به نام خدا

برنامه‌سازی پیشرفته

آرش شفیعی

مدیریت استثنا

مدیریت استثنا

- مدیریت استثنا [1] به فرایند واکنش‌دادن به استثناها در حین اجرای برنامه گفته می‌شود.
- یک استثنا، یک شرایط غیرعادی است که به واکنش ویژه‌ای نیازمند است.
- یک استثنا در جریان عادی اجرای برنامه وقفه ایجاد کرده و اجرای برنامه را به قسمتی جهت مدیریت استثنا منتقل می‌کند.
- زبان‌های برنامه‌سازی معمولا سازوکارهایی برای مدیریت استثنا فراهم می‌کنند.

[1] exception handling

مدیریت استثنا

- معمولا در زبان سی یا زبان‌هایی که سازوکاری برای مدیریت استثنا ندارند، دو راه حل در برخورد با شرایط غیرعادی وجود دارد.
- راه اول این است که وقتی تابعی با مقادیری روبرو می‌شود که آن مقادیر در حوزهٔ مقادیر عادی نیستند، پیام خطایی صادر کند و برنامه را با استفاده از دستوراتی مانند `exit()` یا `abort()` خاتمه دهد.
- برای مثال در دسترسی به عناصر یک وکتور، اگر دسترسی در محدودهٔ عناصر نباشد، می‌توان پیام خطا صادر کرده و از برنامه خارج شد.

```
۱ double& Vector::operator[](int i) {
۲     if (i<0 || size()<=i) {
۳         cout << "error: out of range access\n" ;
۴         exit();
۵     }
۶     return elem[i];
۷ }
```

مدیریت استثنا

- مشکل این راه حل این است که برنامه را خاتمه می‌دهد، در صورتی که در بیشتر مواقع انتظار داریم برنامه به حیات خود ادامه داده و فرصتی دوباره برای اصلاح خطا به کاربر داده شود.

```
۱ double& Vector::operator[](int i) {
۲     if (i<0 || size()<=i) {
۳         cout << "error: out of range access\n" ;
۴         exit();
۵     }
۶     return elem[i];
۷ }
```

مدیریت استثنا

- راه دوم این است که تابعی که با مقادیر غیرعادی روبرو می‌شود، با استفاده از مقداردهی یک متغیر عمومی وجود خطا را به فراخوانی‌کنندهٔ تابع اعلام کند.
- برای مثال در دسترسی به عناصر یک وکتور، در صورتی که دسترسی در محدودهٔ صحیح نباشد، تابع می‌تواند یک متغیر عمومی را مقداردهی کند.

```
۱ double& Vector::operator[](int i) {
۲     if (i<0 || size()<=i) {
۳         out_of_range = -1; // what about data hiding?
۴         return out_of_range;
۵     }
۶     return elem[i];
۷ }
```

مدیریت استثنا

- در راه حل دوم، اگر بخواهیم یک متغیر عمومی را مقداردهی کنیم، قوانین کپسوله‌سازی و پنهان سازی داده‌ها در برنامه‌سازی شیءگرا را نقض کرده‌ایم. در برنامه‌سازی شیءگرا داده‌ها معمولا توسط اشیا کپسوله‌سازی و پنهان‌سازی شده‌اند و نمی‌توان به آنها به طور مستقیم دسترسی پیدا کرد. پس کسی نمی‌تواند با تغییر دادن یک متغیر عمومی منطق برنامه را به هم بزند. در حالی که در اینجا یک متغیر عمومی تعریف کرده‌ایم که می‌تواند توسط دیگر توابع نیز دستکاری و تغییر داده شود.

```
۱ double& Vector::operator[](int i) {
۲     if (i<0 || size()<=i) {
۳         out_of_range = -1; // what about data hiding?
۴         return out_of_range;
۵     }
۶     return elem[i];
۷ }
```

مدیریت استثنا

- راه سوم این است که تابعی که با مقادیر غیرعادی روبرو می‌شود، با استفاده از مقدار خروجی تابع وجود خطا را به فراخوانی‌کنندهٔ تابع اعلام کند.
- برای مثال در دسترسی به عناصر یک وکتور، در صورتی که دسترسی در محدودهٔ صحیح نباشد، تابع می‌تواند مقداری به عنوان کد خطا بازگرداند.

```
۱ double& Vector::operator[](int i) {
۲     if (i<0 || size()<=i) {
۳         return -1; // what if elem[i] = -1?
۴     }
۵     return elem[i];
۶ }
```

مدیریت استثنا

- مشکل راه حل سوم این است که ممکن است همهٔ مقادیر خروجی یک تابع مقادیر مورد نیاز فراخوانی کنندهٔ تابع باشند و هیچ مقداری را نتوان به عنوان کد خطا اعلام کرد.
- همچنین ممکن است در تابع سازنده یا مخرب با خطایی روبرو شویم، در حالی که تابع سازنده و مخرب مقدار خروجی ندارند.
- از طرف دیگر ممکن است تابع f تابع g و تابع g تابع h را فراخوانی کند و به همین ترتیب یک سلسله فراخوانی توابع اتفاق بیافتد و خطا در آخرین تابع در این سلسله شناسایی شود در حالی که اولین تابع در این سلسله نیاز به مطلع شدن از خطا داشته باشد. به عبارت دیگر ممکن است استفاده کنندهٔ خطا فراخوانی کنندهٔ بلاواسطهٔ تابع دارای خطا نباشد و خطا با چندین واسطه نیاز به انتشار داشته باشد.

```
۱ double& Vector::operator[](int i) {
۲     if (i<0 || size()<=i) {
۳        return -1; // what if elem[i] = -1?
۴     }
۵     return elem[i];
۶ }
```

مدیریت استثنا

- در زبان سی++ سازوکاری برای حل این مشکلات فراهم شده است که مدیریت استثنا نامیده می‌شود.
- هر تابع جدا از مقداری که به عنوان خروجی باز می‌گرداند می‌تواند یک مقدار به عنوان مقدار خطا نیز باز گرداند.
- این مقدار با کلمهٔ کلیدی `throw` به فراخوانی کنندهٔ تابع بازگردانده (یا پرتاب) می‌شود. فراخوانی کنندهٔ تابع با استفاده از خطای دریافت شده می‌تواند تصمیم بگیرد چگونه جریان اجرای برنامه را تغییر دهد.

```
۱ double& Vector::operator[](int i) {
۲     if (i<0 || size()<=i) {
۳         throw 1;
۴     }
۵     return elem[i];
۶ }
```

مدیریت استثنا

- پس یک تابع علاوه بر بازگرداندن مقادیر می‌تواند مقادیری را نیز توسط کلیدواژهٔ throw ارسال کند. این مقادیر می‌توانند از هر نوعی (مانند عدد صحیح، عدد اعشاری، رشته، اشیایی از کلاس‌های تعریف‌شده توسط کاربر، و غیره) باشند.

```
۱ throw 1;
۲ throw -1;
۳ int a=0; string s="error";
۴ char message[50] = "error";
۵ complex c;
۶ throw a; throw s;
۷ throw message;
۸ throw c;
```

مدیریت استثنا

- پس ارسال کنندهٔ استثنا در یک تابع دلخواه f استثنایی را با استفاده از کلیدواژهٔ throw پرتاب می‌کند.

```
<return type> f(<arguments>) {
    // ...
    // x is a variable of any type defined somewhere
    throw x;
    // ...
}
```

مدیریت استثنا

- حال استفاده کنندهٔ تابعی که استثنایی را پرتاب می‌کند، می‌تواند استثا را دریافت کند. دریافت استثنا جهت مدیریت توسط دو کلیدواژهٔ `try` و `catch` صورت می‌گیرد.

- پس از اینکه یک استثنا در یک بلوک `try` دریافت شد، دستورات بعدی در بلوک اجرا نمی‌شوند و اجرا به اولین خط از بلوک `catch` منتقل می‌شود. در بلوک `catch` مقداری که توسط `throw` پرتاب شده است، دریافت می‌شود.

```
۱  try {
۲      // ...
۳      f();
۴      // if an exception is caught,
۵      // next command in try block are not executed.
۶      // ...
۷  } catch(<type> e) {
۸      // variable x thrown in f() is caught here in variable e.
۹      // how to handle exception
۱۰ }
```

مدیریت استثنا

- همچنین می‌توان بلوک `catch` را به صورت `catch(...)` نوشت بدین معنا که نوع استثنایی که فرستاده می‌شود بی‌اهمیت است و تنها گیرندهٔ استثنا باید استثنایی از هر نوعی را مدیریت کند.

```
۱ try {
۲     // ...
۳ } catch(...) {
۴     // handle exception of any type
۵ }
```

مدیریت استثنا

- فرض کنید شیئی از کلاس Vector می‌خواهد توسط عملگر زیرنویس به اعضای وکتور دسترسی پیدا کند و عملگر زیرنویس در کلاس وکتور سربارگذاری شده و در حالات غیرعادی استثنایی پرتاب می‌کند. بدین ترتیب استفاده کننده می‌تواند استثنا را به صورت زیر دریافت و مدیریت کند.

```
۱ Vector v {1, 2, 4};
۲ try {
۳     v[5] = 7;
۴ }
۵ catch (int e) {
۶     if (e==1)
۷         cout << "out of range access\n";
۸ }
```

مدیریت استثنا

- دقت کنید هر گاه استثنایی پرتاب می‌شود، کنترل برنامه به نزدیک‌ترین نقطه‌ای می‌رود که در آن بلوک `catch` قرار دارد.
- برای مثال اگر تابع `f1()` تابع `f2()` را فراخوانی کند و `f2()` تابع `f3()` و به همین ترتیب الی آخر و استثنایی در تابع `fn()` پرتاب شود، و بلوک `catch` در کنار فراخوانی تابع `f1()` قرار داشته باشد، کنترل برنامه به اولین خط از دستورات بلوک `catch` در کنار تابع `f1()` منتقل می‌شود.

```
۱ try {
۲     // ...
۳     f1(); // f1() calls f2(), f2() calls f3(), and so on.
۴     // finally an exception is thrown in fn()
۵     // ...
۶ } catch(...) {
۷     // once fn() throws an exception,
۸     // program control is directed here.
۹ }
```

مدیریت استثنا

- همچنین بعد از اجرای دستورات درون بلوک catch، دستورات بعد از بلوک اجرا می‌شوند.

```
۱  try {
۲      // ...
۳      f1(); // f1() calls f2(), f2() calls f3(), and so on.
۴      // finally an exception is thrown in fn()
۵      // ...
۶  } catch(...) {
۷      // once fn() throws an exception,
۸      // program control is directed here.
۹  }
۱۰ // once commands in catch are executed,
۱۱ // program control is directed here
```

مدیریت استثنا

- این امکان وجود دارد که بعد از پرتاب استثنا در `()fn` یکی از توابعی که منجر به فراخوانی تابع `()fn` شده است، (مثلا `()f3`) استثنا را تا حدی مدیریت کند، و اگر مدیریت در سطح تابع `()f3` به درستی صورت نگرفت، استثنایی پرتاب کند که در سطح `()f1` مدیریت شود.

```
۱ void f3() {
۲     try {
۳         // ...
۴     } catch(...) {
۵         // handle if possible, and if not re-thow an exception
۶         // throw ...
۷     }
۸ }
```

مدیریت استثنا

```
۱  void f3() {
۲      try {
۳          // ...
۴      } catch(...) {
۵          // handle if possible, and if not re-thow an exception
۶          // throw ...
۷      }
۸  }
۹  void main() {
۱۰ try {
۱۱     // ...
۱۲     f1(); // f1() calls f2(), f2() calls f3(), and so on.
۱۳     // finally an exception is thrown in fn()
۱۴     // f3() may catch or may re-throw an exception
۱۵ } catch(...) {
۱۶     // ...
۱۷ }
```

مدیریت استثنا

- این امکان وجود دارد که یک شیء از یک کلاس در زمان رخداد استثنا پرتاب شود.

```
۱  class vector_exception {
۲  private:
۳     int error_code;
۴     int var;
۵     string message;
۶  public:
۷     vector_exception(int e, int v, string s) :
۸         var(v), error_code(e), message(s) { }
۹     string what() {
۱۰        if (error_code==1)
۱۱            return  message + ": access to element " + to_string(v);
۱۲    }
۱۳ };
```

مدیریت استثنا

- با پرتاب یک شیء از کلاس مدیریت استثنا می‌تواند کارامدتر صورت بگیرد، بدین دلیل که اطلاعات بیشتری را می‌توان توسط ارسال کنندهٔ استثنا به دریافت کنندهٔ استثنا انتقال داد.

```
۱  double& Vector::operator[](int i) {
۲      if (i<0 || size()<=i) {
۳          throw vector_exception(1, i, "out_of_range");
۴      }
۵      return elem[i];
۶  }
۷  try {
۸      vector v {1,2,3};
۹      cout << v[5] << endl;
۱۰ } catch(vector_exception e) {
۱۱     cout << e.what() << endl();
۱۲ }
```

مدیریت استثنا

- برخی از انواع استثناها در کتابخانهٔ استاندارد <stdexcept> تعریف شده‌اند.
- به طور مثال استثنای خارج از محدودهٔ عناصر یک ظرف با عنوان استثنای out_of_range تعریف شده است که می‌توانیم از آن استفاده کنیم.

```
۱ double& Vector::operator[](int i) {
۲     if (i<0 || size()<=i)
۳         throw out_of_range{"Vector::operator[]"};
۴     return elem[i];
۵ }
```

مدیریت استثنا

- پس می‌توانیم از کلاس استثناهای از پیش تعریف شده استفاده کنیم.

```
۱  void f(Vector& v) {
۲      // ...
۳      try { // exceptions here are handled by the handler defined below
۴          v[v.size()] = 7; // try to access beyond the end of v
۵      } catch (out_of_range& err) {
۶          // out_of_range error
۷          cerr << err.what() << '\n';
۸      }
۹      // ...
۱۰ }
```

مدیریت استثنا

- کلاس‌های تعریف شده در کتابخانهٔ استاندارد `stdexcept` دارای یک سلسله مراتب هستند. به طور مثال کلاس `out_of_range` زیرکلاس `logic_error` می‌باشد و `logic_error` زیرکلاس `exception` است.
- کاربر می‌تواند از این کلاس‌های از پیش تعریف شده به ارث ببرد و کلاس جدیدی به عنوان زیرکلاس یکی از این کلاس‌ها تعریف کند.

مدیریت استثنا

- در صورتی که در تعریف یک تابع از کلیدواژهٔ noexcept استفاده کنیم، تابع استثنایی ارسال نخواهد کرد.

```
۱ void user(int sz) noexcept {
۲     Vector v(sz);
۳     iota(&v[0],&v[sz],1);
۴     // ...
۵ }
```

مدیریت استثنا

- در سازندهٔ کلاس وکتور در صورتی که وکتور با یک عدد منفی ساخته شود و یا تخصیص حافظه به درستی صورت نگیرد نیز می‌توان یک استثنا ارسال کرد.

```
۱  Vector::Vector(int s) {
۲      if (s < 0)
۳          throw length_error{"Vector constructor: negative size"};
۴      elem = new double[s]; // new may also throw an exception
۵      sz = s;
۶  }
۷  void test() {
۸      try { Vector v(-27); } catch (std::length_error& err) {
۹          // handle negative size
۱۰     } catch (std::bad_alloc& err) {
۱۱         // handle memory exhaustion
۱۲     }
۱۳ }
```

مدیریت استثنا

- تابع test می‌تواند برخی از استثناها را خود مدیریت کند و مابقی استثناها را ارسال کند تا توابع فراخوانی‌کنندهٔ تابع test آنها را مدیریت کنند.

- در اینجا در صورتی که وکتور به درستی مقداردهی اولیه نشود، یک پیام خطا چاپ می‌شود و یک استثنا به تابع فراخوانی کنندهٔ test ارسال می‌شود، اما در صورتی که حافظه به درستی تخصیص داده نشود، برنامه متوقف می‌شود، چرا که این برنامه برای چنین استثناهایی هیچ تدارکی ندیده است.

```
۱ void test() {
۲     try { Vector v(-27); } catch (std::length_error&) {
۳         cerr << "test failed: length error\n"; // print the error
۴         throw; // rethrow
۵     } catch (std::bad_alloc&) {
۶         // this program is not designed to handle memory exhaustion
۷         std::terminate(); // terminate the program
۸     }
۹ }
```

مدیریت استثنا

- استثناها در زمان اجرای برنامه تشخیص داده می‌شوند. برخی از خطاها را می‌توان در زمان کامپایل تشخیص داد. برای این کار از `static_assert` استفاده می‌کنیم.
- `static_assert` یک شرط را در اولین ورودی خود می‌گیرد. در صورتی که شرط برقرار نبود، پیام خطای دومین ورودی خود را صادر می‌کند.
- برای مثال در زمان کامپایل می‌توان تشخیص داده اگر نوع دادهٔ عدد صحیح در سیستمی که کد بر روی آن اجرا می‌شود ۲ بایتی است و یا ۴ بایتی. می‌خواهیم در صورتی که نوع عدد صحیح ۲ بایتی است، پیام خطا صادر کنیم که به صورت زیر عمل می‌کنیم. `assert` استفاده می‌کنیم.

```
1 static_assert(4<=sizeof(int), "integers are too small");
```

مدیریت استثنا

- همچنین مقادیر ثابت را می‌توان در زمان کامپایل بررسی کرده، بر روی آنها قید گذاشت و در صورتی که قید برقرار نبود، پیام خطا صادر کرد.

```
۱ constexpr double C = 299792.458;// km/s
۲ void f(double speed) {
۳     constexpr double local_max = 160.0/(60*60);
۴     // 160 km/h == 160.0/(60*60) km/s
۵     static_assert(speed < C, "can't go that fast");
۶     // error : speed must be a constant
۷     static_assert(local_max < C, "can't go that fast"); // OK
۸     // ...
۹ }
```