Recovering from errors

Raul P. Pelaez

November 13, 2024

Contents

1 Introduction

Instead of painstakingly building up our applications from the built-in types (e.g., `char`, `int`, `double`, etc.) and statements (e.g., `if`, `while`, `for`, etc.), we build types (e.g., `string`, , and `thread`) and algorithms (e.g., `sort`, `find` and `draw_all`) that are appropriate form our applications. Such high-level constructs simplify our programming, limit our opportunities for mistake (e.g. you are unlikely to try to apply a tree traversal to a dialog box), and increase the compiler's chances of catching errors. The majority of C++ language constructs are dedicated to the design and implementation of elegant and efficient abstractions (e.g., user-defined types and algorithms using them). One effect of using such abstractions is that the point where a run-time error can be detected is separated from the point where it can be handled. As programs grow, and especially when libraries are used extensively, standards for handling errors become important. It is a good idea to articulate a strategy for error handling early on in the development of a program[1].
In C++, this strategy is based on exceptions. Exceptions are a mechanism for handling errors and other exceptional conditions that arise during the execution of function/piece of code.

2 Exception Handling Basics

Exception handling allows a program to deal with unexpected situations and errors gracefully. It separates error-handling code from regular code. For instance, as part of their execution many STL algorithms will try to allocate some memory, and if that fails, they will throw an exception. We do not have access to the implementation of these algorithms, nor do we care about it. Luckily, the exception that will be thrown gives us the information we need and a chance to handle the error.

[1] This excerpt is taken from "A Tour of C++ 3rd ed." by Bjarne Stroustrup.

2.1 Basic keywords: `try`, `catch`, `throw`

C++ uses three primary keywords for exception handling:

- `try`: Defines a block of code to monitor for exceptions.
- `catch`: Catches exceptions thrown by the 'try' block.
- `throw`: Throws an exception.

Most std library functions throw exceptions when errors occur, such as `std::invalid_argument` or `std::runtime_error`. The documentation for each function specifies the exceptions it can throw and the conditions under which they are thrown.
Here's the basic syntax:

Source code

```
try {
  // Code that might throw an exception
} catch (ExceptionType1& e1) {
  // Handle exception of type ExceptionType1
} catch (ExceptionType2& e2) {
  // Handle exception of type ExceptionType2
} catch(...) {
  // Catch all other exceptions
}
```

You can throw an exception using the `throw` keyword followed by an exception object:

Source code

```
if (error_condition) {
  throw std::runtime_error("An error occurred");
}
```

Catch blocks handle exceptions. You can catch exceptions by reference to avoid object slicing and to allow polymorphic behavior:

Source code

```
try {
  // Code that may throw
} catch (const std::exception& e) {
  std::cerr << "Exception caught: " << e.what() << '\n';
}
```

Info

Example: Division by Zero

Source code

```cpp
#include <iostream>
#include <stdexcept>

int divide(int a, int b) {
  if (b == 0) {
    throw std::runtime_error("Division by zero");
  }
  return a / b;
}

int main() {
  try {
    std::cout << divide(10, 0) << '\n';
  } catch (const std::runtime_error& e) {
    std::cerr << "Error: " << e.what() << '\n';
  }
  return 0;
}
```

Info

Example: Vector Access

Source code

```cpp
#include <iostream>
#include <stdexcept>
#include <vector>

int main() {
  std::vector v = {1, 2, 3};
  try {
    std::cout << v.at(3) << '\n'; // Access out of bounds
  } catch (const std::out_of_range& e) {
    // Also works if we catch std::exception& e
    std::cerr << "Error: " << e.what() << '\n';
  }
  return 0;
}
```

Warning

Exceptions are designed exclusively to be used to report failure to complete a given task that the code in question is not able to handle on its own. Never use exceptions for normal control flow or to return values.

2.2 Standard Exceptions

C++ provides a hierarchy of standard exceptions in the <exception> and <stdexcept> headers. You can also create custom exceptions by deriving from `std::exception`.

Info

Standard Exception Hierarchy

- `std::exception`: Base class for all standard exceptions.
  - `std::logic_error`: Errors in program logic.
    - `std::invalid_argument`
    - `std::length_error`
    - `std::out_of_range`
  - `std::runtime_error`: Errors detectable only during runtime.
    - `std::range_error`
    - `std::overflow_error`
    - `std::underflow_error`

Refer to the C++ documentation for a complete list of standard exceptions.

Advice

`std::runtime_error` is the most common exception type that one throws and receives.

2.3 Stack Unwinding

When an exception is thrown, C++ performs **stack unwinding**, which involves calling destructors for all objects in scope between the throw point and the catch block.
Stack unwinding ensures that resources are properly released when an exception occurs (if all entities inside the offending code follow RAII, that is). Destructors of local objects are called in reverse order of their construction.

3 The `noexcept` specifier

The `noexcept` specifier is used to indicate that a function does not throw exceptions. This can help the compiler optimize code and provide information about the function's behavior.

Advice

Design errors out of existence
It is a good idea to strive for functions and designs that just cannot throw exceptions, not because we simply ignore error checking, but because we design our interfaces and our code so that certain errors are impossible.
For instance, instead of writing this:

Source code

```
void f(int i) {
  if (i < 0) throw std::runtime_error("f(): negative argument");
  // ...
}
```

we might write this:

Source code

```
void f(unsigned int i) noexcept {
  // Now it is impossible for this function to receive a negative argument
  // ...
}
```

A prime example is dealing with pointers:

Source code

```
void f(const char* p) {
  if (!p) throw std::runtime_error("f(): null pointer");
  // ...
}
```

versus:

Source code

```
void f(std::string p) noexcept {
  // The string might be empty, but we designed the "null pointer" error
  ↪  check out of existence
  /// This code will simply not do anything if the string is empty
  //for(char c: p) { /* ... */ }
  // ...
}
```

4 Creating Custom Exceptions

You can define your own exceptions by inheriting from ‘std::exception‘ or any of its subclasses:

Source code

```
#include <exception>
#include <string>

class MyException : public std::exception {
  std::string message;
public:
  explicit MyException(const std::string& msg) : message(msg) {}
  const char* what() const noexcept override {
    return message.c_str();
  }
};
```

In the example above, `MyException` is a custom exception class that inherits from `std::exception` (we could have also inherited from something more specific like `std::runtime_error` or even just a custom class). The `what()` method returns the exception message.

Advice

Note that the method is marked as `noexcept`, indicating that it does not throw exceptions. It would be unwise to throw exceptions from the `what()` method for several reasons:

- The standard states that if an exception is thrown during stack unwinding, the program will terminate. Otherwise we could risk an infinite loop of exceptions being thrown and caught (if the what method is called during stack unwinding).
- No one expects the what method to throw an exception, since `std::exception::what` is marked `noexcept`.
- Handling the what function throwing would require us to catch exceptions in the catch block, which results in very ugly code.

By these same arguments, it is also a good idea to avoid throwing exceptions from destructors, in general marking them as `noexcept`.

5 Rethrowing Exceptions

You can rethrow the current exception using ‘throw;‘ inside a catch block:

Source code

```
try {
  // Code that may throw
} catch (const std::exception& e) {
  // Perform some handling or logging
  std::cerr << "Caught exception: " << e.what() << '\n';
  throw; // Rethrow the exception
}
```

By rethrowing, you can propagate the exception to a higher level in the call stack for further handling.

Info

If an exception reaches the top level of the program without being caught, the program terminates.

Info

Example: Rethrowing Exceptions

Source code

```
#include <iostream>
#include <stdexcept>

void func() {
  try {
    throw std::runtime_error("Error in func");
  } catch (...) {
    // I can use rethrowing to log the exception and propagate it
    std::cerr << "Caught exception in func, rethrowing...\n";
    throw; // Rethrow exception
  }
}

int main() {
  try {
    func();
  } catch (const std::exception& e) {
    std::cerr << "Caught exception in main: " << e.what() << '\n';
  }
  return 0;
}
```

Warning

Never throw exceptions from destructors. If a destructor throws during stack unwinding, ‘std::terminate()‘ is called, leading to program termination.

Advice

In well designed code try-catch blocks are rare. Avoid overuse by systematically using the RAII technique.

6 Alternatives to exceptions

Throwing an exception is not the only way of reporting an error that cannot be handled locally. A function can indicate that it cannot perform its task by:

- Throwing an exception
- Somehow returning a value indicating failure

- Terminating the program (by calling `std::terminate()`)

We return an error code when:

- A failure is normal and expected. For instance, it is normal that a request to open a file fails (the file does not exists, not enough permissions,...).
- An immediate caller can reasonably be expected to handle the failure.

We throw an exception when:

- A failure is so rare that the programmer is likely to forget to check for it. For instance the system running out of memory, or printf failing.
- The caller cannot reasonably be expected to handle the failure, it must be propagated up the call stack to the "ultimate caller".
- Returning an error code is somehow syntactically impractical (e.g., because the function already returns a value).

We terminate the program when:

- An error is irrecoverable. In most systems one cannot really do anything useful after running out of memory.
- Handling the error would be more complex than just restarting the application.

Advice

Error-handling is a very nuanced topic. The boundaries between the three methods are not always clear-cut and depend on the context. The most important thing is to design your code to reduce complexity. If you find yourself writing a lot of error-handling code, it might be a sign that your design is flawed and you should be **designing errors out of existence**.

7 Exercises

Goal

Write a cycles bot.
The server and client codes are provided in this repo:
`https://github.com/RaulPPelaez/cycles`

hint

See the documentation for how to write a bot here.

Milestone

Clone the repository and manage to build the server and example client (randomio).

hint

You will probably have to fight with some quirk or another of your system.
The documentation contains some known issues and solutions.

Milestone

Copy the randomio client and integrate it into the build scripts. Call it client_[your_name].cpp. Build it and modify your run script to run an instance of it along with some randomios.

Milestone

Rewrite the bot logic to make it play better, your bot should be able to beat the randomio bot.

Milestone

Add error handling to your bot's code.
Some parts of your logic are subject to failing, make sure to catch those errors and handle them gracefully.
Think about "defining errors out of existence" and decide whether potential errors should throw and exception, return an error code or just terminate the program.

hint

The network code is particularly error-prone due to its asynchronous nature. However, I wrote the code to either "define errors out of existence" or terminate the program if they are not recoverable. I do not think any API function can throw an exception.