی فقط در صورتی که تابع **quotient** نتیجه‌ای برگشت دهد، به نمایش درخواهد آمد.

تعریف رسیدگی catch برای پردازش DivideByZeroException

استثناها توسط رسیدگی کننده‌های **catch** پردازش می‌شوند، که استثناها را گرفته و پردازش می‌کنند. حداقل بایستی یک رسیدگی کننده **catch** بلافاصله بدنبال هر بلوک **try** آورده شود (خطوط 47-43). هر رسیدگی کننده **catch** با کلمه کلیدی **catch** و پارامتر استثنا در درون پرانتزها که نشاندهنده نوع استثنای است که **catch** می‌تواند پردازش نماید، آغاز می‌شود (در این مثال ***DivideByZeroException***).

زمانیکه یک استثنا در بلوک **try** رخ می‌دهد، رسیدگی کننده **catch** براساس نوع استثنا رخ داده شروع بکار می‌کند. اگر پارامتر استثنا شامل نام یک پارامتر اختیاری باشد، رسیدگی کننده **catch** می‌تواند از آن نام پارامتر برای تعامل با شی استثنا گرفتار شده در بدنه **catch** استفاده کند، که با براکت‌ها (**{}**) حدود آن تعیین شده است. معمولاً یک رسیدگی کننده **catch** یک گزارش خطا به کاربر عرضه کرده، آنرا در یک فایل واقعه (*log*) ثبت و برنامه را بطرز خوبی پایان داده یا سعی می‌کند با اعمال یک استراتژی جایگزین وظیفه شکست خورده را به انجام برساند. در این مثال، رسیدگی کننده **catch** فقط یک گزارش ساده عرضه می‌کند و نشان می‌دهد که کاربر اقدام به تقسیم بر صفر کرده است. سپس برنامه به کاربر اعلان می‌کند تا دو مقدار جدید وارد سازد.

خطای برنامه‌نویسی

*قرار دادن کد مابین یک بلوک **try** و رسیدگی کننده‌های **catch** متناظر، خطای نحوی است.*

خطای برنامه‌نویسی

*هر رسیدگی کننده **catch** فقط می‌تواند یک پارامتر داشته باشد، استفاده از کاما برای افزودن پارامتر، خطای نحوی است.*

مدل خاتمه‌دهی رسیدگی به استثنا

اگر یک استثنا بعنوان نتیجه یک عبارت در یک بلوک **try** رخ دهد، بلوک **try** به پایان می‌رسد (یعنی بلافاصله خاتمه می‌یابد). سپس برنامه جستجوی برای اولین رسیدگی کننده **catch** انجام می‌دهد که بتواند نوع استثنا رخ داده را پردازش نماید. برنامه، **catch** مناسب را با مقایسه نوع استثنا رخ داده با نوع هر پارامتر استثنا در هر **catch** انتخاب می‌کند. زمانیکه مطابقتی یافت شد، کد موجود در رسیدگی کننده **catch** مورد نظر اجرا می‌گردد. زمانیکه پردازش رسیدگی کننده **catch** با رسیدن به براکت بسته سمت راست (**}**) به پایان رسید، به آن استثنا رسیدگی شده است و متغیرهای محلی تعریف شده در رسیدگی کننده **catch** (شامل پارامتر **catch**) از قلمرو خارج می‌شوند. کنترل برنامه به نقطه‌ای که استثنا در آنجا رخ داده بود برگشت داده نمی‌شود (بنام نقطه راه‌اندازی یا پرتاب شناخته می‌شود)، چرا که بلوک **try** به پایان رسیده است. کنترل به اولین عبارت (خط 49) پس از آخرین رسیدگی کننده **catch** که بدنبال **try** آمده است، منتقل می‌شود. این مدل بعنوان *مدل خاتمه رسیدگی به استثنا* شناخته می شود. همانند هر بلوکی از کد، زمانیکه بلوک **try** خاتمه می‌یابد، متغیرهای محلی تعریف شده در آن بلوک از قلمرو خارج می‌شوند.

اگر بلوک **try** با موفقیت اجرای خود را کامل کند (یعنی بدون رخ دادن استثنا در بلوک **try**)، برنامه رسیدگی کننده‌های **catch** را نادیده گرفته، و کنترل برنامه با اولین عبارت پس از آخرین **catch** پس از آن بلوک **try** ادامه می‌یابد. اگر هیچ استثنای در یک بلوک **try** رخ ندهد، برنامه رسیدگی کننده‌های **catch** برای آن بلوک را نادیده خواهد گرفت.

اگر برای استثنائی که در یک بلوک **try** رخ داده، هیچ رسیدگی کننده **catch** مطابق با آن پیدا نشود، یا اگر استثنا در عبارتی رخ دهد که در یک بلوک **try** وجود ندارد، تابعی که حاوی آن عبارت است، بلافاصله خاتمه یافته و برنامه مبادرت به یافتن یک بلوک **try** احاطه کننده در تابع فراخوانی شده می‌کند. این فرآیند با نام *stack unwinding* یا *باز کردن پشته* شناخته می‌شود و در بخش ۸-۱۶ به بررسی آن خواهیم پرداخت.

جریان کنترل برنامه زمانیکه کاربر برای مخرج یک مقدار غیر صفر وارد می‌کند

با فرض اینکه کاربر مقدار 100 را برای صورت و 7 را برای مخرج وارد کرده باشد، به جریان کنترل توجه کنید (یعنی، دو خط ابتدایی در خروجی برنامه شکل ۲-۱۶). در خط 16، تابع **quotient** تعیین می‌کند که **denominator** برابر صفر نیست، از اینرو خط 20 عملیات تقسیم را انجام داده و نتیجه (14.2857) را به خط 38 بعنوان یک نوع **double** برگشت می‌دهد. سپس کنترل برنامه بصورت ترتیبی از خط 38 ادامه می‌یابد، بنابر این خط 39 نتیجه تقسیم را به نمایش در آورده و خط 40 انتهای بلوک **try** است. بدلیل اینکه بلوک **try** بطور موفقیت‌آمیز کامل شده و استثنای به وجود نیامده است، برنامه عبارات موجود در درون رسیدگی کننده **catch** را به اجرا در نیاورده (خطوط 47-43)، و کنترل با خط 49 ادامه می‌یابد که به کاربر اعلان می‌کند دو عدد صحیح وارد سازد.

جریان کنترل برنامه زمانیکه کاربر برای مخرج صفر وارد می‌کند

اکنون اجازه دهید به حالت جالبی بپردازیم که در آن کاربر برای صورت عدد 100 و برای مخرج عدد صفر وارد نماید (یعنی خطوط سوم و چهارم از خروجی شکل ۲-۱۶). در خط 16، **quotient** تعیین می‌کند که مخرج برابر صفر است، که دلالت بر اقدام تقسیم بر صفر دارد. خط 17 یک استثنا به راه می‌اندازد که آنرا بصورت یک شی از کلاس **DivideByZeroException** نشان داده‌ایم (شکل ۱-۱۶).

به استثنا به راه افتاده دقت کنید، خط 17 از کلمه کلیدی **throw** و بدنبال آن یک عملوند که نشاندهنده نوع استثناء به راه افتاده است، استفاده کرده است. معمولاً عبارت **throw** از یک عملوند استفاده می‌کند (در بخش ۵-۱۶، در ارتباط با استفاده از عبارات **throw** که عملوندی ندارند صحبت خواهیم کرد). عملوند یک **throw** می‌تواند از هر نوعی باشد. اگر عملوند یک شی باشد، آنرا یک شی استثنا می‌نامیم که در این مثال، شی استثنا یک شی از نوع **DivideByZeroException** است. با این همه، یک عملوند

throw می‌تواند با مقادیر دیگری در نظر گرفته شود، همانند مقدار در یک عبارت (مثال **throw x>5**) یا مقداری از یک **int** (مثال، **throw 5**). مثال‌های مطرح شده در این فصل انحصاراً بر روی شی‌های استثنا متمرکز هستند.

بعنوان بخشی از راه‌اندازی یک استثناء، عملوند **throw** ایجاد شده و برای مقداردهی اولیه پارامتر در رسیدگی کننده **catch** بکار گرفته می‌شود، که در مورد آن صحبت کوتاهی خواهیم کرد. در این مثال، عبارت **throw** در خط 17 یک شی از کلاس **DivideByZeroException** ایجاد می‌کند. زمانیکه خط 17 استثنا را به راه می‌اندازد، تابع **quotient** بلافاصله از صحنه خارج می‌شود. بنابر این، خط 17 مبادرت به راه‌اندازی استثنا قبل از اینکه **quotient** بتواند تقسیم موجود در خط 20 را به انجام برساند، می‌کند. این رفتار یکی از صفات اصلی رسیدگی به استثنا است: تابع بایستی یک استثنا را قبل از اینکه خطایی فرصت رخ دادن پیدا کند، راه‌اندازی نماید.

بدلیل اینکه تصمیم به احاطه کردن احضار تابع **quotient** در بلوک **try** گرفته‌ایم (خط 38)، کنترل برنامه وارد رسیدگی کننده **catch** می‌شود (خطوط 47-43) که بلافاصله بدنبال بلوک **try** آمده است. این رسیدگی کننده **catch** همانند یک رسیدگی کننده استثنا برای استثنا تقسیم بر صفر خدمت می‌کند. بطور کلی، زمانی که یک استثنا در درون یک بلوک **try** به راه می‌افتد، استثنا توسط یک رسیدگی کننده **catch** گرفتار می‌شود که نوع آن با نوع استثنا مطابقت دارد. در این برنامه، رسیدگی کننده **catch** تصریح می‌کند که شی **DivideByZeroException** را گرفتار ساخته است، این نوع مطابقت با نوع شی به راه افتاده در تابع **quotient** دارد. در واقع رسیدگی کننده **catch** مراجعه به شی **DivideByZeroException** ایجاد شده توسط عبارت **throw** تابع **quotient** را گرفتار می‌سازد.

بدنه رسیدگی **catch** در خطوط 46-45 پیغام خطای متناسب برگشتی از فراخوانی تابع **what** از کلاس مبنای **runtime_error** را به نمایش در می‌آورد. این تابع رشته‌ای برگشت می‌دهد که سازنده **DivideByZeroException** (خطوط 18-12 ازشکل ۱-۱۶) به سازنده کلاس مبنای **runtime_error** ارسال می‌کند.

۱۶-۴ زمان استفاده از رسیدگی به استثنا

رسیدگی به استثنا برای پردازش خطاهای همگام طراحی شده است، که در زمان اجرای یک عبارت رخ می‌دهند. مثال‌های رایج از این قبیل خطاها عبارتند از خارج شدن شاخص آرایه از مرزها، سرریز محاسباتی، تقسیم بر صفر، پارامترهای اشتباه و عدم اخذ موفقیت‌آمیز حافظه (به علت فقدان حافظه). رسیدگی به استثنا برای پردازش خطاهای مرتبط با رویدادهای ناهمگام طراحی نشده است (همانند عملیات

I/O دیسک، ارسال پیغام در شبکه، کلیک‌های ماوس و ضربه کلید)، که بصورت موازی و مستقل از جریان کنترل برنامه رخ می‌دهند.

همچنین مکانیزم رسیدگی به استثنا برای پردازش مسائلی که در زمان تعامل برنامه با عناصر نرم‌افزاری همانند توابع عضو، سازنده‌ها، نابود کننده‌ها و کلاس رخ می‌دهند، مناسب است.

معمولاً برنامه‌های کاربردی پیچیده، متشکل از کامپونت‌های نرم‌افزاری از پیش تعریف شده و کامپونت‌های خاص برنامه هستند که از کامپونت‌های از پیش تعریف شده استفاده می‌کنند. زمانیکه یک کامپونت از پیش تعریف شده با مشکلی مواجه می‌شود، آن کامپونت نیاز به مکانیزمی برای بیان مشکل با کامپونت خاص برنامه دارد، در واقع کامپونت از پیش تعریف شده اطلاعی از اینکه برنامه چگونه مشکل رخ داده را پردازش می‌کند، ندارد.

۱۶-۵ راه‌اندازی مجدد استثنا

امکان دارد که یک رسیدگی کننده به استثنا، به محض دریافت یک استثنا، تصمیم بگیرد که نمی‌تواند استثنا را پردازش کند یا می‌تواند اندکی از آنرا پردازش نماید. در چنین مواردی رسیدگی کننده می‌تواند رسیدگی به استثنا را به رسیدگی کننده استثنا دیگری تسلیم نماید (یا بخشی از آنرا). در هر دو مورد، رسیدگی کننده اینکار را با راه‌اندازی مجدد استثنا از طریق عبارت زیر به انجام می‌رساند

```
throw;
```

صرفنظر از اینکه یک رسیدگی کننده بتواند استثنایی را پردازش کند یا نه، رسیدگی کننده می‌تواند برای پردازش استثنا در خارج از رسیدگی کننده، آنرا مجدداً راه‌اندازی کند. بلوک **try** بعدی راه‌اندازی مجدد استثنا را تشخیص می‌دهد، که یک **catch** قرار گرفته پس از بلوک **try** مبادرت به رسیدگی به استثنا می‌کند.

برنامه شکل ۳-۱۶ به بررسی راه‌اندازی مجدد یک استثنا پرداخته است. در بلوک **try** خطوط 37-32، خط 35 تابع **throwException** را فراخوانی می‌کند (خطوط 27-11). تابع **throwException** نیز حاوی یک بلوک **try** است (خطوط 18-14) که عبارت throw در خط 17 یک نمونه از کلاس کتابخانه استاندارد **exception** را راه‌اندازی می‌کند. رسیدگی کننده **catch** این تابع در خطوط 24-19 این استثنا را گرفته، پیغام خطا چاپ کرده (خطوط 22-21) و استثنا را مجدداً راه‌اندازی می‌کند (خط 23). با اینکار تابع **throwException** خاتمه یافته و کنترل به خط 35 در بلوک **try…catch** در **main** باز می‌گردد. بلوک **try** خاتمه یافته (از اینرو خط 36 اجرا نمی‌شود) و رسیدگی کننده **catch** در **main** این استثنا را گرفته (خطوط 41-38) و پیغام خطا را چاپ می‌کند (خط 40).

```
1  // Fig. 16.3: Fig16_03.cpp
2  // Demonstrating exception rethrowing.
3  #include <iostream>
```

```
4  using std::cout;
5  using std::endl;
6
7  #include <exception>
8  using std::exception;
9
10 // throw, catch and rethrow exception
11 void throwException()
12 {
13    // throw exception and catch it immediately
14    try
15    {
16       cout << "  Function throwException throws an exception\n";
17       throw exception(); // generate exception
18    } // end try
19    catch ( exception & ) // handle exception
20    {
21       cout << "  Exception handled in function throwException"
22          << "\n  Function throwException rethrows exception";
23       throw; // rethrow exception for further processing
24    } // end catch
25
26    cout << "This also should not print\n";
27 } // end function throwException
28
29 int main()
30 {
31    // throw exception
32    try
33    {
34       cout << "\nmain invokes function throwException\n";
35       throwException();
36       cout << "This should not print\n";
37    } // end try
38    catch ( exception & ) // handle exception
39    {
40       cout << "\n\nException handled in main\n";
41    } // end catch
42
43    cout << "Program control continues after catch in main\n";
44    return 0;
45 } // end main
```

```
main invokes function throwException
 Function throwException throws an exception
 Exception handled in function throwException
 Function throwException rethrows exception

 Exeption hendled in main
 Program control continues after catch in main
```

شکل ۳-۱٦ | راه‌اندازی مجدد استثنا.

۱٦-٦ مشخصات استثنا

یکی از مشخصات اختیاری استثنا لیستی از استثناها است که یک تابع می‌تواند آنها را به جریان در آورد. برای مثال، به اعلان تابع زیر توجه کنید

```
int someFunction( double value )
      throw( ExceptionA, ExceptionB, ExceptionC)
{
   //function body
}
```

در این تعریف، مشخصات استثنا، که با کلمه کلیدی **throw** شروع شده و بدنبال آن پرانتزهای با لیست پارامتری تابع قرار دارند، بر این نکته دلالت دارد که تابع **someFunction** می‌تواند استثناهای از نوع

ExceptionA، **ExecptionB** و **ExceptionC** راه‌اندازی کند. تابع فقط می‌تواند استثناهای از نوع‌های مشخص شده در مشخصات را راه‌اندازی کند یا استثناهای از هر نوع مشتق شده از این نوع‌ها. اگر تابع استثنائی را سبب شود که متعلق به یک نوع مشخص شده باشد، تابع **unexpected** فراخوانی می‌شود، که معمولاً به برنامه خاتمه می‌دهد.

تابعی که مشخصات استثنا برای آن تدارک دیده نشده است، می‌تواند هر نوع استثنایی را به راه بیاندازد. با قرار دادن **throw()** پس از لیست پارامتری یک تابع، نشان داده می‌شود که تابع استثناهای را به جریان نخواهد انداخت. اگر تابع مبادرت به راه‌اندازی یک استثنا کند، تابع **unexpected** احضار می‌شود. در بخش ۷-۱۶ نشان داده خواهد شد که چگونه تابع **unexpected** می‌تواند با فراخوانی تابع **set_unexpected** بهینه‌سازی شود.

۷-۱۶ پردازش استثناهای غیرمنتظره (unexpected)

تابع **unexpected** تابع ثبت شده با تابع **set_unexpected** را فراخوانی می‌کند (تعریف شده در فایل سرآیند <**exception**>). اگر هیچ تابعی به این روش ثبت نشده باشد، تابع **terminate** بصورت پیش‌فرض فراخوانی می‌شود. حالت‌های که تابع **terminate** فراخوانی می شود عبارتند از:

۱- مکانیزم استثنا قادر به یافتن یک **catch** مطابق با استثنا رخ داده نباشد.

۲- نابود کننده‌ای مبادرت به راه‌اندازی یک استثنا در زمان باز کردن پشته کند.

۳- مبادرت به راه‌اندازی مجدد یک استثنا شود زمانیکه به استثنا جاری رسیدگی نشده است.

۴- فراخوانی تابع **unexpected** در حالت پیش‌فرض که تابع **terminate** را فراخوانی می‌کند.

تابع **set_terminate** می‌تواند تصریح کننده تابعی باشد که به هنگام فراخوانی **terminate** احضار می‌شود. در غیر اینصورت **terminate** مبادرت به فراخوانی **abort** می‌کند، که برنامه را بدون فراخوانی نابود کننده‌های شی‌های باقیمانده از کلاس ذخیره‌سازی استاتیک یا اتوماتیک خاتمه می‌دهد. اینکار می‌تواند سبب فقدان منابع شود چرا که برنامه نابهنگام خاتمه می‌پذیرد.

هر یک از توابع **set_terminate** و **set_unexpected** یک اشاره‌گر به آخرین تابع فراخوانی شده توسط **terminate** و **unexpected** برگشت می‌دهند (صفر، در اولین فراخوانی). این ویژگی به برنامه‌نویس امکان می‌دهد تا اشاره‌گر تابع را ذخیره کرده و از اینرو بتواند آنرا بعداً بازیابی کند.

توابع **set_terminate** و **set_unexpected** اشاره‌گرهای بعنوان آرگومان برای توابعی که نوع برگشتی آنها **void** بوده و آرگومانی ندارند، دریافت می‌کند.

اگر آخرین عمل تابع خاتمه دهنده تعریف شده از سوی برنامه‌نویس نتواند به برنامه خاتمه دهد، تابع **abort** فراخوانی شده و به اجرای برنامه خاتمه می‌دهد (البته پس از اجرای عبارات تابع خاتمه دهنده تعریف شده از سوی برنامه‌نویس).

۱۶-۸ باز کردن پشته

زمانیکه یک استثنا به راه می‌افتد اما در یک قلمرو خاص گرفتار نمی‌شود، پشته فراخوانی تابع باز می‌شود، و سعی می‌شود تا استثنا در بلوک **try...catch** خارجی گرفتار گردد. منظور از باز کردن پشته فراخوانی تابع این است که در تابعی که استثنا رخ داده و نتوانسته گرفتار شود، تمام متغیرهای محلی در آن تابع نابود شده، تابع خاتمه یافته و کنترل به عبارتی که در اصل آن تابع را فراخوانی کرده برگشت داده شود. اگر یک بلوک **try** آن عبارت را احاطه کرده باشد، مبادرت به گرفتن استثنا خواهد کرد. اگر بلوک **try** آن عبارت را احاطه نکرده باشد، مجدداً باز کردن پشته صورت خواهد گرفت. اگر هیچ رسیدگی کننده **catch** این استثنا را گرفتار نسازد، تابع **terminate** فراخوانی می‌شود تا برنامه را خاتمه دهد. برنامه شکل ۱۶-۴ به بررسی باز کردن پشته پرداخته است.

```
1  // Fig. 16.4: Fig16_04.cpp
2  // Demonstrating stack unwinding.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <stdexcept>
8  using std::runtime_error;
9
10 // function3 throws run-time error
11 void function3() throw ( runtime_error )
12 {
13    cout << "In function 3" << endl;
14
15    // no try block, stack unwinding occur, return control to function2
16    throw runtime_error( "runtime_error in function3" );
17 } // end function3
18
19 // function2 invokes function3
20 void function2() throw ( runtime_error )
21 {
22    cout << "function3 is called inside function2" << endl;
23    function3(); // stack unwinding occur, return control to function1
24 } // end function2
25
26 // function1 invokes function2
27 void function1() throw ( runtime_error )
28 {
29    cout << "function2 is called inside function1" << endl;
30    function2(); // stack unwinding occur, return control to main
31 } // end function1
32
33 // demonstrate stack unwinding
34 int main()
35 {
36    // invoke function1
37    try
38    {
39       cout << "function1 is called inside main" << endl;
```

```
40       function1(); // call function1 which throws runtime_error
41    } // end try
42    catch ( runtime_error &error ) // handle run-time error
43    {
44       cout << "Exception occurred: " << error.what() << endl;
45       cout << "Exception handled in main" << endl;
46    } // end catch
47
48    return 0;
49 } // end main
```

```
function1 is called inside main
function2 is called inside function1
function3 is called inside function2
In function 3
Exception occurred: runtime_error in function3
Exception handled in main
```

شکل ٤-١٦ | باز کردن پشته (stack unwind).

در **main**، بلوک **try** (خطوط 41-37) مبادرت به فراخوانی تابع **funcation1** می‌کند (خطوط 31-27). سپس تابع **function1**، تابع **function2** را فراخوانی می‌کند (خطوط 24-20)، که آن هم تابع **function3** را فراخوانی می‌کند (خطوط 17-11). خط 16 از **function3** یک شی **runtime_error** را راه‌اندازی می‌کند. با این وجود، چون هیچ بلوک **try** عبارت **throw** را احاطه نکرده است (در خط 16)، باز کردن پشته رخ می‌دهد، **function3** در خط 16 خاتمه می‌یابد، سپس کنترل به عبارتی در **function2** برگشت داده می‌شود که **function3** را فراخوانی کرده بود (یعنی خط 23). چون هیچ بلوک **try**، خط 23 را احاطه نکرده است، مجدداً باز کردن پشته رخ می‌دهد، **function2** خاتمه یافته (در خط 23) و کنترل به عبارتی در **function1** برگشت داده می‌شود که **function2** را فراخوانی کرده بود (یعنی خط 30). چون هیچ بلوک **try**، خط 30 را احاطه نکرده است، باز کردن پشته یکبار دیگر اتفاق می‌افتد، **function1** در خط 30 خاتمه یافته و کنترل به عبارتی در **main** باز می‌گردد که تابع **function1** را فراخوانی کرده بود (یعنی خط 40). بلوک try در خطوط 41-37 این عبارت را احاطه کرده است، از اینرو اولین رسیدگی کننده **catch** قرار گرفته پس از این بلوک **try** پیدا شده (خطوط 46-42) و استثنا را گرفتار و آنرا پردازش می‌کند. خط 44 از تابع **what** برای نمایش پیغام استثنا استفاده کرده است. بخاطر داشته باشید که تابع **what** یک تابع **virtual** (مجازی) از کلاس **exception** است که می‌تواند توسط یک کلاس مشتق شده به کنار گذاشته شود تا پیغام خطای مناسب برگشت داده شود.

۹-۱٦ سازنده‌ها، نابود کننده‌ها و رسیدگی به استثنا

ابتدا اجازه دهید تا به بررسی مسئله‌ی بپردازیم که قبلاً مطرح کردیم، اما به قدر کفایت راضی کننده نبود: زمانیکه یک خطا در یک سازنده تشخیص داده می‌شود چه اتفاقی می‌افتد؟ برای مثال، چگونه باید سازنده یک شی به هنگام رخ دادن واماندگی **new** از خود واکنش نشان دهد، چرا که سازنده قادر نیست تا حافظه مورد نیاز برای ذخیره‌سازی نمایندگی درونی آن شی را اخذ کند؟ بدلیل اینکه سازنده نمی‌تواند مقداری را

که نشاندهنده یک خطا است برگشت دهد، بایستی یک روش جایگزین انتخاب کنیم که نشان دهد شی بدرستی ایجاد نشده است. یک روش این است که شی که بدرستی ایجاد نشده است را برگشت داده و امیدوار باشیم که هر کسی که از آن استفاده می‌کند، تست یا آزمون‌های مناسبی بر روی آن اعمال کند تا مشخص شود آن شی در وضعیت پایداری قرار ندارد. روش دیگر تنظیم چند متغیر خارج از سازنده است.

شاید بهترین جایگزین این باشد که سازنده را ملزم به راه‌اندازی به یک استثنا کنیم که حاوی اطلاعات خطا باشد و از اینرو به برنامه فرصت مناسبی برای رسیدگی به واماندگی داده می‌شود. راه‌اندازی استثناها توسط یک سازنده سبب می‌شود نابود کننده‌ها برای هر شی که بعنوان بخشی از شی قبل از راه‌اندازی استثنا ایجاد شده‌اند، فراخوانی شوند. نابود کننده‌ها برای هر شی اتوماتیک ایجاد شده در هر بلوک **try** قبل از اینکه استثنا رخ داده باشد، فراخوانی می‌شود. باز کردن پشته تضمینی بر انجام کار از نقطه‌ای است که رسیدگی کننده به استثنا شروع به اجرا شدن می‌کند. اگر نابود کننده‌ای بعنوان نتیجه‌ای از باز کردن پشته فراخوانی گردد، تابع **terminate** فراخوانی خواهد شد.

اگر شی دارای شی‌های عضو باشد، و اگر یک استثنا قبل از اینکه شی خارجی کاملاً ایجاد شود، اتفاق افتد، پس نابود کننده‌ها برای شی‌های عضوی که قبل از رخ دادن استثنا ساخته شده‌اند، فراخوانی خواهند شد. اگر آرایه‌ای از شی‌های به هنگام رخ دادن یک استثنا تا حدی ایجاد شده باشد، نابود کننده‌ها فقط بر روی شی‌های ساخته شده در آرایه فراخوانی خواهند شد.

یک استثنا می‌تواند مانع از عملیات کدی شود که می‌خواهد بطرز عادی منبعی را رها سازد، از اینرو سبب، فقدان منبع می‌شود، یکی از تکنیک‌های حل این مشکل، مقداردهی اولیه یک شی محلی با آن منبع است. زمانیکه استثنا رخ می‌دهد، نابود کننده برای آن شی فراخوانی و می‌تواند منبع را آزاد کند.

۱۰-۱۶ استثناها و توارث

کلاس‌های استثنا گوناگونی را می‌توان از یک کلاس مبنای مشترک یا عمومی مشتق کرد، همانطوری که در بخش ۳-۱۶ زمانیکه کلاس **DivideByZeroException** را بعنوان یک کلاس مشتق شده از کلاس **exception** ایجاد کردیم. اگر یک رسیدگی کننده **catch** یک اشاره‌گر یا مراجعه‌ای را به یک شی استثنا از یک نوع کلاس مبنا، را گرفتار سازد، در ضمن می‌تواند یک اشاره‌گر یا مراجعه به تمام شی‌ها از کلاس‌های عمومی مشتق شده از آن کلاس مبنا را هم گرفتار کند. این فرآیند امکان پردازش چند ریختی خطاهای مرتبط را فراهم می‌آورد.

۱۱-۱۶ پردازش واماندگی new

زبان C++ استاندارد تصریح می‌کند که در زمان واماندگی عملگر **new**، استثنا **bad_alloc** به راه می‌افتد (تعریف شده در فایل سرآیند <**new**>). با این وجود، برخی از کامپایلرها سازگار با C++ استاندارد نیستند و بنابر این از آن نسخه new استفاده می‌کند که به هنگام واماندگی، صفر برگشت می‌دهد. برای مثال Microsoft Visual Studio .NET در زمان واماندگی **new** یک استثنا **bad_alloc** ایجاد می‌کند، در حالیکه Microsoft Visual C++ 6.0 در زمان واماندگی **new**، مقدار صفر برگشت می‌دهد.

کامپایلر در رسیدگی به واماندگی **new** به روش‌های متفاوتی عمل می‌کنند. بسیاری از کامپایلرهای قدیمی **C++** به هنگام واماندگی **new** بصورت پیش‌فرض صفر برگشت می‌دهند. برخی از کامپایلرها اگر فایل سرآیند <**new**> (یا <**new.h**>) بکار گرفته شده باشد، از راه‌اندازی یک استثنا توسط **new** پشتیبانی می‌کنند. کامپایلرهای دیگر در حالت پیش‌فرض **bad_alloc** را به راه می‌اندازند، صرفنظر از اینکه آیا فایل سرآیند <**new**> بکار گرفته شده است یا خیر. برای کسب اطلاعات بیشتر در این زمینه به مستندات کامپایلر خود مراجعه کنید.

در این بخش، به عرض سه مثال در ارتباط با واماندگی **new** می‌پردازیم. مثال اول در زمان واماندگی **new**، صفر برگشت می‌دهد. مثال دوم از نسخه‌ای از **new** استفاده می‌کند که در زمان واماندگی **new** یک استثنا **bad_alloc** به راه می‌اندازد. مثال سوم از تابع **set_new_handler** برای رسیدگی به واماندگی **new** استفاد می‌کند. [نکته: مثال‌های مطرح شده در شکل‌های ۵-۱۶ الی ۷-۱۶ میزان بسیاری زیادی از حافظه دینامیکی کامپیوتر را اخذ می‌کنند، از اینرو می‌توانند کامپیوتر شما را آهسته نمایند.]

برگشت دادن صفر در واماندگی new

برنامه شکل ۵-۱۶ به بررسی **new** پرداخته که در زمان واماندگی (یعنی زمانیکه تقاضای اخذ مقداری از حافظه را می‌کند و موفق نمی‌شود) صفر برگشت می‌دهد. عبارت **for** در خطوط 14-13 حلقه‌ای بوجود آورده که 50 بار تکرار می‌شود و در هر گذار، یک آرایهٔ 50.000.000 برای مقادیر از نوع **double** اخذ می‌کند (یعنی 400.000.000 بایت، چرا که یک double معمولاً 8 بایت است). عبارت **if** در خط 17 نتیجه هر عملیات new را تست می‌کند تا مشخص کند که آیا اخذ حافظه توسط **new** با موفقیت همراه بوده است یا خیر. اگر new دچار واماندگی شود و صفر برگشت دهد، خط 19 یک پیغام خطا چاپ کرده و حلقه خاتمه می‌یابد. [نکته: ما از Microsoft Visual C++ 6.0 در اجرای این مثال استفاده کرده‌ایم، چرا که Microsoft Visual Studio .NET در زمان واماندگی **new** بجای صفر، یک استثنا **bad_alloc** راه‌اندازی می‌کند.]

```
1  // Fig. 16.5: Fig16_05.cpp
2  // Demonstrating pre-standard new returning 0 when memory
3  // is not allocated.
```

```
4  #include <iostream>
5  using std::cerr;
6  using std::cout;
7
8  int main()
9  {
10    double *ptr[ 50 ];
11
12    // allocate memory for ptr
13    for ( int i = 0; i < 50; i++ )
14    {
15       ptr[ i ] = new double[ 50000000 ];
16
17       if ( ptr[ i ] == 0 ) // did new fail to allocate memory
18       {
19          cerr << "Memory allocation failed for ptr[ " << i << " ]\n";
20          break;
21       } // end if
22       else // successful memory allocation
23          cout << "Allocated 50000000 doubles in ptr[ " << i << " ]\n";
24    } // end for
25
26    return 0;
27 } // end main
```

```
Allocated 50000000 doubles in ptr[0]
Allocated 50000000 doubles in ptr[1]
Allocated 50000000 doubles in ptr[2]
Memory allocation failed for ptr[3]
```

شکل ٥-١٦ | برگشت صفر در زمان واماندگی new.

خروجی برنامه نشان می‌دهد که برنامه فقط سه بار قادر به تکرار حلقه قبل از واماندگی new بوده و حلقه خاتمه یافته است. براساس میزان حافظه فیزیکی، فضای دیسک در دسترس برای حافظه مجازی بر روی سیستم شما و کامپایلری که بکار گرفته‌اید، احتمالاً خروجی شما با خروجی نشان داده شده در اینجا متفاوت خواهد بود.

راه‌اندازی bad_alloc در زمان واماندگی new

برنامه شکل ۶-۱۶ نشان می‌دهد که در زمان واماندگی **new** در اخذ حافظه مورد نیاز، **bad_alloc** راه‌اندازی شده است. عبارت **for** در خطوط 24-20 در درون بلوک **try** بایستی 50 بار تکرار شود و در هر گذار، یک آرایه 50.000.000 برای مقادیر **double** اخذ شود، اگر **new** دچار واماندگی شود، یک استثنا **bad_alloc** به راه افتاده، حلقه خاتمه می‌یابد و برنامه از خط 28 بکار خود ادامه می‌دهد محلی که رسیدگی کننده **catch** استثنا را گرفتار کرده و آنرا پردازش می‌کند.

```
1  // Fig. 16.6: Fig16_06.cpp
2  // Demonstrating standard new throwing bad_alloc when memory
3  // cannot be allocated.
4  #include <iostream>
5  using std::cerr;
6  using std::cout;
7  using std::endl;
8
9  #include <new> // standard operator new
10 using std::bad_alloc;
11
12 int main()
13 {
14    double *ptr[ 50 ];
15
16    // allocate memory for ptr
```

```
17    try
18    {
19       // allocate memory for ptr[ i ]; new throws bad_alloc on failure
20       for ( int i = 0; i < 50; i++ )
21       {
22          ptr[ i ] = new double[ 50000000 ]; // may throw exception
23          cout << "Allocated 50000000 doubles in ptr[ " << i << " ]\n";
24       } // end for
25    } // end try
26
27    // handle bad_alloc exception
28    catch ( bad_alloc &memoryAllocationException )
29    {
30       cerr << "Exception occurred: "
31          << memoryAllocationException.what() << endl;
32    } // end catch
33
34    return 0;
35 } // end main
```

```
Allocated 50000000 doubles in ptr[0]
Allocated 50000000 doubles in ptr[1]
Allocated 50000000 doubles in ptr[2]
Exception occurred: bad allocation
```

شکل ۶-۱۶ | برگشت bad_alloc در زمان واماندگی new.

رسیدگی کننده **catch** استثنا را گرفته و پردازش می‌کند. خطوط 31-30 پیغام "Exception occurred:" را بدنبال پیغام برگشتی از تابع **what** چاپ می‌کنند. خروجی نشان می‌دهد که برنامه فقط سه بار حلقه را قبل از واماندگی **new** و راه افتادن استثنا **bad_alloc** تکرار کرده است. ممکن خروجی برنامه بر روی کامپیوتر شما با خروجی این برنامه متفاوت باشد.

زبان C++ استاندارد تصریح می‌کند که کامپایلرهای استاندارد سازگار می‌تواند به استفاده از نسخه‌ای از **new** که در مواجه با واماندگی صفر برگشت می‌دهند، ادامه دهند. به همین منظور، فایل سرآیند <**new**> شی **nothrow** (از نوع **nothrow_t**) را تعریف کرده است که بصورت زیر بکار گرفته می‌شود:

```
double *ptr = new( nothrow ) double[50000000];
```

عبارت فوق از نسخه‌ای از **new** استفاده می‌کند که به هنگام اخذ حافظه برای آرایه 50.000.000 از نوع **double** استثنا **bad_alloc** را سبب نمی‌شود (یعنی **nothrow**)..

رسیدگی به واماندگی new با استفاده از تابع set_new_handler

یکی از ویژگیهای دیگر در رسیدگی به واماندگی **new** تابع **set_new_handler** است (نمونه اولیه در فایل سرآیند استاندارد <**new**>). این تابع یک اشاره‌گر به تابعی که هیچ آرگومانی دریافت نمی‌کند و **void** برگشت می‌دهد، بعنوان آرگومان دریافت می‌کند. این اشاره‌گر به تابعی اشاره دارد که اگر **new** دچار واماندگی شود، فراخوانی خواهد شد. این قابلیت یک روش منسجم در رسیدگی به تمام واماندگی‌های **new**، صرفنظر از اینکه واماندگی در کجای برنامه رخ داده است، در اختیار برنامه‌نویس قرار می‌دهد. زمانیکه **set_new_handler** یک رسیدگی کننده **new** در برنامه را ثبت کرد، عملگر **new** نمی‌تواند در زمان واماندگی مبادرت به راه‌اندازی **bad_alloc** کند و بجای آن، خطا را تحویل تابع رسیدگی کننده **new** می‌دهد.

اگر **new** موفق شود، حافظه مورد نیاز را اخذ کند، یک اشاره‌گر به آن حافظه برگشت خواهد داد. اگر **new** در اخذ حافظه دچار واماندگی شود و **set_new_handler** برای یک تابع رسیدگی کننده **new** ثبت نشده باشد، آنگاه یک استثنا **bad_alloc** به جریان خواهد انداخت. اگر **new** در اخذ حافظه دچار واماندگی شود و تابع رسیدگی کننده **new** ثبت شده باشد، این تابع فراخوانی خواهد شد. C++ استاندارد تصریح می‌کند که تابع رسیدگی کننده **new** بایستی یکی از وظایف زیر را انجام دهد:

۱- تهیه حافظه مورد نیاز با حذف سایر حافظه‌های اخذ شده دینامیکی (یا به کاربر اعلان شود تا برنامه‌های دیگر را خاتمه دهد) و برگشت به عملگر **new** برای مبادرت به اخذ مجدد حافظه.

۲- راه‌اندازی یک استثنا از نوع **bad_alloc**.

۳- فراخوانی تابع **abort** یا **exit** (که هر دو در فایل سرآیند <**cstdlib**> وجود دارند) برای خاتمه دادن برنامه.

برنامه شکل ۷-۶ به بررسی **set_new_handler** پرداخته است. تابع **customNewHandler** در خطوط 14-18 یک پیغام خطا چاپ کرده (خط 16)، سپس برنامه را از طریق فراخوانی **abort** خاتمه می‌دهد (خط 17). خروجی نشان می‌دهد که برنامه فقط سه بار قبل از واماندگی **new** و فراخوانی تابع **customNewHandler** موفق به تکرار حلقه شده است. مجدداً احتمال دارد خروجی این برنامه بر روی کامپیوتر شما با خروجی به نمایش در آمده در اینجا متفاوت باشد.

```
1  // Fig. 16.7: Fig16_07.cpp
2  // Demonstrating set_new_handler.
3  #include <iostream>
4  using std::cerr;
5  using std::cout;
6
7  #include <new> // standard operator new and set_new_handler
8  using std::set_new_handler;
9
10 #include <cstdlib> // abort function prototype
11 using std::abort;
12
13 // handle memory allocation failure
14 void customNewHandler()
15 {
16    cerr << "customNewHandler was called";
17    abort();
18 } // end function customNewHandler
19
20 // using set_new_handler to handle failed memory allocation
21 int main()
22 {
23    double *ptr[ 50 ];
24
25    // specify that customNewHandler should be called on
26    // memory allocation failure
27    set_new_handler( customNewHandler );
28
29    // allocate memory for ptr[ i ]; customNewHandler will be
30    // called on failed memory allocation
31    for ( int i = 0; i < 50; i++ )
32    {
33       ptr[ i ] = new double[ 50000000 ]; // may throw exception
```

```
34       cout << "Allocated 50000000 doubles in ptr[ " << i << " ]\n";
35    } // end for
36
37    return 0;
38 }  // end main
```

```
Allocated 50000000 doubles in ptr[0]
Allocated 50000000 doubles in ptr[1]
Allocated 50000000 doubles in ptr[2]
customNewHandler was called
```

شکل ۷-۱۶ | set_new_handler نشاندهنده فراخوانی تابع در زمان واماندگی new است.

۱۲-۱۶ کلاس auto_ptr و تخصیص حافظه دینامیکی

یکی از رویه‌های معمول در برنامه‌نویسی اخذ یا تخصیص حافظه دینامیکی، تخصیص آدرس آن حافظه به یک اشاره‌گر، استفاده از اشاره‌گر برای دستکاری کردن حافظه و رهاسازی حافظه با **delete** در زمانی است که دیگر به آن حافظه نیازی نداریم. اگر یک استثنا پس از تخصیص موفقیت‌آمیز حافظه رخ دهد اما قبل از اجرای عبارت **delete** باشد، فقدان حافظه می‌تواند اتفاق بیفتد. زبان C++ استاندارد الگوی کلاس **auto_ptr** را در سرآیند فایل <**memory**> تدارک دیده است که به این وضعیت رسیدگی می‌کند.

یک شی از کلاس **auto_ptr** یک اشاره‌گر به حافظه اخذ شده دینامیکی را نگهداری می‌کند. زمانیکه نابود کننده شی **auto_ptr** فراخوانی می‌شود (برای مثال، زمانیکه شی **auto_ptr** از قلمرو خارج می‌شود)، یک عملیات **delete** بر روی اشاره‌گر عضو داده خود انجام می‌دهد. الگوی کلاس **auto_ptr** سربارگذاری عملگرهای * و -> را تدارک دیده است و از اینروست که یک شی **auto_ptr** می‌تواند بعنوان یک متغیر اشاره‌گر عادی بکار گرفته شود. برنامه شکل۱۰-۱۶ به بررسی یک شی **auto_ptr** پرداخته است که به یک شی اخذ شده دینامیکی از کلاس **Integer** اشاره دارد (شکل۸-۱۶ و ۹-۱۶).

خط 18 از شکل ۱۰-۱۶ شی **ptrToInteger** را از **auto_ptr** ایجاد کرده و آنرا با یک اشاره‌گر به شی **Integer** اخذ شده دینامیکی مقداردهی می‌کند که حاوی مقدار 7 است. خط 21 از عملگر سربارگذاری شده -> برای احضار تابع **setInteger** بر روی شی **Integer** مورد اشاره توسط **ptrToInteger** استفاده کرده است. خط 24 از عملگر سربارگذاری شده * برای بازیابی اطلاعات از طریق **ptrToInteger** استفاده کرده است، سپس از عملگر نقطه (.) برای فراخوانی تابع **getInteger** بر روی شی **Integer** مورد اشاره توسط **ptrToInteger** سود برده است همانند یک اشاره‌گر عادی، عملگرهای سربارگذاری شده -> و * می‌توانند در دسترسی به شی که **auto_ptr** به آن اشاره می‌کند، بکار گرفته شوند.

بدلیل اینکه **ptrToInteger** یک متغیر اتوماتیک محلی به **main** است، **ptrToInteger** در زمان خاتمه **main** نابود می‌شود. نابود کننده **auto_ptr** یک **delete** را بر روی شی **Integer** اشاره شده توسط **ptrToInteger** اعمال می‌کند، که در ادامه نابود کننده کلاس **Integer** فراخوانی می‌شود. حافظه‌ای که **Integer** اشغال کرده بود آزاد می‌شود، صرفنظر از اینکه چگونه کنترل از بلوک خارج شده باشد (یعنی

توسط یک عبارت **return** یا توسط یک استثنا). از همه مهمتر، با استفاده از یک تکنیک می‌توان جلوی فقدان حافظه را گرفت.

```
1  // Fig. 16.8: Integer.h
2  // Integer class definition.
3
4  class Integer
5  {
6  public:
7     Integer( int i = 0 ); // Integer default constructor
8     ~Integer(); // Integer destructor
9     void setInteger( int i ); // set Integer value
10    int getInteger() const; // return Integer value
11 private:
12    int value;
13 }; // end class Integer
```

شکل ۸-۱٦ | تعریف کلاس Integer.

```
1  // Fig. 16.9: Integer.cpp
2  // Integer member function definition.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Integer.h"
8
9  // Integer default constructor
10 Integer::Integer( int i )
11    : value( i )
12 {
13    cout << "Constructor for Integer " << value << endl;
14 } // end Integer constructor
15
16 // Integer destructor
17 Integer::~Integer()
18 {
19    cout << "Destructor for Integer " << value << endl;
20 } // end Integer destructor
21
22 // set Integer value
23 void Integer::setInteger( int i )
24 {
25    value = i;
26 } // end function setInteger
27
28 // return Integer value
29 int Integer::getInteger() const
30 {
31    return value;
32 } // end function getInteger
```

شکل ۹-۱٦ | تعریف تابع عضو از کلاس Integer.

```
1  // Fig. 16.10: Fig16_10.cpp
2  // Demonstrating auto_ptr.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <memory>
8  using std::auto_ptr; // auto_ptr class definition
9
10 #include "Integer.h"
11
12 // use auto_ptr to manipulate Integer object
13 int main()
14 {
15    cout << "Creating an auto_ptr object that points to an Integer\n";
16
```

```
17    // "aim" auto_ptr at Integer object
18    auto_ptr< Integer > ptrToInteger( new Integer( 7 ) );
19
20    cout << "\nUsing the auto_ptr to manipulate the Integer\n";
21    ptrToInteger->setInteger( 99 ); // use auto_ptr to set Integer value
22
23    // use auto_ptr to get Integer value
24    cout << "Integer after setInteger: "<<( *ptrToInteger ).getInteger()
25       << "\n\nTerminating program" << endl;
26    return 0;
27 }  // end main
```

```
Creating an auto_ptr object that points to an Integer
Constructor for Integer 7

Using the auto_ptr to manipulate the Integer
Integer after setInteger: 99

Terminating program
Destructor for Integer 99
```

شکل ۱۰-۱۶ | شی auto_ptr تخصیص حافظه دینامیکی را مدیریت می‌کند.

یک **auto_ptr** می‌تواند مالکیت حافظه دینامیکی را که مدیریت می‌کند از طریق عملگر تخصیص سربارگذاری شده خود یا سازنده کپی کننده انتقال دهد. آخرین شی **auto_ptr** که اشاره‌گر به حافظه دینامیکی را نگهداری می‌کند، حافظه را حذف خواهد کرد. این ویژگی **auto_ptr** را تبدیل به مکانیزم مناسبی را برای باز گرداندن حافظه اخذ شده دینامیکی به کد سرویس‌گیرنده کرده است. زمانیکه **auto_ptr** در کد سرویس‌گیرنده از قلمرو خارج می‌شود، نابود کننده **auto_ptr** حافظه دینامیکی را حذف می‌کند.

۱۳-۱۶ سلسله مراتب استاندارد استثنا

تجربه نشان داده است که استثناها بخوبی در یکی از چندین رده تعیین شده قرار می‌گیرند. کتابخانه استاندارد C++ شامل یک سلسله مراتب از کلاس‌هایی استثنا است (شکل ۱۱-۱۶). همانطوری که در ابتدای بخش ۳-۱۶ بیان کردیم، سرسلسله این سلسله مراتب کلاس مبنای **exception** است (تعریف شده در فایل سرآیند <**exception**>)، که حاوی تابع مجازی **what** است که کلاس‌های مشتق شده می‌توانند پیغام‌های مناسب خطا را توسط آن صادر کنند.

بلافاصله پس از کلاس مبنای **exception** کلاس‌های مشتق شده **runtime_error** و **logic_error** قرار دارند (هر دو در سرآیند <**stdexcept**> تعریف شده‌اند)، هر یک از این دو، دارای چندین کلاس مشتق شده هستند. همچنین از کلاس **exception** استثناهای مشتق شده‌اند که توسط عملگرهای C++ به راه می‌افتند، برای مثال، **bad_alloc** توسط **new** (بخش ۱۱-۶)، **bad_cast** توسط **dynamic_cast** (فصل ۱۳) و **bad_typeid** توسط **typeid** (فصل ۱۳) به جریان می‌افتند. منظور از **bad_exception** این است که اگر یک استثنا غیرمنتظره رخ دهد، تابع **unexpected** می‌تواند **bad_exception** را بجای خاتمه دادن به

اجرای برنامه (حالت پیش‌فرض) با فراخوانی تابع دیگر مشخص شده توسط **set_unexpected** راه‌اندازی کند.

شکل ۱۱-۱۶ | کلاس‌های استثنا در کتابخانه استاندارد.

کلاس **logic_error** کلاس مبنا برای چندین کلاس استثنا است که دلالت بر خطا در منطق برنامه دارند. برای مثال، کلاس **invalid_argument** بر این نکته دلالت دارد که یک آرگومان نامعتبر به تابع ارسال شده است. کلاس **length_error** نشان می‌دهد که طول، بیش از حداکثر سایز اجازه داده شده برای آن شی می‌باشد. کلاس **out_of_range** نشان می‌دهد که مقداری همانند شاخص یک آرایه از مرزهای آرایه تجاوز کرده است.

کلاس **runtime_error** که بطور خلاصه در بخش ۸-۱۶ بکار گرفته شد، کلاس مبنا برای چند کلاس استثنا استاندارد است که دلالت بر خطاهای زمان اجرا دارند. برای مثال، کلاس **overflow_error** نشاندهنده خطای سرریز محاسباتی و کلاس‌های **underflow_error** دلالت بر خطای پاریز دارد (یعنی نتیجه یک عملیات عددی کوچکتر از کوچکترین عددی است که می‌توان در کامپیوتر ذخیره کرد).

۱۴-۱۶ تکنیک‌های رسیدگی به خطا

در سرتاسر این فصل به بیان انواع روش‌های مقابله با استثناهای رخ داده پرداختیم. در این بخش بصورت چکیده این تکنیک‌ها و سایر تکنیک‌های مرتبط با رسیدگی به خطا را بیان می‌کنیم:

- نادیده گرفتن استثنا. اگر استثنای رخ دهد، برنامه می‌تواند در مقابل استثنا گرفتار نشده دچار واماندگی شود. این حالت می‌تواند برای محصولات نرم‌افزاری تجاری یا نرم‌افزارهای خاصی که از اهمیت خاص و حیاتی برخوردار هستند، عیب محسوب می‌شود، اما برای نرم‌افزارهای که برای خودتان طراحی می‌کنید، می‌توان برخی از خطاها را نادیده گرفت.
- خاتمه برنامه. البته اینکار جلوی اجرای برنامه را گرفته و از تولید نتایج اشتباه ممانعت بعمل می‌آورد. برای بسیاری از انواع خطاها، این روش مناسب است، بویژه برای خطاهای غیرعظیم **(nonfatal)** که به برنامه اجازه می‌دهند تا اجرای خود را دنبال کند (در صورتیکه برنامه با خطا کار خود را دنبال می‌کند). این استراتژی برای برنامه‌های کاربردی حیاتی مناسب نمی‌باشد. البته بحث منابع در اینجا مهم است. اگر برنامه یک منبع را بدست گرفته باشد، بایستی قبل از اینکه برنامه خاتمه یابد، منبع را رها کند.
- تنظیم شاخص‌های خطا. مشکلی که این روش دارد این است که برنامه نمی‌تواند در تمام وضعیت‌ها مبادرت به تنظیم این شاخص‌های خطا کند.

• تست کردن شرط خطا، ارسال پیغام خطا و فراخوانی **exit** (در <**cstdlib**>) برای ارسال کد خطا متناسب به محیط برنامه.

• استفاده از توابع **setjump** و **longjump**. این توابع کتابخانه‌ای از <**csetjmp**> به برنامه‌نویس امکان می‌دهند تا یک پرش بلادرنگ از عمق یک فراخوانی تودرتو تابع به یک رسیدگی کننده خطا، انجام دهد. بدون استفاده از **setjump** یا **longjump**، برنامه باید چندین برگشت انجام دهد تا از عمق فراخوانی تودرتوی تابع خارج گردد. توابع **setjump** و **longjump** توابع خطرناکی هستند، چرا که مبادرت به باز کردن پشته می‌کنند بدون اینکه نابود کننده‌ها برای شی‌های اتوماتیک فراخوانی شوند. خود همین مسئله می‌تواند مشکلات جدی بدنبال داشته باشد.

• برخی از انواع خطاهای خاص دارای قابلیت‌های اختصاصی در رسیدگی به مشکل هستند. برای مثال، زمانیکه عملگر **new** دچار واماندگی می‌شود، می‌تواند تابع **new_handler** را برای رسیدگی به خطا به اجرا در آورد.