5-Error Handing and Exceptions

For functions that interact with data external to the program such as user input, files, or other forms of I/O, it is unrealistic to assume that the data will always be available and valid. Instead, we need to specify well-defined behavior for what happens when the requirements for such a function are violated.

In writing a complex program, we subdivide it into individual functions for each task. Each function only knows how to handle its own job, and it returns results to its caller. As a result, it generally is not the case that the function that detects an error is equipped to handle it.

考虑如下例子：

```
1  // EFFECTS: Opens the file with the given filename and returns the
2  //          contents as a string.
3  string read_file_contents(const string &filename) {
4      ifstream fin(filename);
5      if (!fin.is_open()) {
6          ? ? ? // file did not open; now what?
7      }
8      ...
9  }
```

`read_file_contents` 的认为是从一个文件中读取数据，它对程序的其它部分一无所知，因此它无法决定在错误发生时应该采取什么措施。取决于整个程序，可能是仅仅打印一条错误消息并退出，提示用户输入新的文件名，忽略给定的文件并继续执行，等等。此外，`read_file_contents` 函数可能在几种不同的上下文中被使用，每种上下文都可能需要不同的错误恢复形式。所以，`read_file_contents` 唯一的责任就是检测并传达发生了错误，而调用栈中更上层的函数则应执行必要的操作来处理该错误。

因此，我们需要一种机制，将错误检测与错误处理分离开来，同时提供一种方法来通知调用者发生了错误。下面我们将探讨几种不同的策略来实现这一点。

1. Global Error Codes

A common strategy in the C language is to set the value of a global variable to an error code, which provides information about the type of error that occurred. For example, many implementations of C and C++ functions use the `errno` global variable to signal errors.

考虑如下例子：

```cpp
#include <cerrno>
#include <cstdlib>
#include <iostream>

using std::cout;
using std::endl;

int main(int argc, char **argv) {
    char *end;
    errno = 0; // reset errno before call
    double number = std::strtod(argv[1], &end);
    if (errno == ERANGE) { // check errno after call
        cout << "range error" << endl;
    } else {
        cout << number << endl;
    }
}
```

当 `strtod` 被给一个超出 `double` 范围的数字， `strtod` 就将 `errno` 设置为 `ERANGE`：

```
./main.exe 1e1000
range error
```

The strategy of setting a global error code can be (pardon the pun) error-prone. For example, `errno` must be set to zero before calling a standard-library function that uses it, and the caller must also remember to check its value after the function call returns. Otherwise, it may get overwritten by some other error down the line, masking the original error.

2. Object Error Codes

🥇 Another strategy used by C++ code is to set an error state on an object. This avoids the negatives of a global variable, since each object has its own error state rather than sharing the same global variable. C++ streams use this strategy:

```
#include <fstream>
#include <iostream>

int main(int argc, char **argv) {
    if (argc < 2) {
        std::cerr << "Usage: " << argv[0] << " <filename>" << std::endl;
        return 1;
    }

    std::ifstream input(argv[1]);
    if (!input) {
        std::cerr << "Error opening file: " << argv[1] << std::endl;
        return 1;
    }

    std::cout << "File opened successfully." << std::endl;
    return 0;
}

```

🥇 As with a global variable, the user must remember to check the error state after performing an operation that may set it.

3. Return Error Codes

🥇 Return error codes are another strategy used in C and C++ code to signal the occurrence of an error. In this pattern, a value from a function's return type is reserved to indicate that the function encountered an error. The following is an example of a `factorial()` function that uses this strategy:

```
// EFFECTS: Computes the factorial of the given number. Returns -1
//          if n is negative.
int factorial(int n) {
    if (n < 0) {
        return -1; // error
    } else if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

🥇 As with a global error code or an object state, it is the responsibility of the caller to check the code to determine whether an error occurred. Furthermore, a return error code only works if there is a value that can be reserved to indicate an error. For a function such as `atoi()`, which converts a C-style string to an int, there is no such value. In fact, `atoi()` returns 0 for both the string `"0"` and `"hello"`, and the caller cannot tell whether or not the input was erroneous.

4. Exceptions

🥇 All three strategies above have the problem that the caller of a function must remember to check for an error; neither the compiler nor the runtime detect when the caller fails to do so. Another common issue is that error checking is interleaved with the regular control flow of the caller, as in the following:

```
int main(int argc, char **argv) {
    double number = std::stod(argv[1]);
    string input;
    cin >> input;
    if (!input) {
        cout << "couldn't read input" << endl;
    } else if (input == "sqrt") {
        cout << std::sqrt(number) << endl;
    } else if (input == "log") {
        cout << std::log(number) << endl;
    }
    ...
}
```

🏅 The control flow for the error case is mixed in with that of non-error cases, making the code less readable and harder to maintain. Furthermore, the code above fails to check for errors in `sqrt()` or `log()`, which is not immediately clear due to the interleaving of control flow.

What we want is a mechanism for error handling that:

- Separates error detection from error handling, providing a general means for signaling that an error occurred.
- Separates the control flow for error handling from that of non-erroneous cases.
- Detects when an error is not handled, which by default should cause the program to exit with a meaningful message.

The mechanism provided by C++ and other modern languages is *exceptions*.

下面就是一个使用异常的例子：

```
1  // Represents an exception in computing a factorial.
2  class FactorialError {};
3
4  // EFFECTS: Computes the factorial of the given number. Throws a
5  //          FactorialError if n is negative.
6  int factorial(int n) {
7      if (n < 0) {                // error case
8          throw FactorialError(); // throw an exception
9      }
10     if (n == 0) { // non-error case
11         return 1;
12     } else {
13         return n * factorial(n - 1);
14     }
15 }
16
17 int main(int argc, char **argv) {
18     try {
19         int n = std::stoi(argv[1]);
20         int result = factorial(n);
21         cout << n << "! = " << result << endl;
22     } catch (const std::invalid_argument &error) {
23         cout << "Error converting " << argv[1] << " to an int: "
24              << error.what() << endl;
25     } catch (const FactorialError &error) {
26         cout << "Error: cannot compute factorial on negative number"
27              << endl;
```

```
    }
}
```

The individual components of the exception mechanism used above are:

- The `FactorialError` class is an exception type, and objects of that type are used to signal an error.
- When the `factorial()` function detects an error, it *throws* an exception object using the `throw` keyword. In the example above, the function throws a default-constructed `FactorialError`. The standard-library `stoi()` function throws a `std::invalid_argument` object when given a string that does not represent an integer.
- The compiler arranges for the exception to propagate outward until it is handled by a *try/catch* block. In the example above, the first catch block is executed when a `std::invalid_argument` is thrown, while the second is run when a `FactorialError` is thrown.

When an exception is thrown, execution pauses and **can only resume at a catch block**. The code that is between the throw and the catch that handles the exception is entirely skipped.

4.1 Exception Objects

异常对象表示一种异常。

C++ allows an object of any type to be thrown as an exception. However, a proper exception carries information about what caused the error, and using exceptions of different types allows a program to distinguish between different kinds of errors. More specifically, the example above illustrates that a program can perform different actions in response to each exception type.

There are several exceptions defined by the standard library, such as `invalid_argument` or `out_of_range`. The standard-library exceptions all derive from `std::exception`, and it is common for user-defined exceptions to do so as well:

```
class EmailError : public std::exception {
  public:
    EmailError(const string &msg_in) : msg(msg_in) {}
```

```
4
5       const char *what() const override {
6           return msg.c_str();
7       }
8
9     private:
10      string msg;
11  };
```

> 🥇 A user-defined exception is a class type, so it can carry any information necessary to pass between the function that detects an error and the one that handles it.

比如上述的 `EmailError`，它的构造函数接受一个字符串并将其存储到成员变量 `msg` 中，这样在创建 `EmailError` 对象时可以指定任何消息：

```
1   throw EmailError("Error sending email to:" + address);
```

`what()` 成员函数是由 `std::exception` 定义的虚函数，`EmailError` 重写了它以返回传递给构造函数的错误信息。我们可以在捕获异常时调用它来获取错误信息：

```
1   try {
2       ...
3   } catch (const EmailError &error) {
4       cout << error.what() << endl;
5   }
```

4.2 Try/Catch Blocks

> 🥇 A try/catch block consists of a try block followed by one or more catch blocks:

```
1   try {
2       int n = std::stoi(argv[1]);
3       int result = factorial(n);
4       cout << n << "! = " << result << endl;
5   } catch (const std::invalid_argument &error) {
6       cout << "Error converting " << argv[1] << " to an int: "
7            << error.what() << endl;
```

```
} catch (const FactorialError &error) {
    cout << "Error: cannot compute factorial on negative number"
         << endl;
}
```

The try/catch can only handle exceptions that occur within the try part of the block, including in functions called by code in the try.

When a thrown exception propagates to a try block, the compiler checks the corresponding catch blocks in order to see if any of them can handle an exception of the type thrown. Execution is immediately transferred to the first applicable catch block:

- The catch parameter is initialized from the exception object. Declaring the parameter as a reference to const avoids making a copy.
- The body of the catch block is run.
- Assuming the catch block completes normally, execution proceeds past the entire try/catch. The remaining code in the try or in the other catch blocks is skipped.

In matching an exception object to a catch block, C++ takes into account subtype polymorphism – if the dynamic type of the object is derived from the type of a catch parameter, the catch is able handle that object, so the program uses that catch block for the exception.

C++ 还提供了一个“捕获所有”（catch-all）的异常处理机制，可以处理任何类型的异常：

```
try {
    // some code here
} catch (...) {
    cout << "Caught an unknown error" << endl;
}
```

The `...` as a catch parameter enables the catch block to handle any type of object. However, this should in general be avoided – instead, a particular try/catch should only catch the specific kinds of errors that it is able to recover from.

4.3 Exception Propagation

When an exception is thrown, if the current function is not within a try block, the exception propagates to the function's caller. Similarly, if the current function is within a try block but there is no associated catch block that can handle an object of the exception's type, the exception also propagates to the caller.

To handle an exception, the program immediately pauses execution, then looks for an exception handler as follows. The process starts at the statement that throws the exception:

- Determine if the current statement is within the try part of a try/catch block in the current function. If not, the function call is terminated, and the process repeats in the caller.
- If the execution is within a try, examine the catch blocks in order to find the first one that can handle an exception of the given type. If no matching catch is found, the function call is terminated, and the process repeats in the caller.
- If a matching catch block is found, the catch parameter is initialized with the exception object, and the body of the catch immediately runs. Assuming it completes successfully, execution proceeds past the entire try/catch.

As described above, if the code is not within a try block that can handle the exception object, execution returns immediately to the caller, and the program looks for a handler there. The exception propagates outward until a viable handler is found.

If the exception proceeds outward past `main()`, the default behavior is to terminate the program. This ensures that either the exception is handled, or the program crashes, informing the user that an error occurred.

5. Error Handing vs. Undefined Bahavior

The error-detection mechanisms we discussed provide well-defined behavior in case of an error. As such, a function that uses one of these mechanisms should document it, describing when and what kind of error can be generated:

```
1  // EFFECTS: Computes the factorial of the given number. Returns 0 if
2  //          n is negative.
3  int factorial(int n);
4
5  // EFFECTS: Emails the given student the given grade. Throws
6  //          EmailError if sending the email fails.
7  void email_student(const string &name, double grade);
```

🎖️ These functions do not have REQUIRES clauses that restrict the input; violating a REQUIRES clause results in undefined behavior, whereas these functions produce well-defined behavior for erroneous input.

On the other hand, when code does produce undefined behavior, it cannot be detected through any of the error-handling mechanisms above. For example, dereferencing a null or uninitialized pointer does not necessarily throw an exception, set a global error code, or provide any other form of error detection. It is the programmer’ s responsibility to avoid undefined behavior, and there are no constraints on what a C++ implementation can do if a program results in undefined behavior.